Model Based Synthesis and Validation of High Performance Supervisory Controllers

`Distributed BLDC Control Network`

Stavros Martini

Supervisor: Prof. George Kornaros



Department of Electrical and Computer Engineering

Division of Electronics, Systems and Computer Technology

Hellenic Mediterranean University

May 2026

Submitted in partial fulfilment of the requirements for the degree of Integrated Master (B.Sc. & M.Sc.) in Electrical and Computer Engineering

Abstract

Coordinating logical decision-making with real-time physical control is one of the toughest challenges in embedded systems. Typically, the discrete logic and continuous control are developed separately using different tools, then manually stitched together. Most logic errors happen during this manual integration step. This thesis proposes a two-layer architecture based on formal supervisory control synthesis to solve this problem.

The upper layer uses the Eclipse ESCET toolkit and CIF specification language. Each subsystem is modeled as an Extended Finite Automaton (EFA). Safety and coordination requirements are written as formal constraints, and ESCET automatically synthesizes a supervisor. The resulting supervisor has mathematical guarantees: it is controllable, non-blocking, and maximally permissive. Before code generation, we verify additional properties like bounded response time and confluence.

The lower layer handles real-time control. Each motor node runs field-oriented control (FOC) using the SimpleFOC library at 20 kHz, completely independent of the network supervisor. Communication between layers happens through a minimal, formally-defined interface: CIF input variables receive sensor data that trigger uncontrollable events, while controllable events generate callbacks that send actuator commands to local nodes. Importantly, local nodes handle their own safety reactions without waiting for supervisor approval.

We tested the architecture on a physical two-node BLDC motor network connected via CAN 2.0B, implementing a Heat Recovery Ventilation (HRV) system. The application required 46 formal requirements covering coordinated startup, sequential calibration, fault recovery with bounded retries, and strict "both or nothing" fan operation. We injected electrical faults, thermal faults, and power loss on live hardware. In all cases, the supervisor behaved exactly as the formal model predicted—no deadlocks, no requirement violations.

The proposed approach drastically reduces logical integration errors and provides a commercially viable solution from the formal specifications to the embedded code.

Keywords: Embedded System Architecture, ESCET, CIF, Supervisory Control, Real-time Control, Embedded Systems

Σύνοψη Διπλωματικής Εργασίας

Η συντονισμένη λειτουργία διακριτής λογικής λήψης αποφάσεων με συνεχή έλεγχο σε πραγματικό χρόνο αποτελεί μία από τις μεγαλύτερες προκλήσεις στα ενσωματωμένα συστήματα. Κανονικά, η διακριτή λογική και ο συνεχής έλεγχος αναπτύσσονται ξεχωριστά με διαφορετικά εργαλεία και ενσωματώνονται χειροκίνητα, όπου προκύπτουν τα περισσότερα λογικά σφάλματα. Η παρούσα εργασία προτείνει διμερή αρχιτεκτονική βασισμένη στη τυπική σύνθεση επιβλέποντος ελεγκτή με χρήση του Eclipse ESCET toolkit.

Στο ανώτερο επίπεδο, το ESCET μοντελοποιεί κάθε υποσύστημα ως Extended Finite Automaton (EFA) στη γλώσσα CIF. Οι απαιτήσεις ασφαλείας και συντονισμού εκφράζονται ως τυπικοί περιορισμοί, και το ESCET συνθέτει αυτόματα επιβλέποντα ελεγκτή με μαθηματικές εγγυήσεις: ελεγχιμότητα, μη αποκλεισιμότητα και μέγιστη επιτρέψιμη συμπεριφορά. Πριν την παραγωγή κώδικα, επαληθεύονται πρόσθετες ιδιότητες, όπως περιορισμένος χρόνος απόκρισης.

Στο κατώτερο επίπεδο, κάθε κινητήρας εκτελεί field-oriented control (FOC) με τη βιβλιοθήκη SimpleFOC με συχνότητα 20 kHz, πλήρως ανεξάρτητα από τον δικτυακό επιβλέποντα ελεγχτή. Η επικοινωνία μεταξύ επιπέδων πραγματοποιείται μέσω τυπικά ορισμένης διεπαφής: οι εισόδοι CIF λαμβάνουν δεδομένα αισθητήρων που πυροδοτούν μη ελεγχόμενα γεγονότα, ενώ τα ελεγχόμενα γεγονότα παράγουν συναρτήσεις επανακάλεσης (callbacks) για αποστολή εντολών στους τοπικούς κόμβους. Οι κόμβοι διαχειρίζονται αυτοτελώς τις αντιδράσεις ασφαλείας χωρίς αναμονή έγκρισης από τον επιβλέποντα ελεγχτή.

Η αρχιτεκτονική δοκιμάστηκε σε φυσικό δίκτυο δύο κόμβων BLDC κινητήρων συνδεδεμένων σε δίκτυο CAN 2.0B, υλοποιώντας σύστημα Αερισμού με Ανάκτηση Θερμότητας (Heat Recovery Ventilation). Η εφαρμογή απαιτούσε 46 τυπικές προδιαγραφές για συντονισμένη εκκίνηση, διαδοχική βαθμονόμηση, ανάκαμψη βλαβών με περιορισμένους επαναληπτικούς κύκλους και αυστηρή λειτουργία «όλα ή τίποτα» στους ανεμιστήρες. Εισήχθησαν ηλεκτρικές βλάβες, θερμικές βλάβες και διακοπή τροφοδοσίας εν λειτουργεία. Σε όλες τις περιπτώσεις, ο επιβλέπων ελεγκτής συμπεριφέρθηκε ακριβώς όπως προέβλεπε το τυπικό μοντέλο, χωρίς αποκλεισμούς ή παραβιάσεις προδιαγραφών.

Η προτεινόμενη προσέγγιση μειώνει δραστικά τα λογικά σφάλματα ενσωμάτωσης και παρέχει εμπορικά βιώσιμη λύση από τις τυπικές προδιαγραφές έως τον ενσωματωμένο κώδικα.

Λέξεις κλειδιά: Αρχιτεκτονική Ενσωματωμένων Συστημάτων, ESCET, CIF, Επιβλέπων Έλεγχος, Έλεγχος Πραγματικού Χρόνου, Ενσωματωμένα Συστήματα

Contents

Abstract i
Σύνοψη Διπλωματικής Εργασίας ii
1 Introduction 1
1.1 Industrial Control Systems 1
1.2 Synthesis-Based Engineering 5
1.3 Research Objectives and Contributions 6
1.4 Thesis Organization 7
2 Theoretical Foundations 8
2.1 Discrete Event System 8
2.2 Supervisory Control Theory 10
2.3 Extended Finite Automata 16
2.4 Mathematical Verification and Guarantees 20
3 Synthesis-Based Engineering with ESCET 21
3.1 The Synthesis-Based Engineering Workflow 21
3.2 CIF Language Fundamentals 22
3.3 Plant and Requirement Modeling 25
3.4 Supervisor Synthesis 30
3.5 Simulation and Verification 30
3.6 Code Generation 33
4 Evaluation: *Distributed BLDC Control Network* 39
4.1 System Overview 39
4.2 Motor Drive Node 43
4.3 Application Case: Heat Recovery Ventilation System 60
4.4 Modeling 63
4.5 Results 73
5 Conclusions 79
5.1 Summary 79
5.2 Contributions 79
5.3 Limitations 80
5.4 Future Work 81
5.5 Closing Remarks 81
Bibliography 83
6 Appendices 85
6.1 Appendix A: Node Plant Model 85
6.2 Appendix B: Coordinator Model 89
6.3 Appendix C: Net Definition and Requirements 91

List of Figures

Figure 1 Block diagram of a Motor Drive System [1] 1
Figure 2 Automated manufacturing system (Image source: Pixabay) 1
Figure 3 Industrial process control system. source: Pixabay 2
Figure 4 ASML High NA EUV lithography machine. Source:ASML Media Library © ASML
..... 2
Figure 5 Traditional view of a control system (source: Eclipse ESCET documentation) . 4
Figure 6 General DES state-machine topology. 9
Figure 7 Example System Plant 11
Figure 8 R_1: Machine start permitted only when power supply is On. 13
Figure 9 R_2: Power supply turn_off permitted only when machine is Off. 13
Figure 10 R_3: Controlled stop requires active power supply. 14
Figure 11 This graph shows a state space illustration of a plant composed of 6 automata with 4 states each. $(k^n = 4^6 = 4096)$ 16
Figure 12 Synthesis-Based Workflow, source: Escet doc [2] 21
Figure 13 Sensor-Actuator system. Sensor (left) has uncontrollable events representing physical signal transitions. Actuator (right) has controllable events guarded by Sensor state. Dashed arrows: uncontrollable; solid arrows: controllable. 23
Figure 14 CIF simulator interface showing current state (locations) and events that can be enabled. Green buttons indicate controllable events, red indicates uncontrollable events. Users select events to execute, observing resulting state changes. 31
Figure 15 Application Topology Illustration 36
Figure 16 Test Result. 37
Figure 17 Network System Architecture 40
Figure 18 Open-loop velocity control with voltage torque mode Source: SimpleFOC docs 46
Figure 19 Diagram of the voltage torque controller. Source:SimpleFOC docs 47
Figure 20 FSM Diagram of the Node Firmware. 55
Figure 21 Heat Recovery Ventilation System. 61
Figure 22 Physical test bench. Supervisor running in MPU, two DRV8302 motor drive nodes with BLDC motor, shared 12 V power supply, and CAN bus interconnect. 73
Figure 23 Supervisor TUI during operation, accessed via SSH connection. The display shows coordinator and node states with diagnostic counters, velocity setpoint, real-time event log, and operator key inputs 76
Figure 24 Phase currents from both nodes during synced operation. UART data from the Teensy boards shows both motors spinning at the exact same operator-commanded speed, with classic sinusoidal waveforms from Space Vector PWM under control. 77

List of Tables

Table 1 Manual design vs. synthesis-based engineering 5
Table 2 Mode-select pin configuration and its consequences for the fault model. 43
Table 3 `nFAULT` assertion conditions, thresholds, and clear mechanisms [SLVSA73]. 44
Table 4 Fault classification by pin state after 50 ms confirmation window. × = don't-care. Gate behaviour is identical for all three classes [SLVSA73]. 45
Table 5 SimpleFOC control layers and available modes. ✓ marks the configuration used in this work. (CS) requires a current sensor; (PS) requires a position sensor. .. 46
Table 6 Voltage torque control sub-modes in SimpleFOC. Each level activates automatically when the corresponding parameters are supplied before `motor.init()`. 48
Table 7 SinePWM vs. SpaceVectorPWM for a two-level three-phase inverter. ✓ denotes the mode used in this work. 49
Table 8 CANCommander 29-bit extended identifier layout (bit 28 reserved, unused). All routing and command semantics are encoded in the arbitration ID. 51
Table 9 CANCommander packet types encoded in bits 19–16 of the 29-bit ID. Values 0x4–0xE are reserved for future use. 52
Table 10 Selection of built-in SimpleFOC registers accessible via CANCommander. `float` = IEEE 754 single-precision, `uint32` = unsigned 32-bit integer, both little-endian. Custom registers begin at address `0xE0`.

..... 53
Table 11 Node states and their physical meaning. 56
Table 12 CAN register map. Each writable register maps to one controllable event in the CIF plant model. 59
Table 13 Cumulative and Consecutive counters 64

1 Introduction

1.1 Industrial Control Systems

Pick almost any industrial system built in the last three decades and you will find the same split: a discrete layer handling sequencing, mode switching, and safety interlocks, sitting on top of a continuous layer that regulates physical variables in real-time. The two layers exist because they solve fundamentally different problems - one is what to do next, the other about how to do it precisely - yet they must work in lockstep. A few examples make the point concrete.

Motor Drive Systems [1] are the closest relative of the work in this thesis. A drive must decide when to start, when to brake, and when to declare a fault (discrete logic), while simultaneously regulating phase currents at tens of kilohertz (continuous control) with goal of desired current,position or velocity .Section 4.2 describes in detail our approach in this type of system.

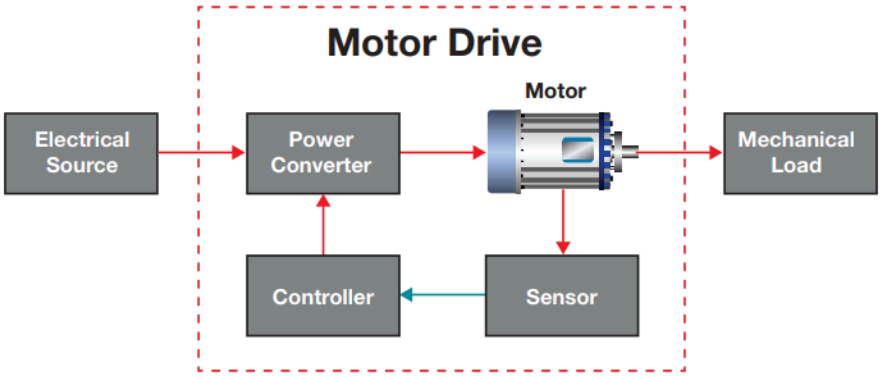


Figure 1: Block diagram of a Motor Drive System [1]

Manufacturing Automation Systems such as robotic arms, conveyors, tool changers face a similar split at a larger scale. A PLC sequences the production steps; servo loops inside each actuator keep trajectories on track. The coordination problem here is harder because subsystems interact physically: a robot cannot place a part if the conveyor has not indexed. Wonham and Cai [3] document several case studies where this coordination was handled with supervisory control theory.

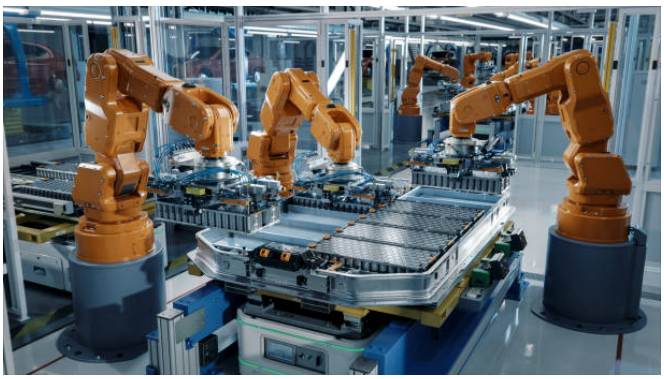


Figure 2: Automated manufacturing system (Image source: Pixabay)

Process Control Systems add a chemical dimension. batch recipes define sequences of heating, mixing, and holding steps (discrete), while PID loops regulate temperature, pressure and flow rate to keep each step within safe bounds (continuous). Awad [4] surveys SCT applications in this domain.



Figure 3: Industrial process control system. source: Pixabay

Semiconductor Manufacturing Equipment pushes both layers to their limits. ASML's lithography machines, for instance, coordinate hundreds of discrete wafer-handling steps while maintaining sub-nanometer positioning and vibrations isolation [5] .The ASML LOEW controller project, which used ESCET for supervisory synthesis, is one of the motivating industrial examples for this work.



Figure 4: ASML High NA EUV lithography machine. Source:ASML Media Library © ASML

Concluding Remarks:

Discrete decision-making logic and continuous dynamical control must work closely together in these systems as well as many others found in various industrial domains [3], [6]. This thesis is motivated by the fact that designing each layer separately is no longer the primary obstacle. There are well-established methods for continuous control and discrete behavior can also be successfully modeled. Instead of using ad hoc engineering the challenge is to integrate these layers into a coherent system in a scalable and methodical manner. In this regard the ESCET toolchain makes the development process more organized and manageable by offering a useful framework for characterizing simulating and coordinating the discrete-event behavior of complex systems.

1.1.1 The Integration Challenge

Development approaches address discrete and continuous domains separately:

Discrete Logic Design employs finite state machines, ladder diagrams, or manual programming to implement coordination sequences and safety interlocks. While capable of handling complex logical requirements, these methods provide no guarantees about physical realizability or formal correctness [7].

Continuous Control Design utilizes PID control, state feedback, and optimal control to achieve stability and performance. These methods excel at regulating physical processes but cannot adequately address discrete mode transitions or logical safety constraints [3].

Manual Integration of these separately-designed components creates systems that are difficult to verify, maintain, and certify. Engineers must manually ensure that discrete mode transitions do not destabilize continuous control loops, that safety interlocks are complete and non-conflicting, and that the system cannot deadlock [6].

Scale is the core issue. A modern automotive ECU can have 50 or more distinct operating modes. An industrial production line might coordinate thousands of components across dozens of subsystems. When an engineer hand-writes the glue code that ties discrete sequencing to continuous control loops across all of these modes, the number of possible state combinations grows combinatorially. Testing can catch some of the resulting bugs. It cannot catch all of them.

Industry 4.0 is making this worse, not better. Cyber-physical integration, smart automation, and networked manufacturing all push systems towards more modes, more interaction, and more failure scenarios that earlier generations had to cope with. Self-driving vehicles, robotic manufacturing cells, and smart grid controllers share a common trait: the discrete-continuous interaction surface is large and growing.

Functional safety standards recognize the problem. IEC 61508 [8], [9], [10] demands formal verification and traceability, requirements that ad hoc development/integration methods struggle to satisfy. When a safety assessor asks “can this system deadlock under fault condition X?”, the honest answer for a hand-coded system is usually “we tested for it and did not see it.”, that is not the same as a proof. The approach taken in this thesis treats the discrete-continuous boundary as a first-class design concern rather than an afterthought. Section 3 describes the ESCET toolkit and Section 4.5 shows our results on the real hardware.

There is a well-established way to think about this layering. Industrial control architectures stack several tiers, each running at a different time scale, as shown in Figure 5 (adapted from the Eclipse ESCET documentation). Our two-layer architecture maps directly onto this hierarchy: the resource controller tier and the supervisory controller tier.

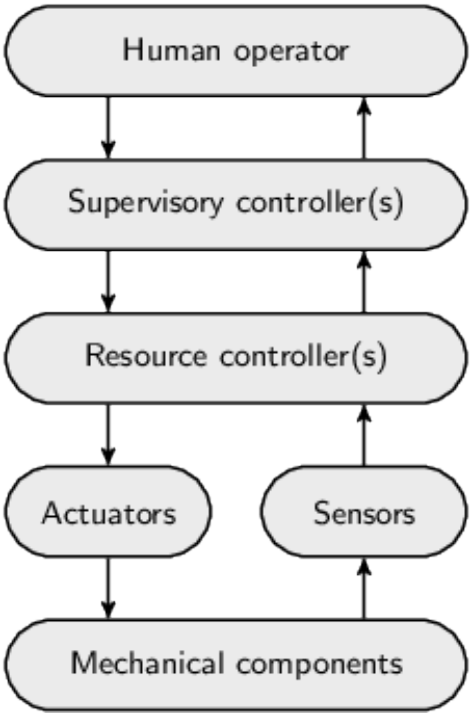


Figure 5: Traditional view of a control system (source: Eclipse ESCET documentation)

At the lowest level of the system architecture lie the **mechanical components**, including motors, switches, levers, and valves. Their operation is influenced through **actuators**, while their current state is observed via **sensors**. Together, sensors and actuators form the physical interface between the digital control layers and the real-world process.

Above the physical layer, **resource controllers** provide a first level of automation and regulation. Their responsibilities typically include compensating for sensor noise and jitter, real-time deterministic classical control, calibration, prediction and converting continuous signals into discrete events upwards, and detecting anomalous operating conditions. In certain cases, they may also apply corrective actions to maintain stable and safe operation.

Supervisory controllers implement system-wide coordination and safety logic at a level of abstraction above resource controllers hardware actuation. Rather than reacting to individual signals, they govern the sequencing, enabling, and constraining of lower-level resource controllers ensuring that the system as a whole operates within safe and correct boundaries, even when individual subsystems behave nominally in isolation. Core responsibilities include enforcing system-level invariants (such as mutual exclusion between conflicting operations), orchestrating state transitions across multiple subsystems, and responding to abnormal conditions by driving the system toward a safe and recoverable state. This is distinct from local protective reactions, which are handled at the device or controller level, and instead concerns the global consistency of system behavior: *what is permitted, in what order, and under what conditions .*

In hierarchical architectures, supervisory control may appear at multiple layers. A supervisor at one level may itself be a controlled plant from the perspective of a higher-level supervisor, with its own controllable and uncontrollable events. This compositional structure describes

and segments complex system behavior into manageable, verifiable layers, each with a well-defined scope of responsibilities and observations.

Many systems operate nowadays in a fully autonomous manner, but close to all of them incorporate a **human–machine interface (HMI)**. This enables operators to monitor system status, logs, and intervene manually when required, thus providing transparency, flexibility, and an additional layer of safety.

Supervisory controllers are essential to the reliable and safe functioning of cyber-physical systems, regardless of the particular architectural configuration and level of automation. For current automated environments to achieve robustness, fault tolerance, and operational safety, their existence in systems is essential.

1.2 Synthesis-Based Engineering

Supervisory Control Theory (SCT), introduced by Ramadge and Wonham [7], provides a formal framework for automatically synthesizing controllers that restrict system behavior to meet specified requirements. Rather than manually designing coordination logic, the engineer models the plant (what the system *can* do) and the requirements (what it *should* do); a synthesis algorithm then computes a supervisor that enforces the requirements while being able to guaranty safety, nonblocking, and maximal permissiveness.

Table 1: Manual design vs. synthesis-based engineering

Manual Design	**Synthesis-Based**
Engineer designs controller manually	Model plant + requirements; synthesis automatic
Controller behavior implicit in code	Plant and requirements explicit and separate
Testing finds bugs	Mathematical proof of correctness + simulation
Modify code to add requirements	Modify requirement model; re-synthesize
No formal guarantees	Safety, nonblocking, and maximal permissiveness can be guaranteed

This thesis uses the Eclipse Supervisory Control Engineering Toolkit (ESCET) [2], an open-source model-based framework that implements SCT using the CIF modeling language, symbolic synthesis via Binary Decision Diagrams, and code generation targeting C, Java, PLC, and Simulink. Synthesis-based engineering with ESCET has been applied in manufacturing automation (flexible manufacturing cells, assembly lines) [3], semiconductor lithography (ASML LOEW controller) [5], chemical batch process control [4], and transportation systems (railway interlocking, traffic coordination) [3]. The toolkit and its workflow are described in detail in Chapter 3.

1.2.1 Research Gap

SCT provides formal synthesis for discrete coordination, and classical control theory provides well-established methods for continuous regulation, but no systematic methodology exists for integrating these two domains in embedded systems. While commercial model-based solutions enable both discrete and continuous behavior but lack formal assurances, applications of SCT typically focus only on discrete coordination. Hierarchical control techniques aid in structuring systems across abstraction layers, although their practical application is

frequently less evident. This thesis addresses the gap directly, the specific contributions are outlined in the following section.

1.3 Research Objectives and Contributions

1.3.1 Primary Objective

This thesis develops a systematic two-layer embedded control architecture that couples formal supervisory synthesis (via Eclipse ESCET) with classical real-time control, maintaining provable correctness guarantees across the discrete–continuous boundary.

1.3.2 Contributions

1. Two-Layer Architecture with Formal Interface

We decompose embedded control into a **Supervisor Layer**, discrete coordination logic synthesized as Extended Finite Automata with guaranteed safety, nonblocking, and controllability, and a **Resource Control Layer** implementing real-time continuous algorithms (e.g. PID, state feedback) for individual actuators. The key contribution is the *interface specification* between layers: a formally defined event semantics that ensures supervisor enable/disable decisions translate correctly to physical actuation while preserving synthesis guarantees. This addresses a gap in hierarchical control literature, where layer integration is typically handled informally.

2. End-to-End ESCET Workflow for Embedded Deployment

We establish, to our knowledge, the first documented workflow taking ESCET synthesis-based engineering from requirements capture to deployed embedded code. The workflow covers component-based CIF modeling, automated synthesis, C99 code generation for resource-constrained platforms, and integration procedures that preserve both formal correctness and real-time timing.

3. Industrial Validation: Distributed BLDC Control Network

A distributed motor-control configuration with several CAN-connected BLDC drive nodes under the supervision of a centralized synthesized controller is used to assess the suggested architecture. Coordinated multi-node operation, fault scenarios like thermal overloads, electrical malfunctions, and power outages, and real-time continuous motor control are all part of the experimental system. The outcomes demonstrate that the synthesized supervisory behavior may be applied uniformly throughout the entire system on actual hardware. Chapter 4 contains a thorough explanation of the implementation and experimental setup.

4. Reusable Engineering Guidance

We distill practical modeling patterns, representing actuators, sensors, shared resources, and safety interlocks as EFAs, together with rules for mapping event-based supervisory control onto a CAN protocol. These patterns aim to reduce the barrier between supervisory control theory and embedded practice.

1.3.3 Positioning

Compared with plain *model-based design* (Simulink/Stateflow), our approach replaces manual testing-based verification with automatic synthesis backed by mathematical proof. Compared

with *prior SCT applications*, we provide systematic integration with real-time resource controllers and demonstrate it on distributed hardware rather than treating discrete coordination in isolation. Compared with *hierarchical control theory*, we provide implementation guidance validated on a physical platform, bridging the gap between multi-level abstractions and deployed systems.

1.4 Thesis Organization

Chapter 2 covers the formal foundations: Discrete Event Systems, Supervisory Control Theory, and Extended Finite Automata.

Chapter 3 introduces the ESCET/CIF toolset and the synthesis-based engineering workflow from modeling, simulation and deployment code generation.

Chapter 4 applies the approach to a distributed BLDC motor control network. CIF plant models capture individual node behavior, application-specific requirements are formalized for a balanced ventilation (HRV) use case, and the supervisor is synthesized and integrated with the physical system over CAN bus. And summarizes our results.

Chapter 5 discusses limitations, and outlines future work.

2 Theoretical Foundations

Before diving into the toolchain and the case study, we need to lay out the theory that makes synthesis-based engineering possible. This chapter covers three topics. First, Discrete Event Systems and finite automata the basic modeling language for event-driven logic. Second, Supervisory Control Theory [7] the mathematical machinery that turns plant and requirement models into provably correct supervisors. Third, Extended Finite Automata which provides a practical extension that adds variables and guards to classical automata, which is what ESCET actually uses under the hood. Readers already comfortable with SCT and EFAs may want to skim this chapter and move to Chapter 3.

2.1 Discrete Event System

A discrete event system can be conceptualized as a system that evolves through a sequence of states $q_0, q_1, q_2, ...$ where transitions between states are triggered by events $\sigma_1, \sigma_2, ...$ from an event alphabet Σ. The main characteristics of a DES are:

- **Discrete State Space:** The system state takes values from a finite or countably infinite set, rather than continuous real-valued spaces.
- **Event-Driven Dynamics:** State changes occur instantaneously upon event occurrences, with the system remaining in each state for finite (but possibly unknown) duration.

2.1.1 Finite Automata as Modeling Framework

The workhorse of DES modeling is the finite automaton [4]. The concept is straightforward: a system sits in one of finitely many states, and it jumps between states when discrete events occur. Nothing happens between events, the system just waits. This is a deliberate abstraction. We throw away all continuous-time dynamics and keep only the event-driven skeleton.

Why is this useful? Because it lets us reason about sequencing, safety, and liveness with mathematical precision. A state might represent "motor running" or "gate driver faulted"; a transition might represent "thermal shutdown detected." If we can enumerate all reachable states and all possible event sequences, we can prove that certain dangerous states are unreachable. That is the payoff. Cassandras and Lafortune [6] provide the standard reference; Ramadge and Wonham [7] showed how to exploit this structure for automatic controller synthesis.

Definition (Discrete Event System):

A Discrete Event System is modeled as a finite automaton

$$G = (Q, \Sigma, \delta, q_0, Q_m) \tag{1}$$

where:

- Q: finite set of states
- $\Sigma = \Sigma_c \cup \Sigma_{\text{uc}}$: finite alphabet of events (controllable and uncontrollable)
- $\delta : Q \times \Sigma \to Q$: partial transition function
- $q_0 \in Q$: initial state
- $Q_m \subseteq Q$: set of marked (accepting) states

The transition function $\delta(q, \sigma)$ is defined only when event σ is admissible in state q. If undefined, σ cannot occur from q, this models physical or logical constraints.

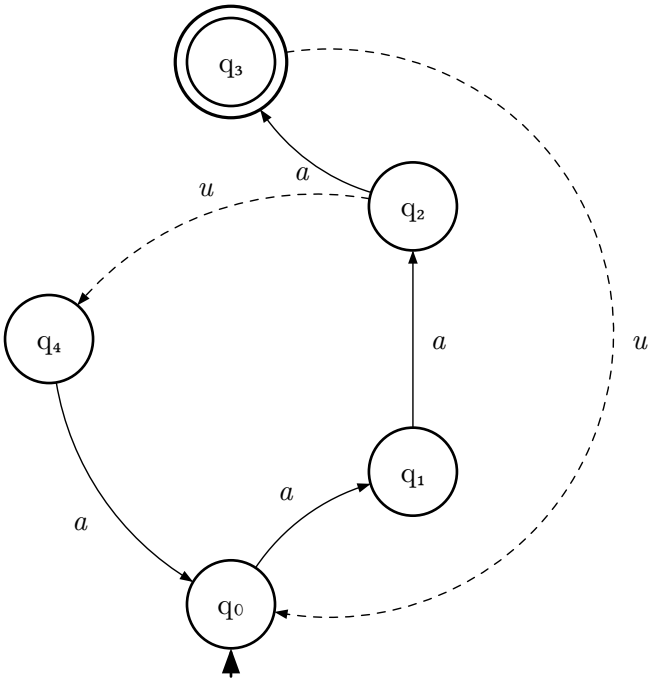


Figure 6: General DES state-machine topology.

The **closed language** $L(G)$ represent all possible event sequences the automaton can execute from its initial state. The **marked language** $L_{m(G)} \subseteq L(G)$ contains only those sequences that lead to marked states, representing completed or accepted behaviors.

In diagrams illustrating DES, the initial state is shown by an arrow coming from nowhere, marked states are indicated by a double circle and as for the events, solid and dashed transition arrows represent controllable and uncontrollable respectively. These conventions make it easy to distinguish the role of each state and event in the system.

Fundamental properties characterize the behavior of discrete event systems and are fundamental for supervisory synthesis.

Definition (Event): An event represents an instantaneous occurrence that causes a state transition. Events are classified as:

- **Controllable:** Can be disabled by the supervisor (e.g. actuator commands)
- **Uncontrollable:** Cannot be prevented (e.g. sensor readings, disturbances)

This classification is fundamental to supervisory control theory [7].

The distinction between controllable and uncontrollable events directly reflects the physical reality of embedded control systems. For example, in motor control system, the command to start the motor is controllable (can be prevented by the supervisor), while a fault signal from an over-current sensor is uncontrollable (cannot be prevented from occurring) which can only be observed.

2.2 Supervisory Control Theory

Supervisory Control Theory, introduced by Ramadge and Wonham [7], provides a systematic framework for the automated synthesis of discrete event controllers. The fundamental idea is to decompose the control problem into two parts: the **plant** (network of automata depicting what the system can do) , and the **requirements** (what the system should do).

2.2.1 The Supervisory Control Problem

The supervisory control problem is formulated as follows:

"given an uncontrollable plant and a specification of desired behavior, can we automatically construct a supervisor/controller that enforces the specification while maintaining essential system properties?"

Given:

- **Plant model** representing all physically possible behaviors and **Requirements** specifying desired behaviors and safety constraints.

Find:

- Supervisor S that restricts plant behavior to satisfy requirements while ensuring :
  1. **Controllability**: Only controllable events are restricted.
  2. **Non-Blocking**: System can always find a way to a marked state (can always complete tasks)
  3. **Maximal Permissiveness**: Plant behavior is restricted minimally.

The supervisor observes events generated by the plant and controls the plant by **enabling** or **disabling** controllable events based on the current system state.

2.2.2 Plant and Requirements Modeling

In practical supervisory control, both the plant and the requiremetns are modeled as networks of automata that interact through shared events.

Definition (Plant Model): The **plant** is a network (synchronous composition) of automata $G = G_1 \parallel G_2 \parallel ... \parallel G_N$ where:

- Each $G_i = \left(Q_i, \Sigma_i, \delta_i, q_{0,i}, Q_{m,i}\right)$ represents a physical component or a subsystem
- Automata synchronize on **shared events**: if $\sigma \in \Sigma_i \cap \Sigma_j$, then G_i and G_j must execute σ simultaneously.
- Events not shared by any automaton execute independenlty.
- The plant represents **all physically possible behaviors**, including both safe and unsafe states.

Definition (Requirement Model):

Requirements are specifications of desired behavior, also modeled as automata $R = R_1 \parallel R_2 \parallel ... \parallel R_m$ where:

- Each R_i constrains specific aspects of system behavior (e.g., safety constraints, operational sequences)
- Requirements automata typically have **no marked states** or **all states marked**, focusing on restricting transitions rather than accepting languages

- Requirements synchronize with plant automata through shared events
- The requirement model specifies **what behaviors should be prevented** or **what conditions must be maintained**

The distinction between plant and requirements is crucial:

- **Plant:** Describes physical capabilities and limitations (cannot be changed)
- **Requirements:** Describes design choices and safety policies (can be modified)

Example:

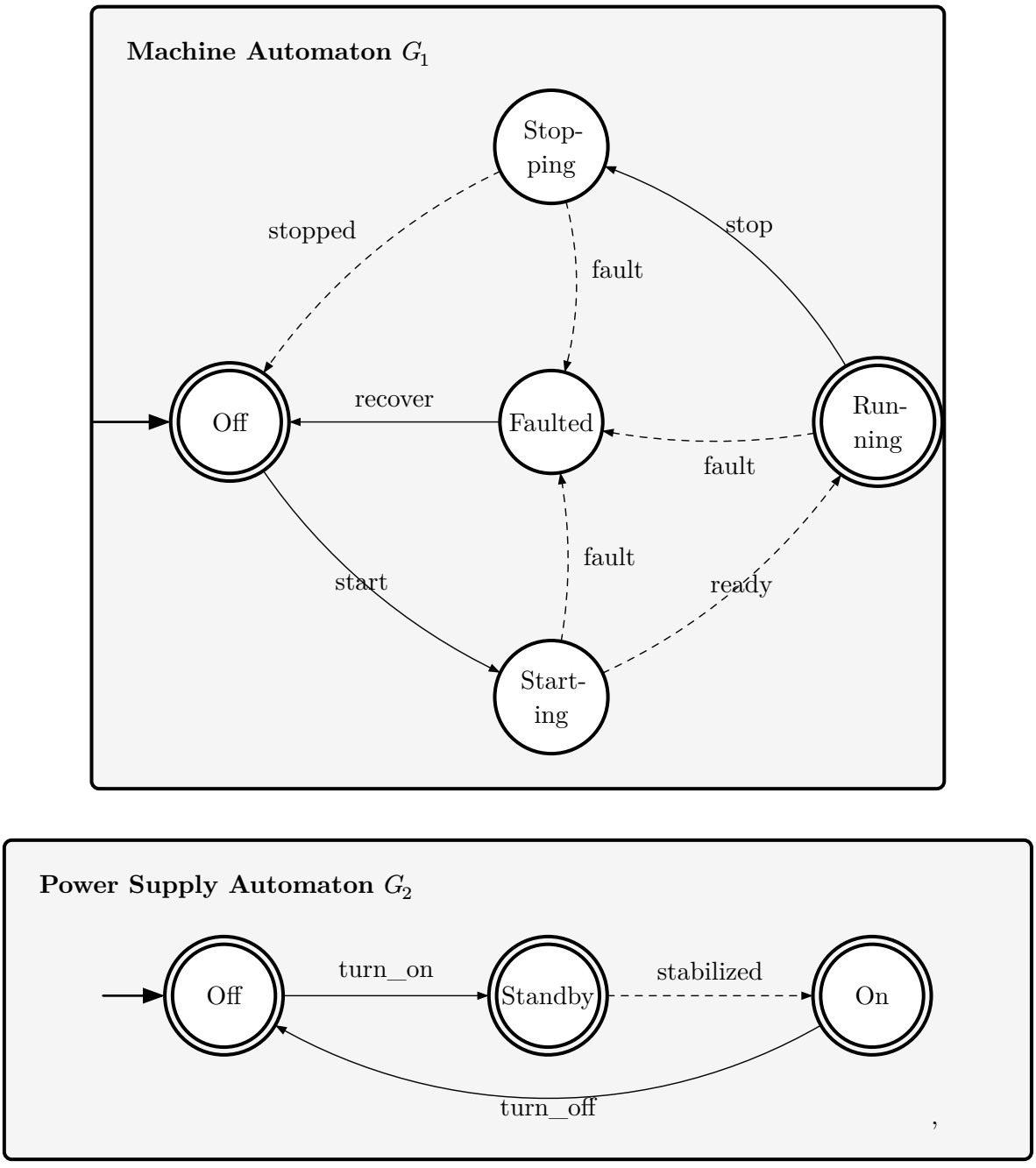


Figure 7: Example System Plant

The example plant Figure 7 is modeled by two automata, each describing a different physical part of the system. The **Machine** automaton G_1 captures the machines operational modes and the events that move it between them.The controllable events are `start`, `stop`, and `recover`, which correspond to operator or controller commands. The uncontrollable events `ready`, `stopped`, and `fault`, which represent signals coming from the machine itself. The normal operational sequence is : the machine is initially in `Off`, then after `start` command it goes to `Starting`, then moves to `Running` when the `ready` signal occurs. From `Running`, a `stop` command sends it to `Stopping`, and when the `stopped` signal occurs, it returns to `Off`. Faults can occur while the machine is `Starting`,`Running` or `Stopping`, when a `fault` event happens in any of these states, the machine moves to the `Faulted` state. To leave `Faulted`, the operator must issue the controllable event `recover`, which brings the machine back to `Off` after the fault has been acknowledged and dealt with. In this model, `recover` is intended to represent a deliberate human or supervisory action, not an automatic reset, because it assumes that the fault cause has been checked and the restart has been explicitly approved.

Only two locations are marked as acceptable system states. The `Off` state is marked because it represents a safe and completed situation in which the machine is not operating. The `Running` state is also marked, since the machine is performing its normal task is treated as a valid and intended condition, not something that needs correction. By contrast, the `Faulted` state is intentionally left unmarked, to indicate that a fault is not an acceptable final outcome; the system must be actively brought out of this state. In other words, from `Faulted`, the controller is expected to apply the controllable event recover so that the automaton can reach the marked `Off` state and thus remain non-blocking under supervision. The intermediate states `Starting` and `Stopping` are also unmarked, because they represent temporary transitions where the machine is neither fully off nor fully running, and the process has not yet reached a completed or steady condition.

Note:

A typical modeling option, as applied here, is to presume that one recovery action consistently succeeds, ensuring that the recovery event reliably restores the machine to a secure marked state. In more complex systems, this assumption might not be valid. A malfunction may demand a cooling phase, multiple reset tries, or possibly manual intervention before the equipment can function again. In these situations, it might be required to designate certain fault-related conditions as acceptable to prevent reporting incorrect blocking behavior, as the model needs to indicate that the system can validly exist in a degraded or fault state for a period, acting as a fail safe state [8]. This kind of design choice is examined more deeply in the case study found in Section Section 4.4.1.

The **Power Supply** automaton models the electrical power subsystem with three states, all of which are marked as they signify normal operation. The `Off` state indicates that the power supply is shut down, meaning no energy is provided. The `Standby` state signifies the power-up stage, during which elements like capacitors are charging and the voltage regulation is still stabilizing. This procedure requires a limited duration on the actual hardware. The `On` state signifies that the power supply is in a stable condition and providing reliable power.

Every one of these states is marked since each is regarded as an acceptable operational condition for the power supply: it may properly be off, in the process of stabilizing, or completely on. By marking Standby, the model highlights that the period of waiting for the power to stabilize is not a mistake and not an “incomplete” condition, but a regular aspect of

the subsystem's behavior during initialization. The uncontrolled event `stabilized` happens when the voltage regulation procedure is finished, transitioning the automaton from `Standby` to `On`. The controllable event `turn_on` initiates the change from `Off` to `Standby`, while the controllable event `turn_off` reverts the system from `On` to `Off`.

These two automata compose synchronously through shared events, with requirements constraining their interaction to enforce safety and operational sequencing across the combined system.

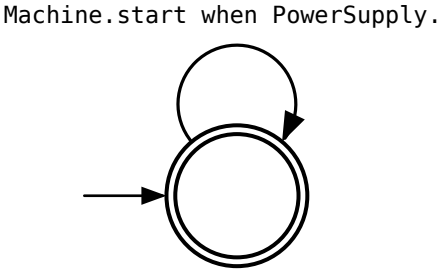


Figure 8: R_1: Machine start permitted only when power supply is On.

Requirement R_1 enforces a prerequisite condition for safe machine startup using a **state/event exclusion requirement**. This requirement directly guards the controllable `start` event with the condition `PowerSupply.On`, which checks whether the Power Supply plant automaton is currently in its `On` location. The requirement is represented as a single-location automaton with a guarded self-loop: the `start` event is only enabled when the guard condition holds. The `On` location is reached after the uncontrollable `stabilized` event occurs and persists until `turn_off` is executed. During synthesis, this requirement adds the guard `PowerSupply.On` to the `start` event, effectively disabling machine startup on unstable or disconnected power and preventing potential damage to motor drivers.

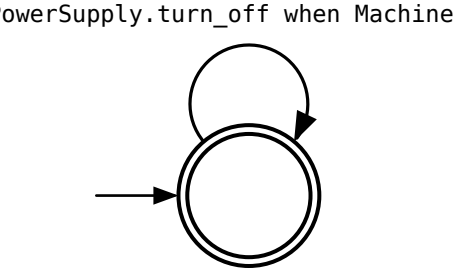


Figure 9: R_2: Power supply turn_off permitted only when machine is Off.

Requirement R_2 consolidates safe shutdown constraints by restricting power supply turn off to the machine's Off state using a **state/event exclusion requirement**. This requirement guards the controllable `turn_off` event with the condition `Machine.Off`, which directly checks whether the Machine plant automaton is in its `Off` location. This single requirement subsumes two previously separate event-tracking requirements: preventing power disconnection during active operation. The `Off` location represents the only safe state where the machine has completed all operational sequences and is not experiencing fault conditions. During synthesis, this requirement adds the guard `Machine == Machine.Off` to the `turn_off` event, ensuring power remains available throughout active operation and fault recovery procedures, thereby protecting motor drivers, safety interlocks, and diagnostic systems from premature intended power loss.

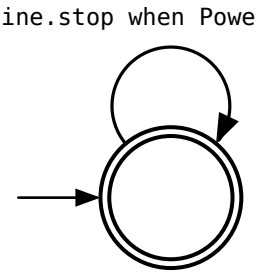


Figure 10: R_3: Controlled stop requires active power supply.

Requirement R_3 guarantees that a `stop` command/event can only be executed when the power supply is operational, ensuring that shutdown is consistently managed and power-supported. In contemporary industrial machinery, electrical energy is essential for safe deceleration: motor controllers need power for both controlled and regenerative braking, position encoders require energy to indicate the final position, and safety circuits must remain powered until the shutdown process is finalized. If a stop command were permitted during power removal or loss, the machine would just coast to a halt without adequate control or supervision. Together with R_2, this indicates that power cannot be eliminated while the machine is `Running` and that the `stop` command is only allowed when power is available, guaranteeing that each switch to the `Off` state employs a controlled, powered shutdown.

2.2.3 Key Properties for Synthesis

Three fundamental properties must be satisfied by a supervisor to ensure correct and practical control:

Definition (Controllability):

A specification is **controllable** with respect to the plant and uncontrollable events Σ_u if:

Whenever the system is in a state satisfying the specification, and an uncontrollable event can occur in the plant, the resulting state must also satisfy the specification.

Intuition: "You cannot prevent what you cannot control."

If a specification is not controllable, no supervisor can enforce it because uncontrollable events will inevitably violate the specification. In such cases, synthesis computes the **supremal controllable sublanguage**, the largest controllable approximation of the desired specification.

Definition (Nonblocking):

A system is **nonblocking** if from every reachable state, there exists a sequence of events (both controllable and uncontrollable) that leads to a marked state.

Intuition: The system can always eventually complete its task and reach an accepting state, provided expected events occur.

Nonblocking is defined with respect to the modeled event set. Generally, systems operate under **progress assumptions**, in our context, meaning that certain uncontrollable events are expected to occur eventually (sensor readings arrive, physical processes complete, communication succeeds). A system may be non-blocking under these assumptions yet appear blocking if modeled without them.

Real embedded systems implement timeout mechanisms and fault detection to handle cases where expected uncontrollable events fail to occur within acceptable bounds. The choice of which locations to mark reflects design intent: unmarked intermediate states represent ongoing processes that must complete before the system considers its task finished.

Definition (Maximal Permissiveness):

A supervisor is **maximally permissive** if it permits all safe behavior while ensuring controllability and non-blocking characteristics.

Intuition: The supervisor limits behavior only to what is essential for safety and accuracy.

Maximal permissiveness ensures that the system retains as much operational flexibility as possible while still satisfying all constraints.

Note: The synthesis procedure automatically achieves these three properties: controllability, nonblocking behavior, and maximal permissiveness. This eliminates entire categories of design errors that frequently arise in manually built controllers, including:

- Prevention of uncontrollable events
- Creating deadlock situations
- Dangerous states
- Unnecessary restriction of system behavior

2.3 Extended Finite Automata

Although classical finite automata offer a sophisticated mathematical basis for supervisory control, they suffer a critical drawback: the **state explosion problem**. Extended Finite Automata (EFA) were developed to address this constraint while maintaining the formal groundwork required for synthesis [11].

2.3.1 Motivation: State Explosion Problem

Classical finite automata require explicit enumeration of all system states. For systems involving data or numerical parameters, this leads to an exponential explosion of states.

Example - Buffer with Capacity:

A simple buffer of capacity N storing items requires $N + 1$ states in a classical automaton:

- State 0: Empty
- State 1: Contains 1 item
- ...
- State N: Full

For $N = 100$, this requires 101 states just for one buffer. A system with 10 such buffers would require $(N + 1)^{10} \approx 10^{20}$ states - clearly impractical.

More generally, for n components each with k states, the synchronous composition yields k^n global states. This exponential growth quickly becomes computationally intractable.

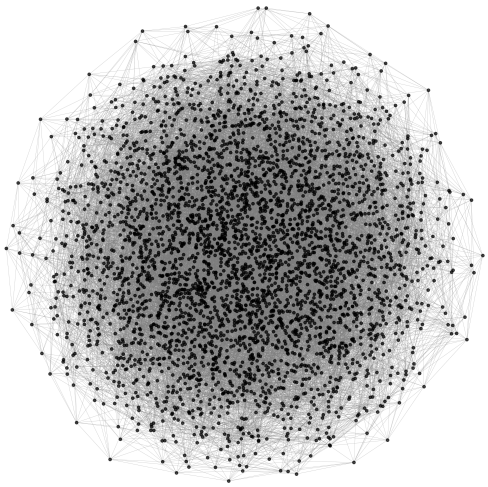


Figure 11: This graph shows a state space illustration of a plant composed of 6 automata with 4 states each. $(k^n = 4^6 = 4096)$

EFAs address this by incorporating **discrete variables** directly into the automaton model, avoiding the need to create explicit states for every possible variable value.

2.3.2 EFA Definition

Definition (Extended Finite Automaton) [11]:

An EFA is formally defined as a 7-tuple:

$$G = (Q, \Sigma, V, T, q_0, Q_m, v_0) \tag{2}$$

where:

- Q: finite set of **locations** (also called control states), representing distinct modes or configurations of the system (e.g., "" or "active")
- Σ: finite set of **events** (alphabet), partitioned into controllable (Σ_c) and uncontrollable (Σ_u) events
- V: finite set of **discrete variables** with finite domains (e.g., integers within a bounded range), used to store data like counters or flags
- $T \subseteq Q \times \Sigma \times \mathcal{G} \times \mathcal{A} \times Q$: **transition relation**, where each transition is a 5-tuple $(q_s, \sigma, g, \alpha, q_t)$:
  - $q_s, q_t \in Q$: source and target locations
  - $\sigma \in \Sigma$: triggering event
  - $g : \text{Val}(V) \to \{\text{true}, \text{false}\}$: **guard predicate**, a boolean condition over variables that must hold for the transition to be enabled
  - $\alpha : \text{Val}(V) \to \text{Val}(V)$: **update function**, specifying how variable values are modified upon taking the transition

  Note: Where Val(V) denotes the set of all valuations of V
- $q_0 \in Q$: **initial location**, the starting configuration of the automaton
- $Q_m \subseteq Q$: set of **marked locations**, representing acceptable completion states.
- $v_0 : V \to D$: **initial variable valuation**, assigning starting values to all variables (where D denotes the union of all variable domains)

The main advantage of extended finite automata (EFAs) is that they combine two aspects in a single model: the **control structure** (locations and events) and the **data** (variables, guards, and updates). Because of this, an EFA can describe behavior that, in a standard finite automaton, would require a very large number of separate states.

2.3.3 EFA Semantics

The operational semantics of EFAs integrate discrete event-driven dynamics with data manipulation.

Definition (Configuration): A **configuration** of an EFA is a pair (q, v) where:

- $q \in Q$ is the current location
- $v : V \to D$ is the current **valuation** (assignment of values to all variables)

The state space of an EFA is the set of all possible configurations. While this set may be very large (or even infinite if variable domains are unbounded), the EFA representation remains compact.

Definition (Transition): A transition $t = (q_s, \sigma, g, \alpha, q_t)$ is **enabled** in configuration (q_s, v) if and only if:
1. The current location is q_s
2. The guard evaluates to true: $g(v) = \text{true}$
3. Any location or automaton invariants are satisfied

When event σ occurs and an transition t is **enabled**, the system instantaneously moves to configuration $(q_t, \alpha(v))$, where:
- Location changes from q_s to q_t
- Variables are updated according to α

This semantics ensures that EFA behavior is both:
- **Event-driven:** Triggered by discrete events
- **Data-dependent:** Controlled by current variable values and modified by actions

2.3.4 Data-Based Synthesis for EFAs

For EFAs, synthesis cannot rely on explicit state enumeration due to the potentially enormous configuration space. Instead, **data-based synthesis** operates directly on the symbolic EFA representation using guards and updates

The key idea is **guard strengthening**: the supervisor enforces specifications by adding conjunctive conditions to the guards of controllable transitions, without modifying the plant structure.

Data-Based Synthesis Algorithm [2], [12]:

The algorithm computes strengthened guards for controllable events through three iterative steps:

1. **Compute nonblocking conditions:** For each location, determine the conditions on variables under which a marked location is reachable.
2. **Compute bad state conditions:** Identify conditions under which an uncontrollable event can lead to a blocking state (state from which no marked location is reachable).
3. **Strengthen guards:** Add conditions to controllable transitions to prevent reaching bad states via uncontrollable events.

These steps are iterated until a fixpoint is reached, yielding a maximally permissive, controllable, nonblocking supervisor.

Note: Modern supervisory control synthesis tools, such as ESCET's CIF framework [2], implement data-based synthesis using symbolic techniques like Binary Decision Diagrams (BDDs). The practical application of CIF for modeling and synthesis is demonstrated in Chapter 4.

Concluding Remarks:

The combination of locations and variables in EFAs, together with data-based synthesis, provides the modeling power and computational feasibility needed for practical embedded control systems. This makes EFAs the foundation for the supervisor layer in the two-layer architecture presented in this thesis.

2.4 Mathematical Verification and Guarantees

> … Program testing can be used to show the presence of bugs, but never to show their absence!
>
> — E.W.Dijkstra

In supervisory control theory, mathematical verification offers strict formal assurances that synthesized supervisors will adhere to safety regulations while attaining the best permissiveness possible within the EFA framework [7]. Supervisory control theory provides mathematical certainty on system behavior, in contrast to conventional control methods that depend on testing and simulation which can be used also in SBE. This method's strength is its constructive character; in addition to determining whether a control solution exists, the theory offers specific algorithms to calculate the ideal supervisor when practical [3]. Whole classes of runtime mistakes and specification violations that afflict traditional control systems are eliminated as a result. As mentioned the synthesized supervisor provides four essential mathematical guarantees:

Safety Assurance: The supervisor mathematically guarantees that $L(S/G) \subseteq K$, ensuring the controlled system never violates specifications. This represents an absolute guarantee for all possible paths, not probabilistic assurance.

Preservation of controllability: Uncontrollable occurrences may still take place in the monitored system, and they must consistently adhere to the physical constraints of the plant. The supervisor cannot prevent events that fall outside its control authority.

Optimality: Upon a successful synthesis step, the supervisor applies the supremal controllable sublanguage $K^{\uparrow}$. This implies it retains the most extensive collection of safe actions while adhering to the safety and controllability requirements

Nonblocking Property: The supervised system avoids deadlock situations by ensuring that every reachable state can progress to completion without getting trapped in non-marked states.

2.4.1 Validation Paradigm Shift

To confirm behavior, traditional control systems need to be thoroughly tested in a variety of scenarios. A significant change from "testing to detect problems" to "proving lack of problems" is represented by mathematical validation, which provides **a priori** assurances about every possible system behavior [6].

Industrial Impact:

Mathematical verification enables deployment of autonomous systems with unprecedented confidence levels. Rather than relying on extensive testing to catch edge cases, engineers can provide mathematical proofs that critical safety properties will never be violated under any circumstances. This is particularly crucial for safety-critical systems in manufacturing, transportation, and infrastructure control [13].

3 Synthesis-Based Engineering with ESCET

This chapter presents the Eclipse Supervisory Control Engineering Toolkit (ESCET) and its application to embedded control system development. We describe the synthesis-based engineering workflow, introduce the CIF modeling language, explain supervisor synthesis and verification, and detail code generation for embedded deployment.

3.1 The Synthesis-Based Engineering Workflow

Synthesis-based engineering follows a systematic model based four-phase approach [2]:

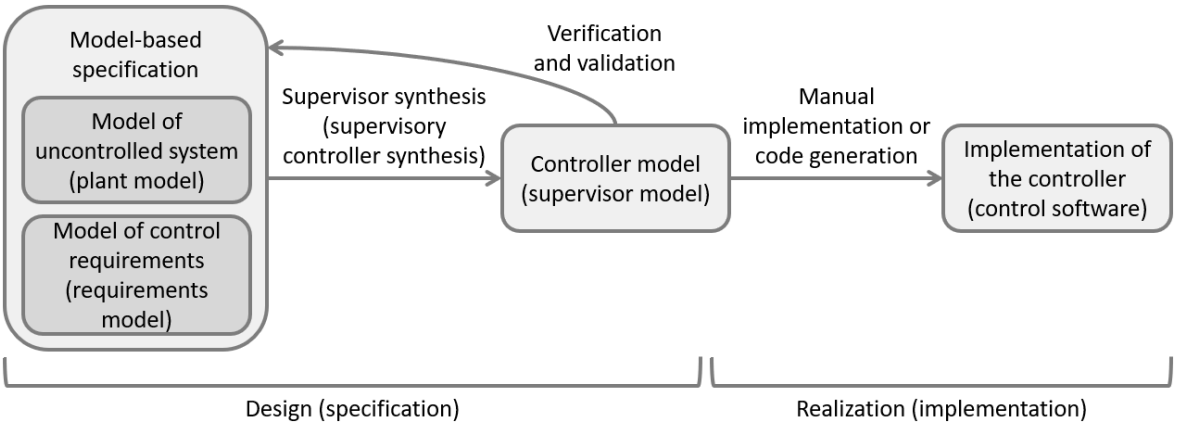


Figure 12: Synthesis-Based Workflow, source: Escet doc [2]

`Phase 1: Modeling` Engineers create CIF specifications describing:

- **`Plant models:`** Physical system capabilities (what CAN happen)
- **`Requirement models:`** Desired behavior constraints (what SHOULD happen)

Plant models represent uncontrolled system behavior including both safe and unsafe states. Requirement models specify safety constraints, operational sequences, and resource limitations.

`Phase 2: Synthesis` ESCET automatically computes a supervisor that:

- Restricts plant behavior to satisfy all requirements
- Guarantees controllability (respects uncontrollable events)
- Ensures nonblocking (no deadlocks)
- Provides maximal permissiveness (minimal restriction)

Synthesis operates on the symbolic EFA representation using guard strengthening—no manual controller design required.

`Phase 3: Simulation and verification`

In this phase, the controlled system (supervisor together with the plant) is tested by running interactive simulations. The tool can:

- Show the current system state and which events are enabled
- Let the user step through different scenarios
- Check whether the requirements are satisfied
- Analyze controller properties, such as non-blocking behavior and controllability

If the simulation reveals problems or unexpected behavior, the models are adjusted and the supervisor is synthesized again.

Phase 4: Code Generation ESCET generates deployable implementations:

- C99 code for embedded systems
- PLC code
- Simulink blocks for integration

3.2 CIF Language Fundamentals

A CIF automaton combines several elements: unique locations (acting as states), events (instant incidents), discrete variables (finite data values), and edges (transitions with criteria and alterations). An instance of a sensor-actuator from the ESCET synthesis documentation [2] that demonstrates the main characteristics of CIF, including both controllable and uncontrollable events, is presented here:

```
input bool SENSE;  // Hardware abstraction: sensor reading from  layer
automaton Sensor:
  uncontrollable u_on, u_off;              // Sensor readings(uncontrollable)

  location Off:
    initial;                               // Sensor starts in Off state
    edge u_on when SENSE goto On;

  location On:
    edge u_off when not SENSE goto Off;
end

automaton Actuator:
  controllable c_on, c_off;                // Actuator commands (controllable)

  location Off:
    initial;                               // Actuator starts in Off state
    edge c_on when Sensor.On goto On;      // Turn on when sensor reads On

  location On:
    edge c_off when Sensor.Off goto Off;   // Turn off when sensor reads Off
end
```

This example demonstrates how the Sensor automaton models physical measurements that cannot be controlled but only observed, while the Actuator automaton models a controllable device (motor, valve, relay) that can be commanded on or off. The guard `when Sensor.On` on the Actuator's `c_on` edge illustrates event scoping and cross-automaton dependencies, the actuator command is only enabled when the sensor is in its `On` location.

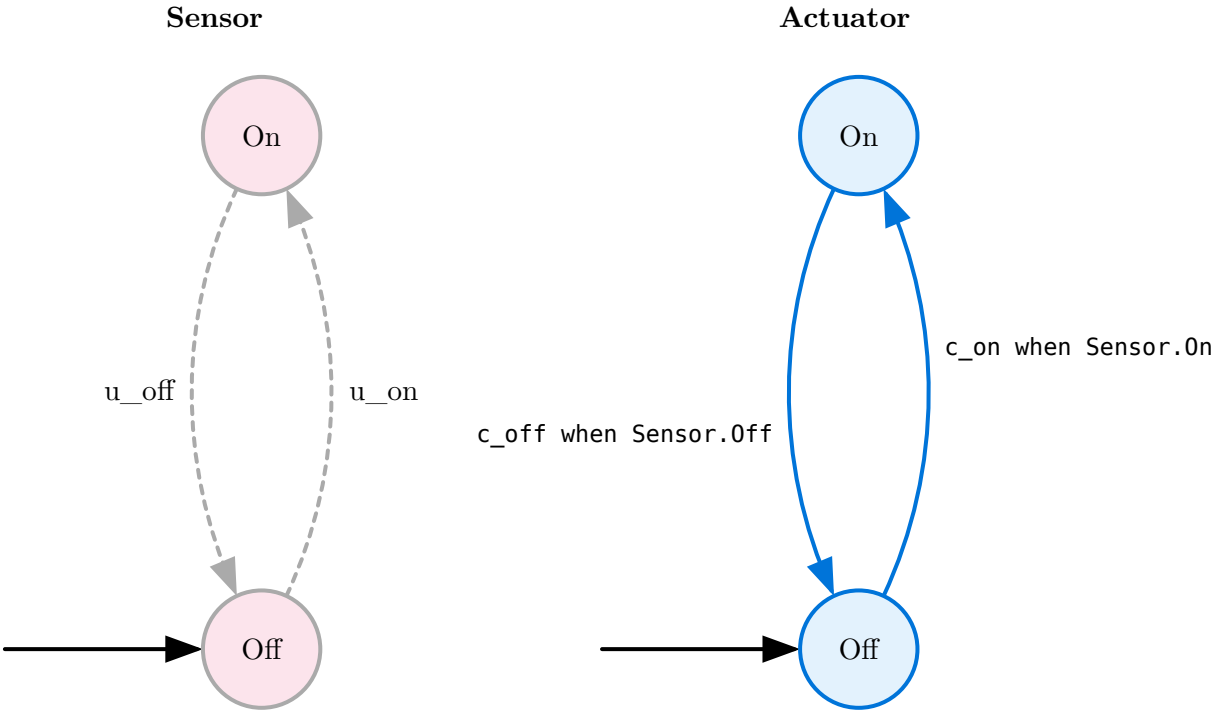


Figure 13: Sensor-Actuator system. Sensor (left) has uncontrollable events representing physical signal transitions. Actuator (right) has controllable events guarded by Sensor state. Dashed arrows: uncontrollable; solid arrows: controllable.

Input Variables and Resource Abstraction:

The `input bool SENSE` variable represents the interface between the discrete event supervisor model and the potentially continuous Resource Control layer. Input variables are read-only values provided by the resource environment at each supervisor evaluation cycle. In this example, `SENSE` abstracts the physical sensor reading—it could represent an ADC conversion, a CAN bus message field, an Ethernet packet payload, or a GPIO pin state etc. The uncontrollable events `u_on` and `u_off` model when the physical signal transitions, with guards `when SENSE` and `when not SENSE` determining event enablement based on the current hardware state. The diagrams show logical event transitions without these implementation details; in the actual implementation , the supervisor reads `SENSE`, evaluates guards, and determines which uncontrollable events have occurred since the last cycle. This separation of concerns, modeling logical event sequencing in automata while handling continuous signal sampling in the resource layer, is central to the supervisor architecture.

Actuator commands are limited by sensor readings to guarantee safe operation, this coordination pattern is essential for supervisory control. The supervisor monitors changes in sensor states through input variables, updates its internal automaton states via uncontrollable events, and then decides which controllable events to activate and in what order.

Core CIF Syntax Elements:

Automaton Declaration: Keyword `automaton` followed by name and colon introduces an automaton block, terminated by `end`. Each automaton represents a discrete system component with its own control states and event-driven behavior.

Event Declarations: Events can be `controllable` (the supervisor is able to prevent them) or `uncontrollable` (the supervisor cannot halt them). This indicates what the system is genuinely capable of controlling physically.

Controllable events typically refer to actuator commands such as `c_on`, `c_off` — actions the system determines whether to execute. Uncontrollable occurrences include sensor readings, outside interferences, or physical items such as `u_on`, `u_off` — the system can observe these events but cannot stop them.

This differentiation is vital for synthesis. The supervisor determines which manageable events to limit to satisfy the requirements, ensuring it functions regardless of the sequence of uncontrollable events

Input Variables: Declared with `input <type> <name>;` to represent values provided by the resource environment. Input variables are read-only within the supervisor model and connect the discrete event abstraction to continuous hardware signals. Guards on uncontrollable events typically depend on input variables to determine when physical state changes have occurred.

Locations: Discrete control states within an automaton. The `initial` keyword designates the starting location upon system initialization. The `marked` keyword (not shown in this example) indicates acceptable completion states, locations where task completion is recognized. Synthesis ensures nonblocking: from any reachable state, there must exist a path to a marked location.

Edges (Transitions): Define state changes triggered by events. Full syntax: `edge <event> [when <guard>] [do <updates>] goto <target>;` where each component is optional except the event. The keyword `when` creates a guard based on some logical condition. And `goto` specifies the destination state. An edge without `goto`, remains in the current location, this edge is called a self-loop.

Event Scoping: Events can reference locations from other automata using dot notation. In the Actuator example, `Sensor.On` refers to the `On` location of the Sensor automaton. This enables guards to condition behavior on the state of other components: `edge c_on when Sensor.On goto On;` means the `c_on` event can only occur when the Sensor is currently in its `On` location. Without explicit scoping, event names are local to their declaring automaton.

Guards: Boolean expressions restricting edge enablement. Syntax: `when <condition>` where condition uses comparison operators (`<`, `<=`, `>`, `>=`, `==`, `!=`), logical operators (`and`, `or`, `not`), and arithmetic expressions. An edge is enabled only when its guard evaluates to true. Guards on controllable events become synthesis targets, synthesis strengthens these guards to enforce requirements.

Discrete Variables: They are defined with `disc <type> <name> = <initial_value>;`. The type should possess a limited domain, which means only these are applicable:

- `int[min..max]` (restricted integers)

- `bool` (true/false)
- `{literal1, literal2, ...}` (lists)

Unlimited types are prohibited as they would generate infinite state spaces, which synthesis cannot manage. Variables allow you to represent many states in a compact manner, a counter that goes from 0 to 100 requires just **one variable** rather than 101 individual locations

Updates: Variable assignments executed atomically when edges are taken. Syntax: `do <assignment>, <assignment>, ...` using `:=` operator. Multiple assignments are comma-separated and execute simultaneously based on old variable values. Example: `do x := y, y := x` swaps values atomically. Updates appear only in edge declarations, never in guards (guards are pure boolean expressions).

3.3 Plant and Requirement Modeling

CIF distinguishes between plant automata (modeling overall system behavior) and requirement automata (specifying constraints) through semantic keywords that guide synthesis. This section demonstrates these concepts using the Machine and PowerSupply example introduced in the theoretical foundations (Chapter 2, Figure 7).

3.3.1 Plant Automata

The `plant` keyword declares automata representing the physical system's capabilities. Plants describe what the system **can** physically do, including both desired behaviors (normal operation) and undesired behaviors (faults, failures). The plant model is permissive, since it represents all physically possible event sequences without enforcement of safety or operational constraints.

The Machine and PowerSupply automata from Figure 7 are implemented in CIF as follows:

```
//Machine.cif
plant  Machine:
    controllable start, stop, recover;
    uncontrollable ready, stopped, fault;
    location Off:
        initial; marked;
        edge start goto Starting;
    location Starting:
        edge ready goto Running;
        edge fault  goto Faulted;
    location Running:
        marked;  // Mark Running (task considered complete)
        edge stop goto Stopping;
        edge fault  goto Faulted;

    location Stopping:
        edge stopped  goto Off;
        edge fault  goto Faulted;

    location Faulted:  //could be overheating, overcurrent etc
        edge recover goto Off;  //  Controllable escape from Faulted
end
```

```
//PowerSupply.cif
plant  PowerSupply:
    controllable turn_on, turn_off;
    uncontrollable stabilized;
    location Off:
        initial; marked;
        edge turn_on goto Standby;
    location Standby:
        marked;
        edge stabilized goto On; // waiting for stabilization
    location On:
        marked;edge turn_off goto Off;
end
```

The Machine plant models a physical device capable of starting, running, stopping, and encountering faults. The `recover` controllable event signifies deliberate fault clearing.

The PowerSupply plant models electrical power delivery through three phases:

- Powered-down (`Off`)
- Voltage stabilization (`Standby`)
- Delivery (`On`)

All locations are marked, reflecting that each phase represents a legitimate operational state. The uncontrollable `stabilized` event signals completion of the physical voltage regulation process.

Multiple plant automata compose through synchronous execution. When events are declared at the specification level (outside automata) and referenced within multiple automata, those automata synchronize on shared event execution. In this example, events are declared locally within each plant, creating independent event spaces that requirements can monitor and constrain.

3.3.2 Requirement Specifications

The `requirement` keyword declares specifications for desired system properties. Unlike plants which model physical capabilities, requirements impose logical constraints that restrict permissible behavior. During synthesis, requirements guide the computation of supervisor guards that disable controllable events when necessary to enforce constraints.

The three requirements from Figure 7 are implemented using **state/event exclusion** syntax:

```
//Requirements.cif
import "Plant.cif";
// R1: Machine start only when PowerSupply is On
requirement R1: Machine.start needs PowerSupply.On;
// R2:  PowerSupply turn_off only when Machine is Off
requirement R2: PowerSupply.turn_off needs Machine.Off;
// R3: Machine stop only when PowerSupply is On
requirement R3: Machine.stop needs PowerSupply.On;
```

State/Event Exclusion Requirements:

The `needs` keyword creates a **state/event exclusion requirement**, which restricts a controllable event to occur only when specified plant locations are active. The general form is:

```
requirement <automaton>.<event> needs <automaton>.<location>;
```

This construct directly guards the event with a location predicate, avoiding the need to create separate requirement automata that track plant state changes. The synthesis tool interprets `Machine.start needs PowerSupply.On` as: "strengthen the guard on `Machine.start` to include the condition `PowerSupply.On`", effectively adding `when PowerSupply.On` to any edge involving the start event.

Event Scoping with Dot Notation:

Requirements reference events and locations from plant automata using qualified names: `<automaton>.<event>` or `<automaton>.<location>`. This scoping mechanism allows requirements to observe and constrain plant behavior without modifying plant automaton definitions. In requirement R1, `Machine.start` refers to the controllable start event declared within the Machine plant, while `PowerSupply.On` refers to the location within the PowerSupply plant automaton.

Direct Location Predicates vs. Event-Tracking Automata:

State/event exclusion requirements can be conceptually understood as single-location requirement automata with guarded self-loops. For instance, requirement R1 is semantically equivalent to:

```
requirement automaton R1:
  location: initial; marked;
  edge Machine.start when PowerSupply.On;
end
```

However, the state/event exclusion form is preferred because it:
1. Eliminates redundant state tracking—the plant already models the `On` location
2. Reduces synthesis complexity by not using additional automaton state
3. By directly expressing the guard condition the readability is improved
4. Follows CIF best practices for "pure restriction" requirements [2]

Requirements **R1** and **R3** make sure machine operations (start, stop) only happen when power is stable (`PowerSupply.On`).

This keeps motor controllers, position encoders, and safety systems powered during commanded transitions. The `PowerSupply.On` state is reached after the uncontrollable `stabilized` event finishes voltage regulation, and it stays there until the controllable `turn_off` event happens.

Requirement **R2** handles **safe shutdown sequencing**: power can't be cut (`turn_off`) unless the machine has finished everything and is safely back in `Machine.Off`.

The `Machine.Off` state is the system's safe resting point — not starting, not running, not stopping, and definitely not faulted.

This prevents power loss while the machine is still moving through `Starting`, `Running`, `Stopping`, or recovering from `Faulted`. That way mechanical motion always completes with

controlled deceleration, and diagnostic systems stay powered during the whole fault handling process.

Requirement Composition and Non-Interference:

The three requirements compose without conflict because their guard conditions are independently satisfiable:

- R1 and R3 restrict machine operations to require power (`PowerSupply.On`)
- R2 restricts power removal to require machine Off (`Machine.Off`)

The requirements collectively enforce a safe operational protocol while maintaining non-blocking behavior—the system can always return to a marked state (`Machine.Off` with `PowerSupply.Off` or `PowerSupply.On`) from any reachable configuration.

Note: CIF supports both state/event exclusion requirements (as shown here) and explicit requirement automata that track event sequences through multiple locations. Requirement automata are necessary when constraints depend on event history or temporal patterns that cannot be expressed as static location predicates. The state/event exclusion form should be preferred when plant automata already model the relevant states, as it produces smaller synthesis problems and clearer specifications.

3.3.3 Input Variables and Resource Interface

As introduced in Figure 13, input variables model external observations — both from the physical environment and from upper-layer interfaces such as a Human-Machine Interface. For the plant model developed in this chapter, input variables gate events to their corresponding physical or operator signals:

```
input bool fault_flag;
input bool ready_flag;
input bool stopped_flag;
input bool stabilized_flag;
input bool stop_request;
```

For controllable events, input variables add an enabling condition while the supervisor retains authority to allow or block the event. An operator stop request, for instance, does not force the machine to stop — it permits the supervisor to issue a stop command if doing so is safe. Rather than adding when guards to individual edges, the plant invariant with needs keyword applies the guard globally to every edge carrying that event:

```
plant invariant Machine.fault          needs fault_flag;
plant invariant Machine.ready          needs ready_flag;
plant invariant Machine.stopped        needs stopped_flag;
plant invariant PowerSupply.stabilized needs stabilized_flag;
plant invariant Machine.stop           needs stop_request;
```

This is equivalent to adding the corresponding when guard to every edge for that event, expressed as a single declarative statement that automatically covers any new automata synchronizing on the same event. The data-based synthesis algorithm supports input variables and plant invariant declarations that reference them [2].

3.3.4 Synthesis-Relevant Modeling Patterns

The Machine and PowerSupply instance demonstrates various essential patterns for synthesis-oriented modeling:

How marking operates for nonblocking: The PowerSupply designates all three states (`Off`, `Standby`, `On`) as each represents a standard operating condition — powered down, voltage stabilizing, or supplying power. The Machine designates `Off` and `Running` as terminal states. `Starting` and `Stopping` remain unmarked because they are merely transient — the machine must move through them. Crucially, `Faulted` is **not** indicated. Errors aren't acceptable conclusions; the system requires restoration. The manageable `recover` event establishes a direct route from `Faulted` to designated `Off`, maintaining a non-blocking approach.

Categories of events relate to true control: Events that can be managed, like `start`, `stop`, `recover`, `turn_on`, `turn_off`, are actions that the supervisor can authentically direct (or disallow). Uncontrollable events like `ready`, `stopped`, `fault`, `stabilized` are concrete occurrences that naturally happen — sensors turn on, faults occur. Synthesis proves effective as it confines manageable situations while anticipating every possible uncontrollable circumstance. Incorrectly categorizing this (labeling sensors as "controllable" or commands as "uncontrollable") will cause your supervisor to be in error.

Plants perform physics, prerequisites ensure safety: The plant models merely outline what the machine **is capable** of doing physically. The Machine plant allows you to begin from any state (including `Faulted`), reduce power while operating, or experience faults during transitions. Requirements impose the constraints: R1 prevents startup without power, R2 prevents power loss during operation, R3 ensures fault recovery protection. This division — flexible plant + limiting criteria — renders specifications modular and easy to understand.

Guard enhancement in progress: While synthesizing, the tool incorporates guards into manageable transitions to uphold specifications. Use the Machine's initial `edge start goto Starting;` — without a guard, making it feasible to execute at any moment from `Off`. R1 compels synthesis to modify this to something akin to `edge start when PowerSupply.On goto Starting;`. The supervisor incorporates enhanced safeguards, restricting the system to safe operations while remaining non-invasive and as permissive as possible

Note: The difference between plant and requirement automata is semantic rather than syntactic —both utilize the same CIF syntax for states, transitions, and conditions. The distinction is found in the interpretation of synthesis and the modeling of intent. Plants outline physical reality (what is possible); requirements dictate expected behavior (what ought to occur). Synthesis generates supervisors that limit plant controllable events to meet requirements.

3.4 Supervisor Synthesis

Synthesis automatically computes a supervisor from plant and requirement models.

Synthesis Command (.toodef [2]):

```
//synthesize.tooldef
from "lib:cif" import *;
cifdatasynth("Models/Requirements.cif -o Synth/output_SUP.cif");
```

Note: The Requirements.cif file imports also the Plants.cif meaning we are providing both plant and requirement specifications.

3.5 Simulation and Verification

ESCET provides interactive simulation and automated property checking for validation before code generation.

3.5.1 Interactive Simulation

The CIF simulator enables step-by-step execution of the controlled system (supervisor + plant).

Simulator Launch:

```
from "lib:cif" import *;
cifsim(
   "Synth/output_SUP.cif",
   "-i gui",              // Interactive GUI input mode
   "--stateviz=1"        // Enable state visualization
);
```

Input Modes: The simulator supports multiple input modes:

- Interactive GUI (default): Graphical interface with color-coded event buttons
- Interactive text: Command-line based interaction
- Automatic: Random or programmed choice selection
- Semi-automatic: Hybrid mode where routine choices are automated

Note: The ESCET interactive simulator (GUI input mode) does not support input variables during step-by-step execution — the simulator requires the user to select events manually, while input variables imply external signal sources that do not exist in the simulation environment. For this reason, the workflow proceeds in two phases: (1) abstract synthesis and simulation are performed without input variable bindings (all `plant invariant` declarations commented out), allowing interactive validation of the supervisory logic; (2) input variable bindings are restored and the model is re-synthesized before code generation, producing deployment-ready code with hardware-coupled guards. This two-phase approach is detailed in Section 4.5.

Simulator Interface:

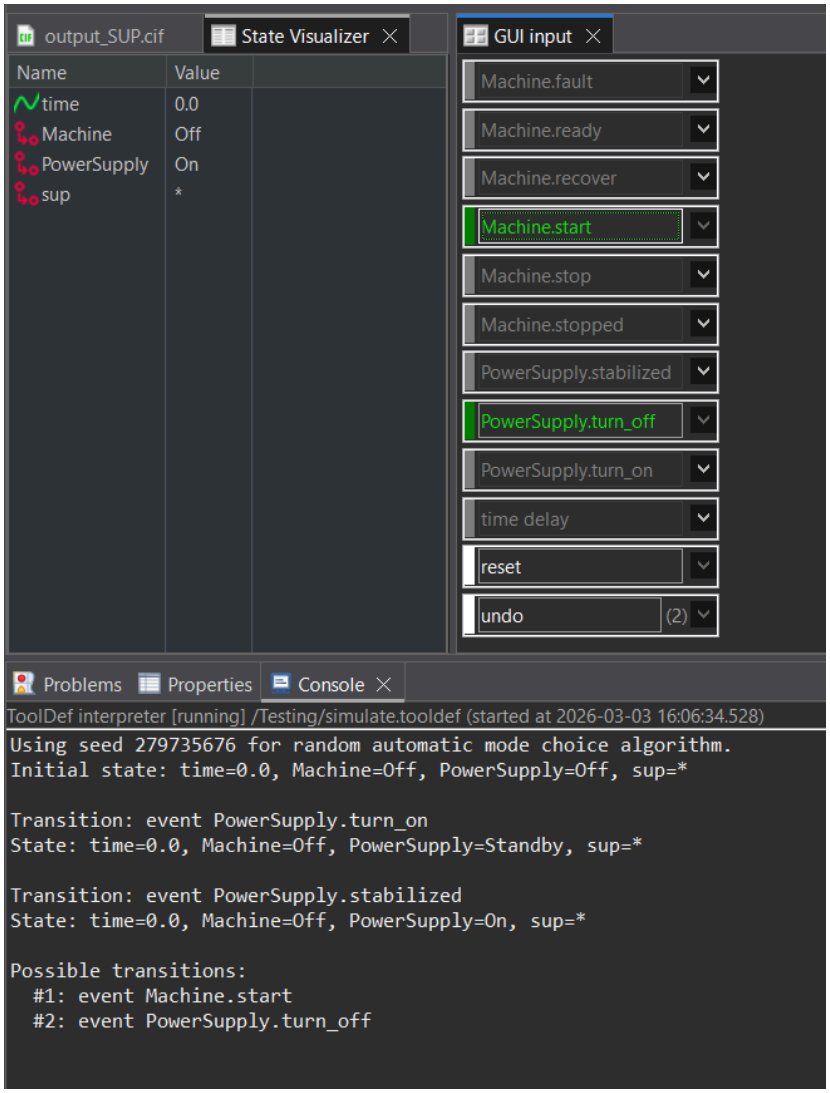


Figure 14: CIF simulator interface showing current state (locations) and events that can be enabled. Green buttons indicate controllable events, red indicates uncontrollable events. Users select events to execute, observing resulting state changes.

Simulation Workflow:

System starts at initial locations with initial variable values and it shows current locations, variable values, and events that can be enabled. User selects event (controllable or uncontrollable) and system updates locations/variables according to edge definitions , the user continues and checks possible behavior and requirement specific scenarios.

Concluding Remarks:

Interactive simulation provides intuitive validation before deployment. Engineers execute realistic scenarios, verifying that synthesized behavior matches intent and identifying modeling errors requiring refinement.

3.5.2 Controller Properties Verification

The Eclipse ESCET toolkit includes a Controller Properties Checker to verify essential properties of the synthesized supervisor before generating code from it. These properties ensure that the theoretical guarantees from synthesis are preserved in the practical, implemented controller. The checker assumes a specific execution scheme used by all ESCET code generators. The results are only valid if this scheme is followed, more information about the execution scheme is discussed in the next chapter.

Running the Checker:

```
cifcontrollercheck("Synth/output_SUP.cif -o Synth/output_SUP_checked.cif");
```

This command reads output_SUP.cif and produces output_SUP_checked.cif, which includes an annotation summarizing the check results.

Verified Properties [2]:

`Bounded Response`: Verifies that for every execution of the generated code, the number of iterations in the event loops for both uncontrollable and controllable events is bounded. This guarantees the absence of livelock (infinite loops) consisting purely of uncontrollable events or purely of controllable events, ensuring the control code always terminates its execution phases.

`Confluence`: Confirms that for the execution of controllable events, the order in which they are processed by the generated code does not matter—any order leads to the same final state. This ensures the implementation can schedule events arbitrarily without violating the synthesized supervisor's intent.

Note: This check may produce false negatives and does not check uncontrollable events.

`Finite Response`: Verifies that the number of transitions for controllable events in any execution is finite. This is a weaker form of the bounded response check, specifically for controllable events. It ensures no livelock exists for controllable events.

Note: Due to potential false negatives, the **bounded response** check is recommended instead.

`Non-blocking under Control`: Ensures that despite the specific, sequential way generated code executes events (first all uncontrollable events, then controllable events), a marked (terminal) state remains reachable. This confirms that the synthesis guarantee of non-blocking behavior is preserved in the implemented controller.

After performing the selected checks, the tool modifies the output model by adding a controller properties annotation that encodes the results. This model should not be altered afterward, as changes could invalidate the verified properties.

```
CONCLUSION:
    [OK] The specification has bounded response:
        - At most 1 iteration is needed for the event loop for uncontrollable events.
        - At most 1 iteration is needed for the event loop for controllable events.
    [OK] The specification is non-blocking under control.
    [OK] The specification has finite response.
```

```
    [ERROR] The specification may NOT have confluence:
        Confluence of the following event pair could not be decided:
          (Machine.start, PowerSupply.turn_off)

The model with the check results has been written to "Synth/output_SUP.cif".
```

Note: The specification does not have confluence, as can be seen in Figure 14: at a given instant, both controllable events Machine.`start` and PowerSupply.`turn_off` are enabled and will be executed. The order of execution matters, because if the machine is ON (i.e., the `start` event has occurred), then the power supply's `turn_off` event cannot be executed, and vice versa. This order dependence means the system lacks confluence.

3.6 Code Generation

The Eclipse ESCET toolkit [2] generates executable implementations from synthesized supervisors for deployment on embedded systems. The code generator translates the entire supervised system, plant automata, supervisor restrictions, and requirement invariants into a self-contained C library that can be called from any execution layer .

3.6.1 C99 Code Generation

Generation Command:

```
cifcodegen("Synth/output_SUP_checked.cif -o gen/ -l c99 -p SUP");
```

The `<prefix>` (here `SUP`) determines the naming of all generated files and symbols.

Generated Files:

- *`<prefix>_library.c`* / *`<prefix>_library.h`*:Generic runtime support for CIF data types (booleans, integers, reals, strings) and utilities functions. This code is identical for every generated project regardless of the model.
- *`<prefix>_engine.c`* / *`<prefix>_engine.h`*: The model-specific supervisor logic (state variables, edge functions and execution loop).
- *`<prefix>_test_code.c`*: A skeleton test harness showing how to call the engine.
- *`<prefix>_compile.sh`*: Build script tying everything together with `gcc` in C99 mode.
- *`<prefix>_readme.txt`*: Documentation of the generated code and its usage.

The engine files act as a library, they do not contain a `main()` function. The user's application code calls into them.

3.6.2 Generated Code Overview

3.6.2.1 State Representation

The code generator flattens all automaton locations from every plant into a single C enumeration. In *`<prefix>_engine.h`*, an `enum` contains literals for every location across all automata, for example `_SUP_Running`, `_SUP_Off`, `_SUP_On`, `_SUP_Standby`, and so on, all merged under a common `_<prefix>_` naming prefix.

Each automaton then becomes a global variable of this enum type. In our case, `Machine_` and `PowerSupply_` are the two state variables tracking which location each automaton

currently occupies. CIF input variables (here `fault_flag_`) become `extern` globals that the environment writes into before each execution cycle.

3.6.2.2 Edge Functions — The Guard-Update Pattern

Every CIF edge is translated into a static `execEdgeN()` function following a fixed template:

```
/*Execute code for edge with index 7 and event "PowerSupply.turn_off".
 @return Whether the edge was performed.*/
static BoolType execEdge7(void) {
    BoolType guard = ((PowerSupply_) == (_SUP_On)) && ((Machine_) == (_SUP_Off));
    if (!guard) return FALSE;
    #if EVENT_OUTPUT
        SUP_InfoEvent(PowerSupply_turn_off_, TRUE); //pre info
    #endif
    PowerSupply_ = _SUP_Off;

    #if EVENT_OUTPUT
        SUP_InfoEvent(PowerSupply_turn_off_, FALSE); // post info
    #endif
    return TRUE;
}
```

The guard expression is the point where the code generator merges two elements into a single boolean evaluation: the plant's inherent topology (identifying which locations permit which transitions) and the supervisor's safety constraints derived from the specifications.

For instance, the `PowerSupply.turn_off` edge has this guard:

- `(PowerSupply_ == _SUP_On) && (Machine_ == _SUP_Off)`.

The initial segment `(PowerSupply_ == _SUP_On)` originates directly from the plant — power can only be shut off when it is already in the `On` condition. The second component `(Machine_ == _SUP_Off)` originates from requirement R2 — a power shutdown is secure only when the machine is entirely in `Off`.

The produced code no longer differentiates between "plant guards" and "supervisor guards." Synthesis has already combined them during processing. As you introduce input variables, these guards inherently become more intricate.

3.6.2.3 The Execution Loop — `PerformEdges()`

This function implements the CIF controller execution scheme, and its structure is always the same regardless of the model:

1. **Uncontrollable events first**: a loop tries all uncontrollable event edges (e.g., `Machine.fault`, `Machine.ready`, `Machine.stopped`, `PowerSupply.stabilized`) repeatedly until none can fire. This lets the plant environment settle completely before the controller acts.
2. **Controllable events second**: a separate loop tries all controllable event edges (e.g., `Machine.recover`, `Machine.start`, `Machine.stop`, `PowerSupply.turn_off`, `PowerSupply.turn_on`) repeatedly until none can fire. This is the supervisor issuing its commands.

```c
/** Repeatedly perform discrete event steps, until no progress can be made any more. */
static void PerformEdges(void) {
    int count = 0;/* Uncontrollables. */
    for (;;) {
        count++;
        if (count > MAX_NUM_ITERS) { /* 'Infinite' loop detection. */
            fprintf(stderr, "Warning: Quitting after performing %d uncontrollable events, infinite loop?\n", count);break;
        }
        BoolType edgeExecuted = false;
        edgeExecuted |= execEdge0(); /* (Try to) perform edge with index 0 and event "Machine.fault". */
        edgeExecuted |= execEdge1(); /* (Try to) perform edge with index 1 and event "Machine.ready". */
        edgeExecuted |= execEdge2(); /* (Try to) perform edge with index 2 and event "Machine.stopped". */
        edgeExecuted |= execEdge3(); /* (Try to) perform edge with index 3 and event "PowerSupply.stabilized". */
        if (!edgeExecuted) {break; /* No edge fired, done with discrete steps. */}
    }
    /* Controllables. */count = 0;
    for (;;) {
        count++;
        if (count > MAX_NUM_ITERS) { /* 'Infinite' loop detection. */
            fprintf(stderr, "Warning: Quitting after performing %d controllable events, infinite loop?\n", count);break;
        }
        BoolType edgeExecuted = false;
        edgeExecuted |= execEdge4(); /* (Try to) perform edge with index 4 and event "Machine.recover". */
        edgeExecuted |= execEdge5(); /* (Try to) perform edge with index 5 and event "Machine.start". */
        edgeExecuted |= execEdge6(); /* (Try to) perform edge with index 6 and event "Machine.stop". */
        edgeExecuted |= execEdge7(); /* (Try to) perform edge with index 7 and event "PowerSupply.turn_off". */
        edgeExecuted |= execEdge8(); /* (Try to) perform edge with index 8 and event "PowerSupply.turn_on". */
        if (!edgeExecuted) { break; /* No edge fired, done with discrete steps. */}
    }
}
```

Both loops include a `MAX_NUM_ITERS` counter (default 1000 after properties checker we can adjust). The uncontrollable-first ordering is mandated by the CIF controller properties execution scheme and guarantees the supervisor always observes the full plant response before making decisions.

3.6.2.4 The Two Entry Points

The engine always exposes exactly two functions:

- `<prefix>_EngineFirstStep()` — called once at system startup. It initializes all state variables to their CIF initial locations (e.g., `Machine_ = _SUP_Off`, `PowerSupply_ = _SUP_Off`), calls `<prefix>_AssignInputVariables()` to read the initial sensor state, then runs `PerformEdges()` to reach a stable initial configuration. This should only be called when the physical system is also being (re-)initialized.
- `<prefix>_EngineTimeStep(double delta)` — called repeatedly by the supervisory layer after each time period `delta`. It reads fresh inputs, advances `model_time` by `delta`, then runs `PerformEdges()`. In models with continuous variables, this is where Euler integration updates them; in purely discrete models like ours, it only updates the clock and re-evaluates edges.

3.6.3 Resource Layer Integration

Since the input variables and their **`plant invariant`** bindings are included in the synthesis model, the generated code already contains the fused guards.The generated supervisor integrates with the resource layer through callback functions that the user must implement externally. These form the integration between the synthesized controller and the physical system.

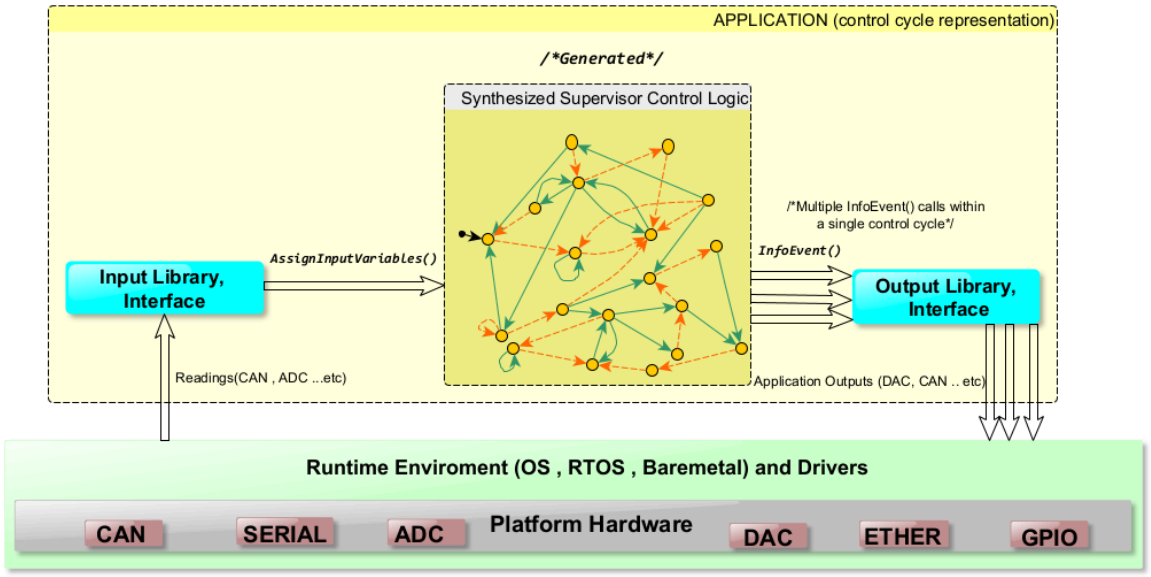


Figure 15: Application Topology Illustration

<prefix>_AssignInputVariables() is called by the engine at the start of every execution cycle, before any edge evaluation. The user's Input library implementation as seen in Figure 15 exposes current information values to be written into the input variables:

```
void SUP_AssignInputVariables(void) {
    fault_flag_      = read_fault_sensor();
    ready_flag_      = read_velocity() > READY_THRESHOLD;
    stopped_flag_    = read_velocity() < STOP_THRESHOLD && brake_engaged();
    stabilized_flag_ = voltage_within_tolerance();
    stop_cmd_        = read_button(); //if stop button is pressed
}
```

This callback is invoked by both `<prefix>_EngineFirstStep()` and `<prefix>EngineStep()`. Input variables are sampled once per cycle and held constant throughout the entire `PerfomrEdges()` execution, they do not change between the uncontrollable and controllable loops. `<prefix>_InfoEvent(<prefix>EventEnum event ,BoolType pre)` is called before `(pre==TRUE)` and after `(pre==FALSE)` each event execution. The `pre` call exposes the state *before transition* and the post call confirms it happened and we are updated. This is the primary mechanism for forwarding supervisor decisions to resource layer.For instance on a post event for `PowerSupply_turn_on_`, the resource layer would energize the physical power supply relay. Compiled in only when `EVENT_OUTPUT` is defined.

```
void SUP_InfoEvent(SUP_Event_ event, BoolType pre) {
    if (!pre) { //care only about post
        switch (event) {
            case PowerSupply_turn_on_:  energize_relay();        break;
            case PowerSupply_turn_off_: de_energize_relay();     break;
            case Machine_start_:        enable_motor_drive();    break;
            case Machine_stop_:         begin_controlled_stop(); break;
            case Machine_recover_:      clear_fault_latch();     break;
            default: break;}}}
```

Integration pattern:

A typical deployment would be periodic execution, with the supervisor's `<prefix>_EngineTimeStep(delta)` called at a fixed period. Within each call, `<prefix>_AssignInputVariables()` samples all sensor inputs, `PerformEdges()` runs both event loops (uncontrollable first, then controllable), and `<prefix>_InfoEvent()` callbacks commands to the specified interface. It is important to note that the supervisor itself must remain a synchronous, single-threaded state machine, we must not preempt or interrupt its own execution.

Test:

Having compiled and included the generated libraries we created a test where we change the input flags and we EngineStep() each time, essentially recreating each control cycle. Via the InfoEvent() we log the events that happen in each step. In the Figure 16 we see the input flags and and how the system responded.

```
Supervisor Validation  CIF synthesis -> C99 generated code

Requirements:  R1: Machine.start needs PowerSupply.On
               R2: PowerSupply.turn_off needs Machine.Off
               R3: Machine.stop needs PowerSupply.On

Flags:  F=fault  R=ready  S=stopped  P=stabilized  C=stop_cmd
       green = ON    dim = OFF
Events: controllable (supervisor)    uncontrollable (plant)

+-----+-----------------+-----------+---------+-------+----------------------+--------------------------------------+
| Cyc | Test            | Machine   | PSU     | FRSPC | Events               | Result                               |
+-----+-----------------+-----------+---------+-------+----------------------+--------------------------------------+
|   0 | initial state   | Off       | Off     | FRSPC | -                    | ok before the 1st PerformEdges()     |
|   0 | power on        | Off       | Standby | FRSPC | turn_on              | ok turn_on fires (controllable)      |
|   1 | R1: blocked     | Off       | Standby | FRSPC | -                    | ok PSU = Standby, start blocked      |
|   2 |   waiting...    | Off       | Standby | FRSPC | -                    | ok stabilized_flag still OFF         |
|   3 | R1: enabled     | Starting  | On      | FRSPC | stabilized, start    | ok P on: stabilized + start fire     |
|   4 |   waiting...    | Starting  | On      | FRSPC | -                    | ok ready_flag still OFF              |
|   5 |   waiting...    | Starting  | On      | FRSPC | -                    | ok motor spinning up                 |
|   6 | input gating    | Running   | On      | FRSPC | ready                | ok R on: ready fires, now Running    |
|   7 |   running...    | Running   | On      | FRSPC | -                    | ok no stop_cmd, supervisor idle      |
|   8 | R3: enabled     | Stopping  | On      | FRSPC | stop                 | ok C on: stop fires (PSU = On)       |
|   9 | R2: blocked     | Stopping  | On      | FRSPC | -                    | ok Machine = Stopping, no turn_off   |
|  10 |   decelerating...| Stopping  | On      | FRSPC | -                    | ok stopped_flag still OFF            |
|  11 |   decelerating...| Stopping  | On      | FRSPC | -                    | ok motor slowing down                |
|  12 | R2: enabled     | Starting  | On      | FRSPC | stopped, start       | ok S on: stopped + start chain       |
|  13 | fault: blocked  | Starting  | On      | FRSPC | -                    | ok fault_flag OFF, no fault          |
|  14 |   waiting...    | Starting  | On      | FRSPC | -                    | ok no fault signal                   |
|  15 | fault: fires    | Starting  | On      | FRSPC | fault, recover, start| ok F on: fault + recover + start     |
|  15 | event chain     | Starting  | On      | FRSPC | fault, recover, start| ok 3 events in 1 PerformEdges()      |
|  16 | isolation       | Starting  | On      | FRSPC | -                    | ok all flags OFF, zero events        |
|  17 |   waiting...    | Starting  | On      | FRSPC | -                    | ok supervisor idle                   |
+-----+-----------------+-----------+---------+-------+----------------------+--------------------------------------+
```

Figure 16: Test Result.

Concluding Remarks:

This chapter presented the synthesis-based engineering workflow using ESCET:

CIF Modeling Language provides Extended Finite Automata specification with:

- Locations and edges for discrete state machines
- Controllable and uncontrollable event classification
- Discrete variables with bounded domains
- Guards and updates for data-dependent behavior
- Plant models (what CAN happen) and requirement models (what SHOULD happen)

Supervisor Synthesis automatically computes controllers that:

- Enforce all requirements through guard strengthening
- Guarantee controllability, nonblocking, maximal permissiveness
- Operate on symbolic EFA representation enabling industrial-scale systems

Simulation and Verification validate controlled systems through:

- Interactive step-by-step execution
- Scenario-based testing
- Automated property checking.

Code Generation produces deployable implementations:

- C99 code for embedded systems
- State representation, initialization, event execution, availability queries
- Integration with resource layer through event-based interface

The next chapter demonstrates this workflow through the distributed BLDC control network case study, showing complete system implementation from requirements through deployed embedded code.

4 Evaluation: *Distributed BLDC Control Network*

4.1 System Overview

The evaluation platform is a distributed motor control network in which a central supervisor coordinates multiple motor drive nodes over a shared CAN bus. Rather than hand-writing coordination logic, the supervisor is automatically generated from formal plant and requirement specifications using the Eclipse ESCET toolchain, as described in Section 3.

4.1.1 Design Philosophy: Bottom-Up Formal Integration

This work follows a bottom-up engineering approach where existing, proven hardware and firmware are integrated through formal modeling:

1. **Hardware Layer (Existing):** Each motor drive node is a self-contained unit built from commercial components — DRV8302 gate driver, BLDC motor, Teensy 4.0 microcontroller. The hardware provides fault detection through diagnostic pins (nFAULT, nOCTW, PWRGD).
2. **Control Layer (Implemented):** Each node runs the SimpleFOC control library for motor actuation and coupled with a local state machine we implemented that handles safety-critical reactions always providing safe states. This firmware is not generated — it is hand-written and battle-tested.
3. **Communication Layer (Implemented):** Nodes expose their events and accept commands via CANCommander, a register-based CAN protocol. Every state transition is observable; every command maps to a CAN register write.
4. **Supervisor (Implemented):** The existing node behavior is abstracted into a CIF plant model. Application-specific coordination requirements are specified formally. The supervisor that enforces these requirements across multiple nodes is then synthesized automatically.

The key insight: the nodes already know how to protect themselves (local safety). The synthesized supervisor orchestrates them to achieve system-level goals (coordinated operation, fault recovery, resource sharing).

4.1.2 Physical Architecture

The network, shown in Figure 17, consists of three classes of participant: the supervisor, the motor drive nodes, and the operator workstation. All motor drive nodes share a common 12 V DC power rail supplied by a 40 A switched-mode power supply, and communicate over a CAN bus operating at 1 Mbit/s with 29-bit extended identifiers. The bus is terminated at both ends with 120 Ω resistors per the CAN specification. Each bus participant interfaces to the physical CAN_H and CAN_L differential pair through an SN65HVD230 transceiver.

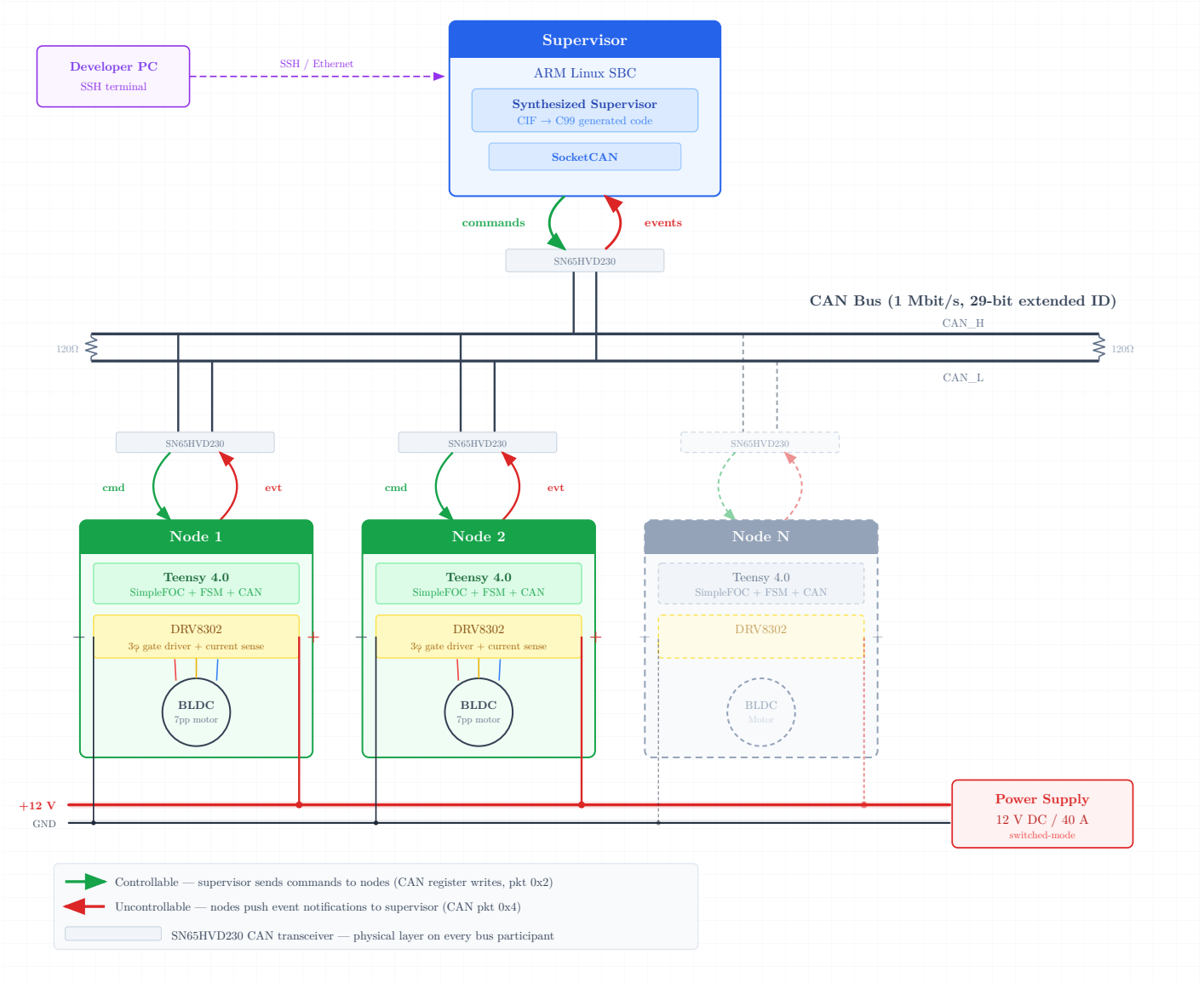


Figure 17: Network System Architecture

4.1.2.1 Supervisor

The supervisor node is an ARM-based Linux single-board computer (MPU). It runs the synthesized supervisor as a standalone C99 application, generated directly from the CIF specification by the ESCET code generation toolchain. The supervisor accesses the CAN bus through the Linux SocketCAN interface, which abstracts the hardware transceiver behind a standard socket API. It issues commands to nodes and receives event notifications from them, maintaining a consistent view of the global system state at all times. The operator interacts with the supervisor remotely via an SSH connection over Ethernet.

4.1.2.2 Motor Drive Nodes

Each motor drive node is a self-contained actuator unit whose design is detailed in Section 4.2.4. At a high level, each node consists of:

- **Computation:** Teensy 4.0 MCU (ARM Cortex-M7, 600 MHz)
- **Power:** DRV8302 three-phase gate driver with integrated fault detection

- **Actuation:** 7 pole-pair 1KV BLDC motor
- **Control:** SimpleFOC library running field-oriented control at ≈20 kHz
- **Safety:** Local state machine with static transition table
- **Communication:** CANCommander register protocol

A critical feature is that each node implements its own state machine locally. When a fault is detected, the node disables the motor immediately and transitions to a safe state before notifying the supervisor. The safety-critical reaction path never depends on network communication. The supervisor learns about faults after the fact and coordinates recovery across nodes, but the immediate protective action is taken locally.

This separation, where nodes handle local safety and the supervisor handles system-level coordination is a central design decision and is discussed in detail in Section 4.2.4.1.

4.1.2.3 Communication Protocol

The CAN protocol multiplexes two logical channels on the same physical bus using the 29-bit extended identifier format. The identifier layout allocates bits 27–20 to the node address, bits 19–16 to the packet type, bits 15–8 to the register or event number, and bits 7–0 to the motor index or sequence number.

The first channel uses the CANCommander register-based protocol (detailed in Table 8) with packet types `0x1` (read request), `0x2` (write request), and `0x3` (read response). This channel carries all controllable actions:

- The supervisor writes to specific registers to request calibration, enable or disable a motor, set a target velocity, initiate fault recovery, or reboot a node. Each register maps directly to one controllable event in the CIF plant model.
- The second channel uses packet type `0x4` for event push notifications. When a node undergoes a state transition triggered by an internal or environmental cause, a fault detection, a completed calibration, a motor reaching standstill, or a power loss, it pushes a zero-payload CAN frame whose identifier encodes the event type and a sequence number. These notifications map directly to uncontrollable events in the CIF model.

This one-to-one correspondence between the physical communication and the formal model is what allows the synthesized supervisor to operate correctly on the real hardware. Every controllable event the supervisor can fire corresponds to a specific register write, and every uncontrollable event the supervisor must observe corresponds to a specific CAN notification arriving on the bus.

4.1.3 Demonstration Application: Balanced Ventilation System

While the architecture supports arbitrary N motor nodes on a CAN bus, this work demonstrates the approach on a dual-motor (N=2) **balanced ventilation system**, specifically a Heat Recovery Ventilator (HRV) [14]. This application is detailed in Section 4.3, after the node hardware and firmware have been fully explained.

At a high level, the system comprises two brushless DC (BLDC) fans operating in coordinated opposition:

- **NODE1 (Supply Fan):** Draws fresh outdoor air into the building
- **NODE2 (Exhaust Fan):** Expels stale indoor air from the building

Safety Requirement: Supply CFM ≈ Exhaust CFM (±10% tolerance) [15].

Airflow imbalance creates building pressure differential, which can cause backdrafting of combustion appliances (CO poisoning risk) [15], inadequate fresh air exchange, and HVAC inefficiency. Both fans **MUST** operate together or not at all. A fault in either fan triggers immediate coordinated shutdown of both.

This safety-critical coordination requirement, where independent node failures must trigger system-level responses , makes balanced ventilation an suitable demonstration case for formal supervisor synthesis.

The remainder of this chapter is organized as follows: Section 4.2 describes the motor drive node hardware and firmware in detail. Section 4.3 presents the balanced ventilation application requirements. Section 4.4 develops the formal CIF models and requirements. Section 5 presents the synthesis results and experimental validation.

4.2 Motor Drive Node

4.2.1 Gate Driver

The physical drive platform is commercially available three-phase inverter module based on the Texas Instruments DRV8302 [16], commercially procured as a pre-assembled board. The board integrates DRV8302 gate driver IC, six N-channel power MOSFETs (NCE80H11D,V_{DS} = 80V, $R_{DS(on)}$ = 11mΩ typical), three external differential current sense amplifiers with a fixed gain of 12.22 V/V, three back-EMF sampling circuits and a integrated buck converter output. The board is rated for a supply voltage of 6-45V and a continuous phase current of 15A (27 A peak) without forced cooling [17].

The DRV8302′s own internal dual current shunt amplifiers (SO1, SO2) are also brought out to header pins, but the three-phase current measurement used in this work is performed by the external amplifier chain rather than the internal amplifiers, for reasons of input common-mode range and noise immunity at the switching node.

Role of the Gate Driver in the System Model

The DRV8302 is not a transparent pass-through element in the system model. Its internal protection logic — thermal, voltage, and current — generates discrete, observable transitions that render the power stage unavailable independently of the control algorithm. These transitions must be captured to correctly characterise the reachable states of the drive and the conditions under which re-energisation is permissible. The operating point relevant to this work is fixed by the state of the mode-select pins at power-on, summarised in Table 2 . Each choice either constrains or simplifies the fault model developed in Section 4.4.

Table 2: Mode-select pin configuration and its consequences for the fault model.

Pin	State	Consequence for System
M_PWM	HIGH	3-PWM mode. One PWM channel per phase; complementary low-side drive with hardware dead-time insertion.
M_OC	LOW	Cycle-by-cycle current limiting. Over-current events are absorbed at the hardware level without latching `nFAULT`. Over-current is therefore not a discrete fault event in this model.
GAIN	LOW	Internal current sense amplifier gain fixed at 10 V/V. Phase current measurement in this work uses the external amplifier chain at 12.22 V/V; the internal amplifiers are available as a secondary path.
OC_ADJ	HIGH	Over-current comparator threshold raised to VREF — effectively disabled. Combined with `M_OC = LOW`, `nOCTW` becomes thermally exclusive: it can only be asserted by the over-temperature warning circuit. This is the prerequisite for unambiguous fault (board bad noise immunity)
DC_CAL	LOW	Normal current sense operation.

Protection Architecture and Observable Outputs

The DRV8302 exposes three status signals that form the observable interface between the protection hardware and the control system. All three are **open-drain** outputs requiring external pull-up resistors to DVDD (3.3 V)

`nFAULT` — Gate Shutdown and Latch `nFAULT` is asserted LOW whenever a **protection event** causes the device to shut down the gate drivers. The datasheet states: *"During fault shut down conditions, all gate driver outputs will be kept low to ensure external FETs at high impedance state"* [16]. The gate drive outputs (GH_x, GL_x) are driven actively LOW — not floated — placing all six MOSFETs in high-impedance. No phase current can flow. Switching ceases within `200ns` of fault detection.

Table 3: `nFAULT` assertion conditions, thresholds, and clear mechanisms [SLVSA73].

Condition	Threshold	Type	Clear Mechanism
PVDD undervoltage (PVDD_UV)	< 6 V	Auto	Clears automatically when PVDD rises above threshold. No EN_GATE cycle required.
GVDD undervoltage (GVDD_UV)	< 8 V	Auto	Charge pump output below threshold. Clears automatically when GVDD recovers.
GVDD overvoltage (GVDD_OV)	> 16 V	Latching	EN_GATE reset required (pulse LOW ≥ 5 µs, then HIGH; device ready within 10 ms). Indicates charge pump fault or gate supply spike.
Thermal shutdown (OTSD)	150 °C	Latching	EN_GATE reset required, but latch will not clear until die has cooled below 130 °C. Attempting reset above this temperature has no effect [SLVSA73].

`nOCTW` — Under this configuration `nOCTW` has a single assertion condition: The **over-temperature warning** (OTW) at 130 °C, clearing automatically below 115 °C.The 15 °C hysteresis prevents oscillation near threshold. Because `OC_ADJ = HIGH` disables the over-current comparator path, `nOCTW` is thermally exclusive, a property exploited for fault classification.

`PWRGD` — Is an open-drain output driven LOW when the regulated buck output falls below 92% or exceeds 109% of its nominal value, de asserting when output recovers to 94–107%. The buck (PVDD2, min 3.5 V) and gate driver (PVDD1, min 8 V) supply pins are independent on the DRV8302 but are tied to the same motor rail on this board. The minimum rail voltage for buck regulation of the 3.3 V output is just above 3.3 V — well below the 6 V PVDD_UV gate driver lockout threshold. Consequently `PWRGD` falling LOW is a **leading indicator** of an impending gate driver UVLO: as the motor supply falls, the buck output droops and `PWRGD` asserts before the PVDD rail reaches 6 V. `PWRGD` is therefore the appropriate primary trigger for a controlled shutdown sequence, providing earlier warning than `nFAULT`/ PVDD_UV alone.

4.2.1.1 Fault Information

PVDD_UV and GVDD_UV clear within microseconds to milliseconds of supply recovery. A 50 ms observation window on `nFAULT` guarantees that any fault still active after this interval is a latched condition, allowing transient supply perturbations to be ignored without explicit cause identification. After the 50 ms window the joint pin state maps injectively onto three fault classes (Table 4).

Table 4: Fault classification by pin state after 50 ms confirmation window. × = don't-care. Gate behaviour is identical for all three classes [SLVSA73].

PWRGD	nFAULT	nOCTW	Fault Class	Gate Driver State
LOW	×	×	**Supply failure.** Buck output outside regulation window. PVDD_UV will follow as the shared rail continues to collapse.	All gate outputs LOW → all six NCE80H11D MOSFETs HiZ. No phase current.
HIGH	LOW	LOW	**Thermal shutdown (OTSD).** Die at 150 °C. `nOCTW` LOW confirms thermal origin (die had already exceeded 130 °C). Latch held until die cools below 130 °C.	All gate outputs LOW → all MOSFETs HiZ. No phase current.
HIGH	LOW	HIGH	**GVDD overvoltage (GVDD_OV).** Charge pump exceeded 16 V. Typical causes: damaged bootstrap capacitor, inductive spike on GVDD rail.	All gate outputs LOW → all MOSFETs HiZ. No phase current.

The mapping is injective because:

1. OTSD always co-asserts OTW — the die must pass through 130 °C before 150 °C, so `nOCTW` is LOW when OTSD fires.
2. GVDD_OV has no path to assert `nOCTW`
3. `OC_ADJ = HIGH` prevents over-current from asserting `nOCTW`.

The three patterns are mutually exclusive and exhaustive over all latched faults reachable in this system.

Recovery Constraints

- For **OTSD**: `EN_GATE` reset is only accepted after the die cools below 130 °C. `nOCTW` returning HIGH is the observable confirmation [16].
- For **GVDD_OV**: `EN_GATE` reset clears the latch only if the overvoltage source is resolved. `nFAULT` remaining LOW after reset indicates a persistent hardware fault and that re-energisation is not safe.

In both cases the reset sequence is: EN_GATE LOW for ≥ 5 µs then HIGH; device ready within 10 ms [16].

4.2.2 Resource Layer Control Framework (SimpleFOC)

Motor actuation in this work is implemented using the SimpleFOC library [17], an open-source field-oriented control framework for BLDC and stepper motors targeting embedded platforms from Arduino UNO to Teensy and STM32. The library separates motor control into three independent layers — **Modulation/Driver**, **Torque Controller**, and **Motion Controller** — each configurable without touching the others. Table 5 summarises the available modes and the selections made in this work.

Table 5: SimpleFOC control layers and available modes. ✓ marks the configuration used in this work. (CS) requires a current sensor; (PS) requires a position sensor.

TorqueControlType	MotionControlType	FOCModulationType
Voltage ✓	Velocity Open-Loop ✓	SinePWM
DC Current (CS)	Velocity (PS)	Space Vector PWM ✓
FOC Current (CS)	Angle Open-Loop	Trapezoid 120°
	Angle (PS)	Trapezoid 150°

The selected combination — `velocity_openloop`, `TorqueControlType::voltage`, and `SpaceVectorPWM` — requires no position sensor and no current sensor for basic operation. The following sections develop the theory behind each layer in the order the control pipeline executes: angle generation, torque control, and PWM modulation.

4.2.2.1 Open-Loop Velocity Control

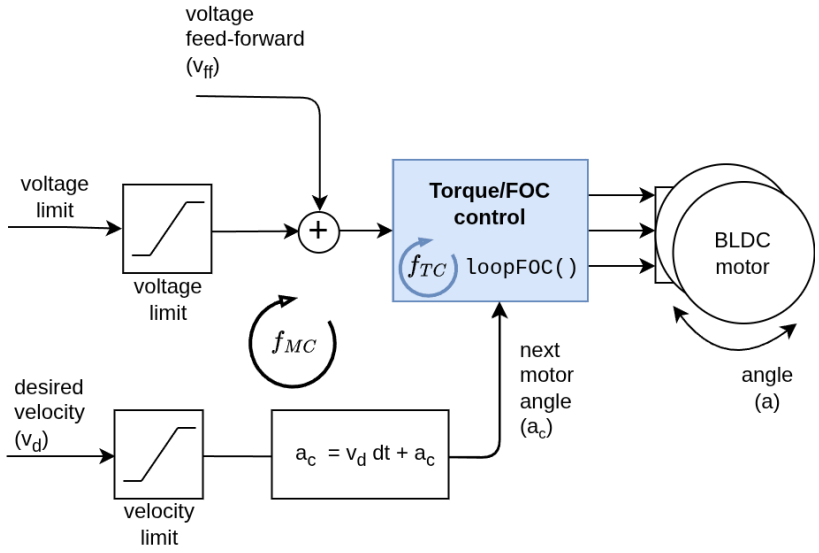


Figure 18: Open-loop velocity control with voltage torque mode Source: SimpleFOC docs

Standard FOC requires two angles: the **mechanical angle** θ_m (shaft position in radians) and the **electrical angle** θ_e (position of the stator flux vector within one electrical cycle). In closed-loop operation both are measured from sensors, whilst in open-loop operation both are computed entirely in software with no physical measurement is needed.

The two angles are related by the motor's pole-pair count n_{pp}:

$$\theta_e = n_{\text{pp}} \cdot \theta_m \tag{3}$$

One full shaft revolution corresponds to n_{pp} complete electrical cycles, so the stator flux vector must rotate n_{pp} times for every single rotation of the rotor. All coordinate transforms in the FOC pipeline — Park, inverse Park, and the SVPWM sector selection — operate on θ_e exclusively [17] .This reflects the physical periodicity of the motor.

On every call to `motor.move()`, the library advances θ_m by the displacement the rotor *would* cover in elapsed time dt at the commanded velocity v_d:

$$\theta_m \leftarrow \theta_m + v_d, dt \tag{4}$$

The time step dt is measured as the elapsed time between consecutive `motor.move()` calls, keeping integration correct even when the loop period is not constant. θ_e is then derived immediately from the updated θ_m.Since neither angle is ever corrected by a measurement, any slip between the assumed and actual rotor position accumulates silently. The rotor is assumed to track the rotating magnetic field by magnetic coupling alone.[17]

4.2.2.2 Voltage Torque Control

`TorqueControlType::voltage` treats the motor analogously to a DC motor: the user sets a target voltage U_q in the rotating (d, q) frame and the library computes the phase voltages that produce a stator flux vector orthogonal to the rotor's permanent-magnetic field. The electrical angle used to orient this vector is the synthetic θ_e produced by the motion controller above — this is the direct handoff point between the two layers.

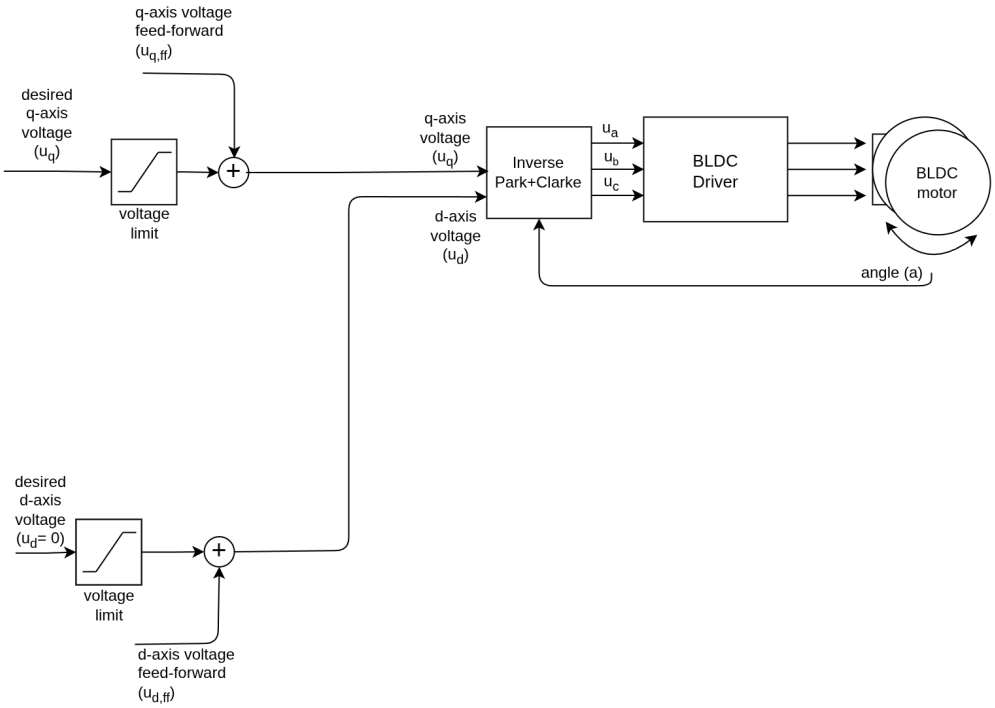


Figure 19: Diagram of the voltage torque controller. Source:SimpleFOC docs

The actual behaviour of this mode depends on which motor parameters are provided, activating one of four progressively more accurate sub-modes:

Table 6: Voltage torque control sub-modes in SimpleFOC. Each level activates automatically when the corresponding parameters are supplied before `motor.init()`.

Level	Parameters	Voltage applied	Practical limitation
0	None	$U_q = U_{\text{target}}$	Current fully uncontrolled; depends on speed and load
1	R	$U_q = I_d R$	Accurate only at standstill; back-EMF reduces current as speed rises
2	R, K_V	$U_q = I_d R + v/K_V$	Steady-state accurate; inductance lag neglected
3	R, K_V, L	$U_q = I_d R + v/K_V + L \cdot dI/dt$	Most accurate; valid also during current transients

In this work the motor is initialised with R and K_V, corresponding to **Level 2**. This is appropriate for open-loop velocity control where the velocity profile is smooth and current transients are negligible, making the inductance lag term of Level 3 unnecessary. At steady state the applied voltage is:

$$U_q = I_d R + \frac{v}{K_V} \tag{5}$$

The first term offsets the resistive drop; the second compensates back-EMF so that the effective current stays near I_d regardless of speed. At low speed the back-EMF term is negligible and the resistive term dominates; at high speed both terms contribute and together prevent the current from collapsing.

Two limits are enforced independently to protect the hardware. `voltage_limit` is an absolute ceiling in volts; `current_limit` is converted at runtime to a velocity-dependent voltage cap via the same model:

$$U_{\text{applied}} = \min\left(U_{\text{voltage_limit}},\quad I_{\text{lim}} \cdot R + \frac{v_d}{K_V}\right) \tag{6}$$

The current-derived cap adapts with speed while `voltage_limit` provides an unconditional backstop. Since there is no current feedback loop the correspondence between I_d and actual winding current is only as accurate as the identified parameters — a limitation acknowledged explicitly and discussed further in the context of the current sensing hardware in .

4.2.2.3 Space Vector PWM Modulation

Given θ_e and U_q, `motor.loopFOC()` synthesises the three PWM duty cycles through a two-step process.

Inverse Park transform. With $U_d = 0$ in voltage mode, the (d, q) command is mapped to the stationary $\alpha\beta$ frame:

$$U_\alpha = -\sin(\theta_e), U_q, \quad U_\beta = \cos(\theta_e), U_q \tag{7}$$

The reference vector $\boldsymbol{V}_{\text{ref}} = U_\alpha + jU_\beta$ has constant magnitude U_q and rotates at electrical frequency $\omega_e = v_d \cdot n_{\text{pp}}$, producing the continuously rotating stator field required for torque production.

Sector decomposition and duty-cycle. The six active switching vectors of the two-level inverter ($\boldsymbol{V}_1$–$\boldsymbol{V}_6$, at 60° intervals) partition the $\alpha\beta$ plane into six sectors. Over each PWM period T_s, the reference vector is approximated by time-weighting the two adjacent active vectors and a zero vector:

$$\boldsymbol{V}_{\text{ref}}, T_s = \boldsymbol{V}_k, t_k + \boldsymbol{V}_{k+1}, t_{k+1} + \boldsymbol{V}_0, t_0, \quad t_k + t_{k+1} + t_0 = T_s \tag{8}$$

With `SpaceVectorPWM`, the zero-vector time is split symmetrically between $\boldsymbol{V}_0$ and $\boldsymbol{V}_7$, centering each phase pulse in the PWM period. This is equivalent to injecting a third-harmonic offset $\frac{1}{6}\sin 3\theta_e$ into all three duty cycles, which cancels in every line-to-line voltage while extending the linear modulation range to the full bus voltage V_{dc} — a 15.5% improvement over Sine PWM at no harmonic cost to the phase currents.

Table 7: SinePWM vs. SpaceVectorPWM for a two-level three-phase inverter. ✓ denotes the mode used in this work.

Property	SinePWM	SpaceVectorPWM ✓
Peak line-to-line voltage	$\frac{\sqrt{3}}{2}V_{\text{dc}}$	V_{dc}
DC bus utilisation	86.6 %	100 %
Third-harmonic injection	None	Implicit, $\frac{1}{6}\sin 3\theta_e$
Zero-vector distribution	Asymmetric	Symmetric ($\boldsymbol{V}_0$ / $\boldsymbol{V}_7$ split)

```
BLDCDriver3PWM driver(PWM_A, PWM_B, PWM_C);
BLDCMotor motor(
    POLE_PAIRS,        // pole pairs (hardcoded)
    PHASE_RESISTANCE,  // R [Ω]      from characteriseMotor()
    MOTOR_KV,          // KV [RPM/V] hardcoded
    PHASE_INDUCTANCE   // L [H]      from characteriseMotor()
);

void setup() {
    driver.voltage_power_supply = 12.0;   // [V]
    driver.pwm_frequency        = 20000;  // 20 kHz
    driver.init();
    motor.linkDriver(&driver);
    motor.foc_modulation    = FOCModulationType::SpaceVectorPWM;
    motor.controller        = MotionControlType::velocity_openloop;
    motor.torque_controller = TorqueControlType::voltage;
    motor.voltage_limit     = VOLTAGE_LIMIT;
    motor.current_limit     = CURRENT_LIMIT;
    motor.target            = 0;
    motor.init();
}
void loop() {
    motor.loopFOC();   // angle → transforms → PWM
    motor.move();      // integrate θ_m, derive θ_e
}
```

4.2.2.4 Motor Parameterisation and Calibration

The library requires four motor parameters: pole-pair count `POLE_PAIRS`, phase resistance R, phase inductance L, and motor constant K_V. Pole pairs and K_V are hardcoded, as their identification requires a position or speed sensor not available in this setup.

Phase resistance and inductance are identified experimentally using the library's built-in routine `motor.characteriseMotor(voltage)`, which applies controlled voltage steps and measures the current response through the linked current sensor. Resistance is extracted from the steady-state response:

$$R = \frac{V_{\text{applied}}}{I_{\text{steady}}} \tag{9}$$

Inductance is estimated from the current transient:

$$L = \frac{V_{\text{applied}} \cdot dt}{dI} \tag{10}$$

where dI is the current change over the observed delay dt. The routine must be called after the current sensor is linked to the motor object.

Concluding Remarks:

The SimpleFOC stack adopted in this work — open-loop velocity control, voltage torque mode, and Space Vector PWM — provides a fully functional, sensor-free actuation baseline. The two synthetic angles θ_m and θ_e replace sensor readings throughout the entire FOC pipeline: θ_m is integrated from the velocity command and θ_e drives every coordinate transform and sector selection. The dual (V/I) limit scheme keeps operation within hardware bounds at all speeds without closed-loop feedback, and SVPWM recovers the full DC bus headroom at no harmonic cost.

This configuration is intentionally a stepping stone. Current sensing is already wired and R and L are identified at runtime, so the upgrade to `TorqueControlType::foc_current` — full closed-loop current regulation — reduces to a single-line configuration change, with no hardware modifications and no impact on the communication or fault-management layers described in the following chapters.

4.2.3 The Commander Interface

SimpleFOC's **Commander** is the library's built-in communication layer [17]. It operates over `Serial/UART` interface and exposes motor parameters, PID gains, limits, and targets as a tree of single-character command identifiers. A host sends a string such as `MV2.5` to set the velocity target of motor `M` to 2.5 rad/s, the commander parser the prefix, dispatches to the registered callback, and echoes a confirmation. Readable parameters can be queried by omitting the value.

The command tree is extensible: in addition to the build-in motor, PID, low-pass filter, and scalar variable handlers, the application can register arbitrary callbacks under any send character. This makes Commander a clean abstraction boundary.

4.2.3.1 CANCommander

CANCommander is a CAN-bus wrapper implementation of the same Commander abstraction, developed by the SimpleFOC community [17]. Instead of ASCII strings over the serial stream, it uses CAN 2.0B extended (29-bit) frames carrying a structured register protocol. The motor parameters that Commander exposes over serial are mapped onto CAN register addresses, so the same motor objects can be controlled identically regardless of transport.

CAN 2.0B extended frames carry a 29-bit identifier in the arbitration field. CANCommander partitions these 29 bits into four sub-fields. Bit 28 is reserved and unused; the remaining 28 bits are allocated as follows:

reserved	Node Address	Packet Type	Register	Motor Index	**Payload**
31-28	*27–20*	*19–16*	*15–8*	*7–0*	0-8B

Table 8: CANCommander 29-bit extended identifier layout (bit 28 reserved, unused). All routing and command semantics are encoded in the arbitration ID.

Field	Bits	Shift	Range	Description
Node address	27–20	20	`0x00–0xFF`	Target node ID. 0x00–0xFE are unicast; 0xFF is broadcast (all nodes). 8 bits allow up to 254 individually addressable nodes on one bus.
Packet type	19–16	16	`0x0–0xF`	Frame role: read request, write request, read response, or sync. 4 bits; only four values are currently defined.
Register	15–8	8	`0x00–0xFF`	8-bit register address. 0x00–0xDF are built-in SimpleFOC registers; 0xE0–0xFF are reserved for application-defined custom registers.
Motor index	7–0	0	`0x00–0xFF`	Identifies one of the 256 possible addressable motor objects a node might own (here each node has only 1, so thies field is constant at 0)

The identifier is assembled in firmware as:

```
uint32_t can_id =
    ((uint32_t)node_id   << CAN_ADDRESS_SHIFT)       // bits 27–20
  | ((uint32_t)pkt_type  << CAN_PACKET_TYPE_SHIFT)   // bits 19–16
  | ((uint32_t)reg_addr  << CAN_REGISTER_SHIFT)      // bits 15–8
  | ((uint32_t)motor_idx << CAN_MOTOR_INDEX_SHIFT); // bits 7–0
```

Encoding all routing in the arbitration ID is intentional: CAN arbitration is non-destructive and priority-based, so the frame with the numerically lowest ID wins. Since the node address uses the most significant bits, nodes with lower addresses naturally possess greater bus priority This also enables efficient hardware acceptance filtering: each node masks on bits 27–20 and accepts only frames addressed to itself or to the broadcast address 0xFF.

Four packet types are defined, encoded in bits 19–16:

Table 9: CANCommander packet types encoded in bits 19–16 of the 29-bit ID. Values 0x4–0xE are reserved for future use.

Type	Name	Meaning	Payload
`0x1`	`CAN_READ_REQUEST`	Host requests the current value of a register from the node.	Empty (0 bytes).
`0x2`	`CAN_WRITE_REQUEST`	Host sends a new value to be written to a register. The node applies it. If the `echo` flag is set, the node replies with a `CAN_READ_RESPONSE` containing the updated value.	1–8 bytes.
`0x3`	`CAN_READ_RESPONSE`	Node replies to a `CAN_READ_REQUEST`, or echoes a write if echo is enabled. Register and motor index fields mirror the original request.	1–8 bytes.
`0xF`	`CAN_SYNC`	Lightweight synchronisation frame sent to broadcast address 0xFF to coordinate timing across multiple nodes simultaneously.	0–1 bytes.

A register read is a two-frame exchange: the host sends a `CAN_READ_REQUEST` (empty payload) and the node replies with a `CAN_READ_RESPONSE` (value in payload). A register write is a single frame (`CAN_WRITE_REQUEST`); no acknowledgement is sent unless the node's `echo` flag is enabled. The `CAN_SYNC` frame targets the broadcast address 0xFF and carries no register payload; it is used to synchronise control loops across multiple nodes.

Each register has an 8-bit address, a declared payload size in bytes, and two callbacks:

```
typedef bool (*RegisterReadHandler)(RegisterIO& comms, FOCMotor* motor);
typedef bool (*RegisterWriteHandler)(RegisterIO& comms, FOCMotor* motor);
```

`RegisterIO` serialises and deserialises typed values to and from the CAN payload using `<<` (write to frame) and `>>` (read from frame). The read handler pushes the current value into `comms`; the write handler extracts and applies the incoming value. Returning `false` signals an error. Because the handler signature is transport-agnostic, the same register implementation works identically over CAN, I2C, or any other transport implementing `RegisterIO`.

SimpleFOCRegisters defines the full set of built-in registers spanning address range `0x00`–`0x7F`, grouped by function. All `float` values are IEEE 754 single-precision; all multi-byte

values are little-endian. Table 10 lists the registers most relevant to this application; the complete list is in `SimpleFOCRegisters.h` on current implementation found in github [17].

Table 10: Selection of built-in SimpleFOC registers accessible via CANCommander. `float` = IEEE 754 single-precision, `uint32` = unsigned 32-bit integer, both little-endian. Custom registers begin at address `0xE0`.

Reg Address	Name	Type	Access	Description
0x00	REG_STATUS	uint8	RO	Motor status byte. Encodes enabled/ disabled state and last error.
0x01	REG_TARGET	float	R/W	Motion control target. Meaning depends on control mode: velocity (rad/ s), angle (rad), or torque (V or A).
0x03	REG_ENABLE_ALL	uint8	WO	Write 1 to enable all motors on the node simultaneously.
0x04	REG_ENABLE	uint8	R/W	Write 1 to enable this motor, 0 to disable.
0x05	REG_CONTROL_MODE	uint8	R/W	Motion control mode (velocity, angle, torque, open-loop).
0x06	REG_TORQUE_MODE	uint8	R/W	Torque control mode (voltage, dc_current, foc_current).
0x09	REG_ANGLE	float	RO	Current shaft angle in radians.
0x11	REG_VELOCITY	float	RO	Current shaft velocity in rad/s.
0x20	REG_VOLTAGE_Q	float	RO	Applied V_q in volts.
0x21	REG_VOLTAGE_D	float	RO	Applied V_d in volts.
0x30	REG_VEL_PID_P	float	R/W	Velocity PID proportional gain.
0x31	REG_VEL_PID_I	float	R/W	Velocity PID integral gain.
0x32	REG_VEL_PID_D	float	R/W	Velocity PID derivative gain.
0x50	REG_VOLTAGE_LIMIT	float	R/W	Absolute voltage cap $U_{\text{voltage_limit}}$ in volts.
0x51	REG_CURRENT_LIMIT	float	R/W	Current limit I_{lim} in amps.
0x52	REG_VELOCITY_LIMIT	float	R/W	Maximum allowable velocity in rad/s.
0x64	REG_PHASE_RESISTANCE	float	R/W	Motor phase resistance R in ohms.
0x65	REG_KV	float	R/W	Motor K_V rating in rpm/V.
0x66	REG_INDUCTANCE	float	R/W	Motor phase inductance in henries.
0x70	REG_NUM_MOTORS	uint8	RO	Number of motor objects registered on this node.
0x71	REG_SYS_TIME	uint32	RO	Node system time in microseconds.

Application-specific state with no equivalent in the built-in map is exposed through custom registers. Addresses `0xE0`–`0xFF` (32 slots, defined by `REG_CUSTOM_START`) are reserved for this purpose and never used by the library itself. A custom register is added after `commander.init()` as:

```
bool Read_Handler_0xE0_(RegisterIO& io, FOCMotor*){
    uint32_t data;
    io << data ; //mail data
    return true;
}
bool Write_Handler_0xE0_(RegisterIO& io, FOCMotor*){
    uint32_t received_data;
    io >> received_data;
    if(received_data == 12345){ //interact with the system}
}
commander.addCustomRegister(
    0xE0,     // address: must be >= REG_CUSTOM_START (0xE0)
    4,        // payload size in bytes (max 8)
    Read_Handler_0xE0_,   // Read Handler.
    Write_Handler_0xE0_   // Write handler
);
```

4.2.4 Node Firmware

Each motor drive node runs a single firmware image on the Teensy 4.0 that combines real-time motor control [Section 4.2.2], a local state machine governing the hardware lifecycle, and a CAN communication layer [Section 4.2.3.1] that bridges local behavior to the network supervisor. This section describes the design of the state machine, the safety rationale behind it, and its correspondence to the CIF plant model used for supervisor synthesis.

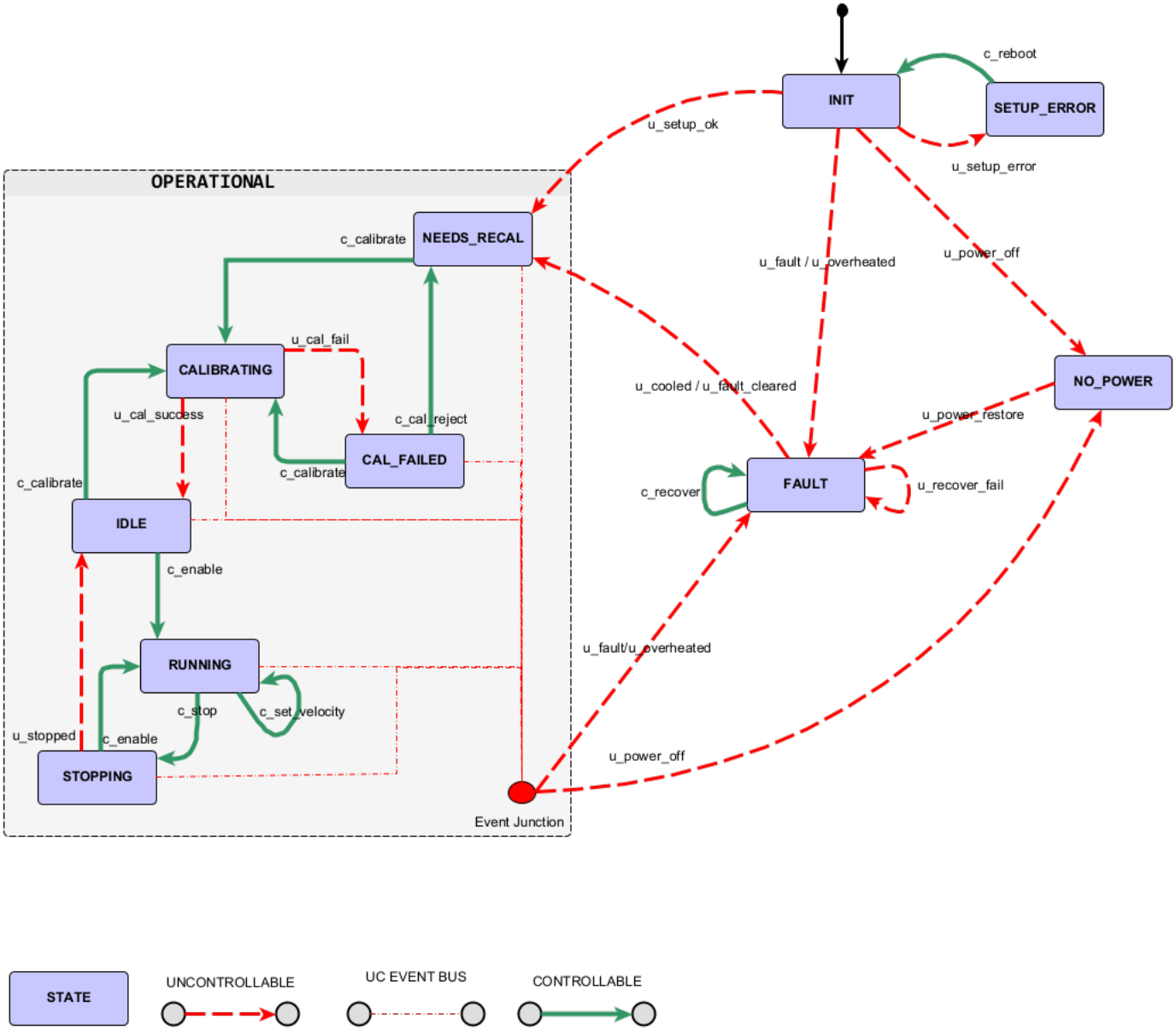


Figure 20: FSM Diagram of the Node Firmware.

4.2.4.1 Local Safety Reaction

In a networked control architecture, a fundamental design decision is where safety reactions are executed. When the DRV8302 [Section 4.2.1] gate driver signals a fault through its nFAULT pin, there are two possible approaches: the node can report the fault to the

supervisor and wait for instructions, or it can react locally and inform the supervisor after the fact.

We use the second strategy. Each node looks after its own fail-safe behavior: the firmware spots a fault, turns off the motor and gate driver, and switches into a safe state on its own. The supervisor only gets a CAN message afterwards and does not take part in the immediate reaction. Because of this, the delay from fault detection to motor shutdown is set only by the firmware loop period, not by CAN latency, supervisor processing time, or lost messages.

In IEC 61508 terms, a safe state is any condition of the equipment where the relevant hazard has been removed or brought under control. In our case, the states `FAULT` and `NO_POWER` are safe states for the motor node: all three windings are unpowered (floating) and the gate driver outputs are high-impedance, so the motor cannot produce torque. The state `STOPPING` is an intermediate safe state during a controlled stop: the motor is slowing down under closed-loop control, and the final safe state `IDLE` is reached when the speed target has dropped to zero and the rotor is at rest.

The split between local safety and network coordination follows the IEC 61508-1 idea of having more than one layer of protection. The node state machine is one layer and the synthesized supervisor is another. Even if the supervisor fails or the CAN bus goes down, the node can still move itself into a safe state, because it never waits for outside permission to shut down. The main firmware loop — which reads diagnostics, handles CAN messages, and updates the state machine — runs at about 150 kHz, separate from the 20 kHz FOC control loop, so the safety logic stays responsive even when the motor control is busy.

4.2.4.2 State Set

The firmware defines ten states. Each state corresponds to one unambiguous physical condition of the hardware. Table 11 lists them.

Table 11: Node states and their physical meaning.

State	Physical condition	Motor
`INIT`	Teensy booting, hardware not yet configured	Disabled
`SETUP_ERROR`	Hardware initialization failed	Disabled
`NEEDS_RECAL`	Hardware ready, calibration parameters missing	Disabled
`CALIBRATING`	`characteriseMotor()` measuring R and L	**Enabled**
`CAL_FAILED`	Calibration returned invalid parameters	Disabled
`IDLE`	Calibrated, motor stopped, ready to run	Disabled
`RUNNING`	Motor spinning under control	**Enabled**
`STOPPING`	Motor decelerating (exponential target decay)	**Enabled**
`FAULT`	Hardware fault (electrical or thermal)	Disabled
`NO_POWER`	Power supply lost (PWRGD low)	Disabled

Note:

The **Motor** column shows if the motor controller is enabled (PWM outputs active) or not(outputs high-impedance). The gate driver is managed by the motor object:

- calling `motor.disable()` automatically invokes `driver.disable()`
- `motor.enable()` calls `driver.enable()`, so both are always in sync.

Three design decisions in the state set are worth explaining.

1. The `NEEDS_RECAL` state is separate from `IDLE`. In `IDLE`, the motor's phase resistance and inductance have been measured and validated so that the control algorithm has correct parameters to function. The `NEEDS_RECAL` state signifies the absence of valid parameters either because the node just booted or because a power cycle or fault reset the DRV8302. The state machine enforces that the `RUNNING` state is only reachable through the path `NEEDS_RECAL -> CALIBRATING -> IDLE -> RUNNING`, preventing the supervisor from enabling a motor with stale or missing calibration data.
2. Second, `SETUP_ERROR` is a terminal state. The only exit is a hardware reboot triggered by the register with value `0xF8`, which writes to the ARM Cortex-M7 system reset register (`SCB_AIRCR`). If the driver, motor, or current sense initialization fails, no software retry can fix the underlying problem.
3. Third, `FAULT` state is unified for both electrical faults (nFAULT low, nOCTW high) and thermal faults (nFAULT low, nOCTW low). The firmware keeps track of the fault cause in an internal `FaultReason` enum, which determines the recovery strategy, but from the state machine perspective both conditions reach the same safe state with the same hardware configuration of motor and gate driver **disabled**.

The distinction only matters during the recovery procedure, not during the safety reaction. The `NO_POWER` state routes through the `FAULT` state on u_power_ restore instead of going directly to `NEEDS_RECAL` state, because a power cycle may leave the DRV8302 in a latched fault state that requires an resewt before the node can be declared healthy.

4.2.4.3 Transition Table

All legal state transitions are defined in a static array which encodes the Figure 20:

```
static const State TRANSITIONS[][2] = {
    { INIT,         NEEDS_RECAL  },   // u_setup_ok
    { INIT,         SETUP_ERROR  },   // u_setup_error
    { INIT,         FAULT        },   // u_fault / u_overheated
    { INIT,         NO_POWER     },   // u_no_power

    { NEEDS_RECAL, CALIBRATING  },   // c_calibrate
    { NEEDS_RECAL, FAULT        },   // u_fault / u_overheated
    { NEEDS_RECAL, NO_POWER     },   // u_no_power

    { CALIBRATING, IDLE         },   // u_cal_success
    { CALIBRATING, CAL_FAILED   },   // u_cal_failed
    { CALIBRATING, FAULT        },   // u_fault / u_overheated
    { CALIBRATING, NO_POWER     },   // u_no_power

    { CAL_FAILED,  CALIBRATING  },   // c_calibrate (retry)
    { CAL_FAILED,  NEEDS_RECAL  },   // c_cal_reject
    { CAL_FAILED,  FAULT        },   // u_fault / u_overheated
    { CAL_FAILED,  NO_POWER     },   // u_no_power

    { IDLE,         RUNNING      },   // c_enable
    { IDLE,         CALIBRATING  },   // c_calibrate (recalibrate)
    { IDLE,         FAULT        },   // u_fault / u_overheated
    { IDLE,         NO_POWER     },   // u_no_power

    { RUNNING,     STOPPING     },   // c_stop
    { RUNNING,     FAULT        },   // u_fault / u_overheated
    { RUNNING,     NO_POWER     },   // u_no_power

    { STOPPING,    IDLE         },   // u_stopped
    { STOPPING,    RUNNING      },   // c_enable (abort stop)
    { STOPPING,    FAULT        },   // u_fault / u_overheated
    { STOPPING,    NO_POWER     },   // u_no_power

    { FAULT,       NEEDS_RECAL  },   // u_cooled / u_fault_cleared
    { FAULT,       FAULT        },   // u_recover_failed (self-loop)
    { FAULT,       NO_POWER     },   // u_no_power

    { NO_POWER,    FAULT        },   // u_power_restore
};
```

Listing 1: Static transition table. Each row is one legal `{from, to}` pair. The comments indicate the corresponding CIF event for traceability.

The function `transition_to()` is the only code path that writes to the `state` variable. It iterates the table, and if the requested `{from, to}` pair is not found, it rejects the transition and logs the attempt. No other function modifies state directly.

This is a whitelist pattern: the table defines what is permitted, and everything else is implicitly forbidden. IEC 61508-3 Annex A lists defensive programming among the techniques recommended for safety-related software development [9]. A direct application of this technique is the transition table, it prevents the system from entering any state combination that was not explicitly designed and analyzed.

4.2.4.4 Fault Detection

The firmware monitors the DRV8302 diagnostic pins (PWRGD, nFAULT, nOCTW) by polling in the main `loop()` function with software debounce timer, bo hardware interrupts are used. PWRGD uses an asymmetric debounce:

- 50 ms for falling edges (power loss)

- 200 ms for rising edges (power restore).

Power loss must be detected quickly because the gate driver becomes unreliable below its undervoltage lockout threshold. Power restoration requires a longer confirmation to avoid reacting to brief voltage transients during the PSU's soft-start.

When nFAULT is confirmed low after 50ms, the firmware reads nOCTW to classify the fault as electrical (nOCTW high) or thermal (nOCTW low), records the reason in the `FaultReason` enum, and proceeds to disable the motor and the gate driver, then call `transition_to(FAULT)`.

Note: The DRV8302 boards employed in this system were noted to be quite susceptible to noise on their diagnostic lines. Consequently, somewhat longer debounce delays were chosen to guarantee dependable fault detection and to avoid erroneous triggers due to electrical interference.

In every fault path, the firmware calls `motor.disable()` *before* calling `transition_to()` since the ordering matters. The hardware reaches its safe state before the state machine updates, before any CAN notification is sent, and before any other logic executes. If `transition_to()` were to fail for any reason, the motor would already be disabled.

4.2.4.5 CAN Register Interface

The supervisor interacts with each node through the CANCommander register protocol Section 4.2.3.1. Table 12 lists the registers and their mapping to CIF events.

Table 12: CAN register map. Each writable register maps to one controllable event in the CIF plant model.

Register	Name	CIF event	Action
`0xF0`	State	—	Returns current state (read-only)
`0xF1`	Calibrate	`c_calibrate`	`transition_to(CALIBRATING)`
`0xF2`	Enable	`c_enable`	`transition_to(RUNNING)`
`0xF3`	Stop	`c_stop`	`transition_to(STOPPING)`
`0xF4`	Recover	`c_recover`	`EN_GATE reset sequence`
`0xF5`	Cal. reject	`c_cal_reject`	`transition_to(NEEDS_RECAL)`
`0xF6`	Velocity	`c_set_velocity`	Writes `target_vel` if RUNNING
`0xF8`	Reboot	`c_reboot`	ARM system reset

Note: The `c_cal_reject` event (register 0xF5) is not used in this application. Failed calibrations are always retried automatically by the supervisor up to maximum number of attempts. The event is disabled during synthesis via needs false to remove it from the supervisor's decision space.

Every register write handler calls `transition_to()`. If the node is in a state where the requested transition is not in the table — for example, the supervisor sends a calibrate command while the node is `RUNNING` — the transition is rejected, the handler returns false, and the node's state does not change. The node firmware thus provides a safety net independent of the supervisor: an illegal command simply has no effect.

The firmware uses a separate CAN channel with packet type `0x4` for uncontrollable events. When internally a transition is triggered (fault detected, calibration finished, motor stopped,

power lost), the `transition_to()` function pushes a CAN frame whose 29-bit identifier encodes the node address, event type, and a sequence number with zero payload. The supervisor by receiving this frame type and updates its view of the node's state, essentialy an uncontrollable edge in the state machine is executed. The sequence number field allows for the identification of missing events or frames, and it also paves the way for future enhancements where error handling at the protocol level can be implemented and refined

```
enum CANEvent : uint8_t {
    EVT_NONE            = 0x00, // never used
    EVT_SETUP_OK        = 0x01, // u_setup_ok      // Setup completed
    EVT_SETUP_ERROR     = 0x02, // u_setup_error   // Setup not completed
    EVT_CAL_SUCCESS     = 0x03, // u_cal_success   // Calibration successfull
    EVT_CAL_FAILED      = 0x04, // u_cal_fail     // Calibration failed
    EVT_FAULT           = 0x05, // u_fault         // Generic fault entry
    EVT_NO_POWER        = 0x06, // u_power_off     // Power was cut
    EVT_POWER_RESTORE   = 0x07, // u_power_restore // Power came back
    EVT_STOPPED         = 0x08, // u_stopped       // The motor actually stopped
    EVT_OVERHEATED      = 0x09, // u_overheated    // Specific thermal event
    EVT_COOLED          = 0x0A, // u_cooled        // Thermal cleared
    EVT_FAULT_CLEARED   = 0x0B, // u_fault_cleared // Electrical fault cleared
    EVT_RECOVER_FAILED  = 0x0C, // u_recover_failed// Recovery failed
};
```

The recovery mechanism in the `FAULT` state attempts the EN_GATE (driver.disable() and then driver.enable()) reset and pushes either a success event (`u_fault_cleared` or `u_cooled`, transitioning to `NEEDS_RECAL`) or a failure event (`u_recover_failed`, staying in `FAULT`).

4.2.4.6 Runtime Enforcement

The firmware includes a runtime consistency check executed every loop iteration, after the CANCommander processes incoming messages:

```
        if (motor.enabled && state != RUNNING &&
        state != CALIBRATING && state != STOPPING) {  // incase motor is enabled
            motor.disable();                          //via the default cmds of CAN IF
        }
        if (state != RUNNING && state != STOPPING && motor.target != 0.0f) {
            motor.target = target_vel = 0.0f; // target should be 0
        }
```

Given that CANCommander offers preset register commands, as shown in Table 10, capable of communicating directly with our node, we must separate our design to ensure that only the specified interface is utilized with our implementation.

When the motor is activated during a state machine's prohibition, or if a nonzero velocity target is present in a non-operational state, the firmware resolves the inconsistency in one loop cycle. This also includes edge scenarios, like a damaged CAN message that avoids the state machine and activates `motor.enable()` through SimpleFOC's internal registers

4.3 Application Case: Heat Recovery Ventilation System

With the motor drive node functionalities established, this section outlines the application: a balanced ventilation system with dual motors (Heat Recovery Ventilator)

4.3.1 System Function

An HRV provides continuous fresh air while recovering thermal energy [14]. Two fans operate through a passive heat exchanger core (Figure 21):

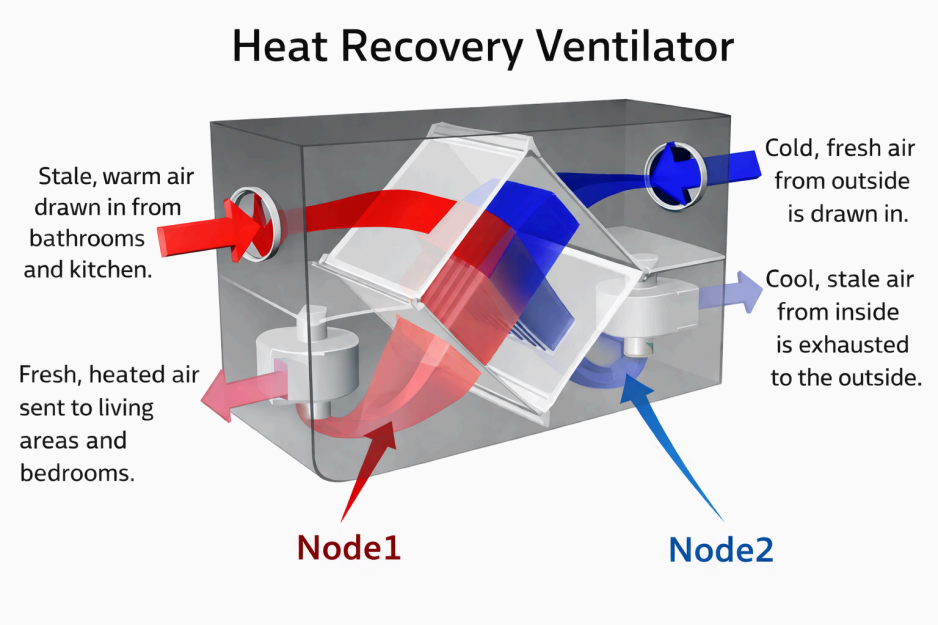


Figure 21: Heat Recovery Ventilation System.

Supply Fan (NODE1): Draws outdoor air through the heat recovery core, where it is pre-conditioned by the exhaust airstream, then delivers tempered fresh air to living spaces.

Exhaust Fan (NODE2): Draws stale air from bathrooms and kitchens, transfers its thermal energy to incoming fresh air via the core, then expels it outdoors.

The heat recovery core is passive (plate stack or desiccant wheel). All system dynamics are determined by fan speeds.

Motor Technology. Modern residential and commercial HRV systems use brushless DC motors for both supply and exhaust fans. The HVAC industry refers to these as ECM (Electronically Commutated Motor) technology — functionally identical to BLDC motors with integrated drive electronics. ECM/BLDC became industry standard in the 2010s due to energy efficiency regulations, precise variable-speed control needed for airflow balancing, and continuous operation requirements. The primary difference is packaging: commercial ECM products integrate the gate driver and controller into the motor housing, while this platform uses discrete components to expose the control architecture for formal modeling.

4.3.2 Safety Requirement: Airflow Balance

The fundamental constraint is airflow balance [15]:

$$Q_{\text{supply}} \approx Q_{\text{exhaust}} \quad (\pm 10\% \text{ tolerance}) \tag{11}$$

where Q is volumetric airflow (CFM).

Consequences of Imbalance:

- **Depressurization** [$Q_{\text{exhaust}} > Q_{\text{supply}}$]: In combustion appliances (furnaces, water heaters) reverse draft can pull into occupied spaces flue gases containing carbon monoxide instead of exhausting them through the chimney [14]. This is a serious life-safety hazard.
- **Pressurization** ($Q_{\text{exhaust}} < Q_{\text{supply}}$): Pushes unfiltered outdoor air through gaps in the building envelope, disregarding filtration and heat recovery. Reduces indoor air quality and consumes energy unnecessarily
- **Mechanical stress:** Sustained pressure differential causes door slamming, window whistling, and envelope damage.

ASHRAE 62.2 and building codes require balanced systems to maintain ±10% tolerance during operation [15]. If either fan fails, **both must stop immediately**.

4.3.3 Coordination Challenges

- **Synchronous Operation:** Both fans must start together, run at matched target velocities and finally stop together. A fault in one fan requires immediate stopping of the other.
- **Sequential Calibration:** Calibration routine injects voltage steps to measure motor phase resistance and inductance L. This creates:
  - High speed transient current draws.
  - Voltage sag on shared 12 V rail during current pulses.
  - Mechanical vibration coupling through shared mounting frame

Note: Power supply ripple and measurement crosstalk make simultaneous calibration unreliable. Calibrations must be sequential: `NODE1` completes, then `NODE2`.

- **Fault Recovery Coordination:** Example scenario goes as follows , `NODE1` thermal fault, `NODE2` healthy:
  1. Stop `NODE2` immediately (prevent imbalance)
  2. Attempt `NODE1` recovery (`EN_GATE` reset or passive cooling)
  3. Wait for `NODE1` to reach `NEEDS_RECAL`
  4. Recalibrate `NODE1` (`NODE2` remains stopped)
  5. Restart both together

This type of coordination logic developed with hand-written code generally is error-prone. Our model based approach coupled with Synthesis guarantees it by construction.

4. Safety Counter Limits: Repeated failures (e.g., thermal faults from blocked inlet) must eventually escalate to an general system `EMERGENCY` state requiring operator intervention. The supervisor tracks fault, power cycle, recovery , and calibration across both nodes.

4.3.4 Why Balanced Ventilation Fits Supervisor Synthesis

- **Clear safety requirements:** Airflow balance is a hard constraint, which can be directly expressed in CIF requirements
- **Non-trivial coordination:** Synchronous operation, sequential calibration, coordinated fault recovery
- **Real-world relevance:** ECM/BLDC-based HRV systems are deployed in millions of buildings, having formally verified coordination has practical value.
- **Scalability:** CIF models/requirements can be used to extend to N>2 nodes (multi-zone buildings, industrial systems)

The Section 4.4.1 develops the CIF formal models that capture node behavior (Section 4.2.4) and HVAC safety requirements. The synthesized supervisor enforces balanced ventilation while maximizing operational flexibility within the safe .

4.4 Modeling

4.4.1 Node Plant Model

The formalization of the firmware FSM seen in Section 4.2.4 is the CIF plant `DRV8302_motor_node` extended finite automaton. Beyond mirroring the 10 states and their transitions, the model adds discrete variables for supervisor bookkeeping and diagnostic counters that capture runtime history invisible to the firmware but essential for system decisions. From the supervisors perspective each controllable event represents a CAN register write, and each uncontrollable event represents a can read.

```
///NODE.cif
plant def DRV8302_motor_node(uncontrollable u_technician_reset; alg int [0..300] tar_vel):
     disc int[0..MAX_RANGE] fault_cnt = 0,          // u_fault, u_overheated (total faults)
                            power_cc_cnt = 0,       // u_no_power        (total power cycles)
                            recovery_att_cnt = 0,   // u_recover_failed  (consecutive recoveries)
                            calibration_att_cnt = 0, // u_cal_failed     (consecutive calibrations)
                            reboot_cnt = 0;         // c_reboot          (consecutive reboots)
// ... locations and edges omitted, see Appendix A
end
```

4.4.1.1 Marked Location Strategy

The formal definition of a marked state (Section 3.2) says it is an acceptable end state — a location where the system sees its current task as finished.The non-blocking property means that from every reachable state, there exists a finite sequence of events that leads to a marked location; in other words, the system can always, in principle, reach a completion state.

In this plant model, marked locations are not “good” and unmarked ones are not “bad”. The difference is simply between states where the node has finished what it was doing and states where it is still working. The model marks seven of the ten locations:`INIT`, `SETUP_ERROR`, `NEEDS_RECAL`, `CAL_FAILED`, `IDLE`, `FAULT`, and `NO_POWER`. All seven are fault-safe states (Section 4.2.4.1): the motor is disabled, the gate driver is in high-impedance, meaning the node can safely stay in them. Each one represents a completed outcome, such as successful initialization, a detected fault, a lost power supply, or a failed calibration, rather than an ongoing operation.

The three unmarked locations — `CALIBRATING`, `RUNNING`, and `STOPPING` are different. Here the motor is active and current is flowing, so the operation must move forward to completion. These are tasks in progress, not finished tasks.

Having the fault-safe states marked is essential because hardware recovery depends on physical conditions that the supervisor cannot control — for example, the die cooling below 130 °C, or the return of power, or a successful `EN_GATE` reset. If `FAULT` or `NO_POWER` states were unmarked, the non-blocking check would require a guaranteed path out of them, which is not possible when recovery is uncertain. The model would then be flagged as blocking. By marking these states, the model correctly says that a node in a fault-safe state with the motor disabled has finished its immediate task, having reached a safe configuration. What

might happen next, either it be recovery, manual intervention, or remaining in standby, is handled by the coordinator and its higher-level requirements.

4.4.1.2 Diagnostic Counters: Cumulative vs Consecutive

The node maintains five diagnostic counters serving different purposes.

- Cumulative counters that track total occurrences, serving as long-term stability indicators.
- Consecutive counters track retry attempts, to make room for autonomy and prevent infinite "retry" loops.

This distinction determines reset policies and drives different classes of requirements.

Cumulative Counters (Long-Term):

The counters `fault_cnt` and `power_cc_cnt` accumulate evidence of system degradation. These are never reset by successful recovery or calibration—only by technician intervention confirming hardware service/check up. A node having gone through and recovered 5 thermal faults still carries `fault_cnt=5`, preserving evidence that might indicate the hardware is degrading. Calibration success indicates only that motor parameters where measured and make sense at this moment, not that underlying causes (loose connections, power supply issues) have been addressed. Only physical inspection warrants resetting cumulative fault history.

Consecutive Counters (Short-Term):

The `recovery_att_cnt`, `calibration_att_cnt`, and `reboot_cnt` counters track consecutive failures of specific operations and reset on success. This enables the following bounded retry: attempt recovery `MAX_RECOVERY` times per fault, then escalate—while preserving the record that faults occur frequently. These counters prevent infinite retry loops and trigger emergency escalation when thresholds are reached.

Table 13: Cumulative and Consecutive counters

Counter	Increments on	Resets on	Purpose
`fault_cnt`	`u_fault`, `u_overheated`	`u_technician_reset`	Track faults
`power_cc_cnt`	`u_no_power`	`u_technician_reset`	Power stability
`recovery_att_cnt`	`u_recover_failed`	`u_cooled`, `u_fault_cleared`, `u_technician_reset`	Limit fault recovery
`calibration_att_cnt`	`u_cal_failed`	`u_cal_success`, `u_technician_reset`	Limit calibration
`reboot_cnt`	`c_reboot`	`u_setup_ok`, `u_technician_reset`	Limit reboots

4.4.1.3 Emergency Intervention Mechanism

When retry limits are exhausted, intervention becomes necessary. The `u_technician_reset` event models this as a shared uncontrollable input provided to all node instances. This creates a network broadcast, one physical reset button affects all nodes simultaneously.

The design is just **state-dependent counter clearing**. Nodes that are in in just error states (`SETUP_ERROR`, `NEEDS_RECAL`, `CAL_FAILED`, `FAULT`) clear all counters on reset, representing hardware service.

This is the template of the node implementation in CIF:

```
// Input group definition
group def CAN_IF():
    // Event flags from CAN node
    input bool Evt_setup_ok      , Evt_setup_error,  Evt_cal_success   , Evt_cal_failed;
    input bool Evt_stopped       , Evt_fault      ,  Evt_no_power      , Evt_power_restored;
    input bool Evt_overheated    , Evt_cooled     ,  Evt_fault_cleared , Evt_recover_failed;
end
group  def  NODE(alg int [1..254] node_id ; uncontrollable u_technician_reset ; alg int [0..300] tar_vel):
      _: DRV8302_motor_node(u_technician_reset, tar_vel);
    CAN: CAN_IF();
    //// Plant invariants linking events to input flags
    // plant invariant _.u_setup_ok         needs CAN.Evt_setup_ok;
    // plant invariant _.u_setup_error      needs CAN.Evt_setup_error;
    // ...
    // ...
    // plant invariant _.u_recover_failed   needs CAN.Evt_recover_failed;
end
```

Listing 3: NODE CIF definition

4.4.2 Coordinator Model

The coordinator is a high level extended finite automaton that orchestrates system-wide operation. It keeps track of the collective health of the nodes and manages transitions through well-defined operational phases, like `startup`, `calibration`, `shutdown` etc. The coordinator does not directly control the individual nodes, it monitors node states and issues phase-level transitions that requirements translate into safe, coordinated actions.

4.4.2.1 Operating Phases:

The coordinator implements eight locations representing system-wide operational modes:

- **`IDLE`:** System idle.
- **`BOOTING`:** Nodes transitioning from initialization (`INIT`) to operational readiness (`NEEDS_RECAL`). Waits for all nodes to complete hardware initializations before proceeding.
- **`CALIBRATING`:** This is the parameter characterizations phase. Nodes measure motor resistance and inductance.
- **`RUNNING`:** Normal coordinated operation where all nodes calibrated and enabled.
- **`STOPPING`:** Coordinated shutdown triggered by the operator.
- **`RECOVERING`:** Fault recovery phase. One or more nodes experienced faults and require recovery attempts (`EN_GATE` reset, passive cooling). System remains stopped during recovery, then returns to `BOOTING` for recalibration.
- **`EMERGENCY`:** Safety thresholds exceeded (retry limits, counter limits). All operations blocked until technician intervention via `u_technician_reset`.
- **`NO_POWER`:** Power supply failure detected. System waits for power restoration, then transitions to RECOVERING for orderly restart.

4.4.2.2 Autonomous Transitions

Phase transitions occur via two mechanisms:

Operator Commands: The operator commands are `u_start`, `u_stop`, `u_technician_reset` all of them represent external intent, which would come from the Human Operator Layer as depicted and explained in Figure 5.

Health Monitoring: The commands `c_nodes_ready`, `c_all_calibrated`, `c_node_unhealthy`, `c_all_healthy`, `c_limits_exceeded`, `c_all_stopped`, `c_all_power_lost`, `c_power_restored` are controllable events that fire when specific system-wide conditions are met. Those very conditions are a logical composition of our requirements, for example, `c_nodes_ready` fires when all NODES are in NEEDS_RECAL. Synthesis ensures these events fire autonomously when conditions hold, guaranteeing responsive phase transitions without explicit programming.

4.4.3 System Requirements

In this chapter the formal requirements that govern the distributed BLDC motor control network are presented. Requirements that translate application-specific constraints and general safety policies into precise conditions that guide automated supervisor synthesis, each requirement specifies when controllable events may execute, ensuring the synthesized supervisor operates within safe boundaries.

The network is defined as such :

```
import "NODE.cif";
import "Coordinator.cif";
// Network instantiation: coordinator + 2 nodes
group Net:
    coord: System_Coordinator();
    NODE1: NODE(1, coord._.u_technician_reset, coord._.velocity);
    NODE2: NODE(2, coord._.u_technician_reset, coord._.velocity);
end
```

The system coordinator provides two shared signals to both nodes:

- `u_technician_reset` event for emergency intervention
- `int[0..300] velocity;` value for synchronized speed commands

Note: This creates network-wide broadcasts where a single physical action effects both mirror nodes at the same time.

4.4.3.1 Safety Threshold Constants

Five configurable constants define the boundaries of automatic recovery:

```
const int MAX_FAULTS   = 5;  // Cumulative fault events
const int MAX_POWER    = 3;  // Power cycle events
const int MAX_CAL      = 3;  // Consecutive calibration failures
const int MAX_RECOVERY = 3;  // Consecutive recovery attempts
const int MAX_REBOOT   = 3;  // Consecutive system reboots
```

These limits stop endless retry cycles on persistent hardware malfunctions while allowing temporary problems to resolve on their own. When a threshold is met, the system escalates to an EMERGENCY state needing technician involvement.

4.4.3.2 Requirement Organization

We ended up with 46 requirements, which we grouped into eight categories. The grouping is mostly driven by what each requirement actually protects against, and it follows the system's lifecycle — startup, operation, fault handling — though some requirements focus on more than one particular phase. The first two categories focus on the coordinator's responsibility as the central state machine. *Coordinator Transitions* (R1-R8) define when the coordinator is allowed to move between phases: `BOOTING` to `CALIBRATING`, `CALIBRATING` to `RUNNING`, and so on. *Phase Gating* (R9-R18) works in the other direction — given the phase the coordinator is currently in, it dictates which node-level operations can and cannot happen.The next pain of categories exists because of a hardware constraint that shaped much of the desing: the shared 12V power supply. *Sequential Calibration* (R19-R20) stops both nodes from calibrating at the same time, the reasons have already been discussed in Section 4.3.3. *Sequential Recovery* (R21) applies the same idea to fault recovery, with

an small difference, the supply fan (NODE1) always goes first,since maintaining positive building pressure matters more than getting both fans back at once. *Counter-Based Safety* (R22-R27) adds hard limits on how many times the system will retry calibration, recovery, or reboot before giving up. The logic is simple: if three or four have failed, the problem is almost certainly not going away on its own, and continuing to retry just wastes time or risks further damage (see Section 4.4.1.2). *HVAC Synchronous Operation* (R28-R29) enforces a straightforward rule: both fans run, or neither does. Operating only the supply fan (NODE1) or only the exhaust fan (NODE2) would create imbalance of the building's pressure (see Section 4.3.2). *Preventative Safety* (R30-R33) tries to catch problems a step earlier than the counter-based rules do. Instead of waiting for retry counters to max out, these requirements look at degradation patterns, signified by the cumulative counters (see Section 4.4.1.2).And last, *Determinism and Priority* (R34-R46), handles the situations the earlier rules leave ambiguous. Situations where multiple controllable events can be enabled, what would be the correct order of execution with respect the specific scenario? This category make our supervisor have confluence , previously discussed in Section 3.5.2.

The table below lists every requirement with its formal condition and the reasoning behind it.

4.4.3.3 Complete Requirement Specification

ID	Requirement	Rationale
Category 1: Coordinator Autonomous Transitions (8 requirements)		
R1	Both nodes in ready-for-calibration states (`NEEDS_RECAL`, `IDLE`, or `CAL_FAILED`)	Coordinator transitions `BOOTING` to `CALIBRATING` when both nodes completed hardware initialization. Accepts `IDLE` and `CAL_FAILED` to handle nodes that were previously calibrated or attempted calibration
R2	Both nodes in `IDLE` state and no counter at maximum threshold	Coordinator transitions `CALIBRATING` to `RUNNING` when both motors successfully measured parameters and system remains below safety limits
R3	Coordinator in `STOPPING` phase and both nodes in `IDLE` or `NEEDS_RECAL` states	Coordinator transitions `STOPPING` to `IDLE` only after both fans confirmed complete spindown, preventing premature state change during shutdown
R4	Any node in `FAULT` or `SETUP_ERROR` state, or asymmetric power loss (one node `NO_POWER`, other not)	Coordinator detects unhealthy condition and transitions to `RECOVERING` phase. Distinguishes single-node power loss from total building power failure
R5	Coordinator in `RECOVERING` phase and both nodes in `IDLE` or `NEEDS_RECAL` states	Coordinator transitions `RECOVERING` to `BOOTING` when both nodes successfully recovered. Accepts both states to handle asymmetric recovery scenarios
R6	Any safety counter on either node reached maximum threshold	Coordinator transitions to `EMERGENCY` when automatic recovery exhausted, blocking all operations until technician physically intervenes
R7	Both nodes simultaneously in `NO_POWER` state	Coordinator transitions to `NO_POWER` state to track total building power failure separately from single-node faults
R8	Coordinator in `NO_POWER` phase and both nodes exited `NO_POWER` state	Coordinator transitions `NO_POWER` to `RECOVERING` when building power restored, initiating orderly restart sequence
Category 2: Phase Gating (10 requirements)		
R9	NODE1 calibration requires coordinator in `CALIBRATING` phase	Prevents parameter measurement during initialization, operation, or shutdown when electrical conditions would corrupt measurements
R10	NODE2 calibration requires coordinator in `CALIBRATING` phase	Prevents parameter measurement during initialization, operation, or shutdown when electrical conditions would corrupt measurements
R11	NODE1 motor enable requires coordinator in `RUNNING` phase	Supply fan starts only after system-wide initialization and calibration completed, ensuring coordinated startup
R12	NODE2 motor enable requires coordinator in `RUNNING` phase	Exhaust fan starts only after system-wide initialization and calibration completed, ensuring coordinated startup
R13	NODE1 motor stop requires coordinator in `STOPPING` or `RECOVERING` phase	Allows controlled shutdown during coordinated stop sequence or fault recovery, prevents stopping during normal operation

ID	Requirement	Rationale
R14	NODE2 motor stop requires coordinator in `STOPPING` or `RECOVERING` phase	Allows controlled shutdown during coordinated stop sequence or fault recovery, prevents stopping during normal operation
R15	NODE1 recovery commands require coordinator in `RECOVERING` phase	Fault recovery attempts occur only during designated recovery period, preventing recovery commands during normal operation
R16	NODE2 recovery commands require coordinator in `RECOVERING` phase	Fault recovery attempts occur only during designated recovery period, preventing recovery commands during normal operation
R17	NODE1 velocity updates require coordinator in `RUNNING` phase	Speed commands accepted only during active operation, preventing velocity changes during initialization or shutdown
R18	NODE2 velocity updates require coordinator in `RUNNING` phase	Speed commands accepted only during active operation, preventing velocity changes during initialization or shutdown
Category 3: Sequential Calibration (2 requirements)		
R19	NODE1 calibration requires NODE2 not in `CALIBRATING` state	Sequential calibration prevents simultaneous high-current draws (1.5-2.5A per node) that would exceed shared 12V power supply capacity and cause voltage sag
R20	NODE2 calibration requires NODE1 not in `CALIBRATING` state	Sequential calibration prevents simultaneous high-current draws (1.5-2.5A per node) that would exceed shared 12V power supply capacity and cause voltage sag
Category 4: Sequential Recovery (1 requirement)		
R21	NODE2 recovery requires NODE1 not in `FAULT` state	Supply fan (NODE1) recovers first to stagger inrush current on shared power supply and prioritize positive building pressure
Category 5: Counter-Based Safety (6 requirements)		
R22	NODE1 calibration requires `calibration_att_cnt` less than `MAX_CAL`	After threshold consecutive failures, persistent calibration problems indicate damaged sensors or motor windings requiring technician diagnosis
R23	NODE2 calibration requires `calibration_att_cnt` less than `MAX_CAL`	After threshold consecutive failures, persistent calibration problems indicate damaged sensors or motor windings requiring technician diagnosis
R24	NODE1 recovery requires `recovery_att_cnt` less than `MAX_RECOVERY`	After threshold consecutive failures, stubborn faults indicate persistent hardware problems requiring physical inspection and repair
R25	NODE2 recovery requires `recovery_att_cnt` less than `MAX_RECOVERY`	After threshold consecutive failures, stubborn faults indicate persistent hardware problems requiring physical inspection and repair
R26	NODE1 reboot requires `reboot_cnt` less than `MAX_REBOOT`	After threshold consecutive failures, persistent initialization problems indicate firmware corruption or hardware defects requiring intervention
R27	NODE2 reboot requires `reboot_cnt` less than `MAX_REBOOT`	After threshold consecutive failures, persistent initialization problems indicate firmware corruption or hardware defects requiring intervention
Category 6: HVAC Synchronous Operation (2 requirements)		
R28	NODE1 enable requires NODE2 not in `FAULT`, `NO_POWER`, `SETUP_ERROR`, or `CAL_FAILED` states	Both-or-nothing policy prevents single-fan operation causing building pressure imbalance and potential backdrafting hazard
R29	NODE2 enable requires NODE1 not in `FAULT`, `NO_POWER`, `SETUP_ERROR`, or `CAL_FAILED` states	Both-or-nothing policy prevents single-fan operation causing building pressure imbalance and potential backdrafting hazard
Category 7: Preventive Safety (4 requirements)		
R30	NODE1 enable requires `fault_cnt` less than `MAX_FAULTS` on both nodes	Block startup when either node shows degradation pattern indicating system instability requiring technician inspection
R31	NODE2 enable requires `fault_cnt` less than `MAX_FAULTS` on both nodes	Block startup when either node shows degradation pattern indicating system instability requiring technician inspection
R32	NODE1 calibration requires `power_cc_cnt` less than `MAX_POWER`	Power supply instability detected; avoid high-current calibration that might trigger another power cycle and damage hardware
R33	NODE2 calibration requires `power_cc_cnt` less than `MAX_POWER`	Power supply instability detected; avoid high-current calibration that might trigger another power cycle and damage hardware
Category 8: Determinism and Priority Enforcement (15 requirements)		
R34	NODE1 reboot requires coordinator in `RECOVERING` phase	System reboots coordinated through `RECOVERING` phase ensuring orderly restart sequence, preventing autonomous reboot during active operation

ID	Requirement	Rationale
R35	NODE2 reboot requires coordinator in `RECOVERING` phase	System reboots coordinated through `RECOVERING` phase ensuring orderly restart sequence, preventing autonomous reboot during active operation
R36	NODE1 calibration requires NODE2 healthy (not `FAULT`, `SETUP_ERROR`, `NO_POWER`) or coordinator in `RECOVERING` phase	Calibration proceeds when other node healthy or during coordinated recovery, preventing operation during asymmetric fault conditions
R37	NODE2 calibration requires NODE1 healthy (not `FAULT`, `SETUP_ERROR`, `NO_POWER`) or coordinator in `RECOVERING` phase	Calibration proceeds when other node healthy or during coordinated recovery, preventing operation during asymmetric fault conditions
R38	NODE1 velocity updates require NODE2 healthy (not `FAULT`, `SETUP_ERROR`, `NO_POWER`)	Speed changes permitted only when both nodes operational, maintaining synchronized airflow critical for balanced ventilation
R39	NODE2 velocity updates require NODE1 healthy (not `FAULT`, `SETUP_ERROR`, `NO_POWER`)	Speed changes permitted only when both nodes operational, maintaining synchronized airflow critical for balanced ventilation
R40	NODE1 reboot requires no counter at limit, NODE2 not `RUNNING`, and NODE2 in `IDLE`	Prevents reboot during emergency conditions or asymmetric states, ensuring stable system configuration before disrupting NODE1
R41	NODE1 calibration requires no counter at limit on either node	System near failure thresholds should not attempt high-current operations; `EMERGENCY` escalation takes priority
R42	NODE1 recovery requires no counter at limit on either node	System with exhausted recovery attempts should escalate to `EMERGENCY` rather than continue automatic recovery loops
R43	NODE2 reboot requires no counter at limit, NODE1 not `RUNNING`, and NODE1 in `IDLE`	Prevents reboot during emergency conditions or asymmetric states, ensuring stable system configuration before disrupting NODE2
R44	NODE2 calibration requires no counter at limit on either node	System near failure thresholds should not attempt high-current operations; `EMERGENCY` escalation takes priority
R45	NODE2 recovery requires no counter at limit on either node	System with exhausted recovery attempts should escalate to `EMERGENCY` rather than continue automatic recovery loops
R46	NODE2 calibration requires NODE1 not in `NEEDS_RECAL` or `CAL_FAILED` states	Strict calibration priority: supply fan (NODE1) calibrates before exhaust fan (NODE2), providing deterministic initialization sequence

Based on the information covered in the Section 3.3.2 below we show how some of the requirements where translated into cif:

```
// State-based requirement (example: R11)
requirement Net.NODE1._.c_enable needs Net.coord._.RUNNING
// Counter-based requirement (example: R22)
requirement Net.NODE1._.c_calibrate needs (Net.NODE1._.calibration_att_cnt < MAX_CAL);
// Combined state and counter requirement (example: R2)
requirement Net.coord._.c_all_calibrated needs
    (Net.NODE1._.IDLE and Net.NODE2._.IDLE) and
    (not LIMIT_EXHAUSTED);
```

The `LIMIT_EXHAUSTED` algebraic variable simplifies complex counter checks across both nodes:

```
alg bool LIMIT_EXHAUSTED = (
    Net.NODE1._.fault_cnt >= MAX_FAULTS or
    Net.NODE1._.power_cc_cnt >= MAX_POWER or
    Net.NODE1._.recovery_att_cnt >= MAX_RECOVERY or
    Net.NODE1._.calibration_att_cnt >= MAX_CAL or
    Net.NODE1._.reboot_cnt >= MAX_REBOOT or
    Net.NODE2._.fault_cnt >= MAX_FAULTS or
    Net.NODE2._.power_cc_cnt >= MAX_POWER or
    Net.NODE2._.recovery_att_cnt >= MAX_RECOVERY or
```

```
    Net.NODE2._.calibration_att_cnt >= MAX_CAL or
    Net.NODE2._.reboot_cnt >= MAX_REBOOT
);
```

4.4.4 Synthesis and Validation

Having completed the modeling iteration, we entered the second phase of the synthesis-based workflow as show in Section 3.1. The second phase Synthesis and validation here consisted of four stages: abstract synthesis, interactive simulation, hardware binding integration, and deployment verification.

4.4.4.1 Abstract Synthesis and Simulation

Initial synthesis took as input the plant and requirement models without hardware input variable bindings, since the interactive simulator (gui input) does not support input variables during interactive execution. All `plant invariant` declarations were temporarily commented out in `NODE.cif` and `Coordinator.cif`.

Synthesis completed in approximately 2.5 seconds on a standard office laptop, producing a supervisor enforcing all 46 requirements while maintaining controllability, non-blocking, and maximal permissiveness. Interactive simulation validated critical scenarios:

- **Normal operation:** System startup through `IDLE` to `BOOTING`, from `BOOTING` to `CALIBRATING` and from `CALIBRATING` to `RUNNING`, coordinated motor enable/disable, synchronized velocity operation and total system shutdown.
- **Fault handling:** Single-node electrical and thermal faults, asymmetric and total power loss followed with the proper recovery sequences.
- **Counter limits:** Calibration and recovery retry exhaustion triggering `EMERGENCY` escalation, technician reset clearing counters
- **Hardware constraints:** Sequential calibration and sequential recovery
- **HVAC policy:** Cross-node health dependencies preventing single-fan operation

All scenarios executed correctly with no deadlocks, livelocks, or requirement violations.

4.4.4.2 Hardware Binding and Re-synthesis

Plant models were updated by just un-commenting `plant invariant` declarations that in the generated code would link uncontrollable events to CAN input flags (see Listing 3). This creates the interface between discrete event abstraction and physical CAN communication. Re-synthesis succeeded, producing production-ready specification with integrated hardware guards.

4.4.4.3 Controller Properties Verification

Before code generation, the synthesized supervisor was verified for embedded deployment properties:

```
CONCLUSION:
    [OK] The specification has bounded response:
        - At most 2 iterations are needed for the event loop for uncontrollable events.
        - At most 2 iterations are needed for the event loop for controllable events.
    [OK] The specification is non-blocking under control.
    [OK] The specification has confluence.
```

Bounded Response: Both event loops part of the escet execution scheme terminate within 2 iterations maximum, guaranteeing bounded execution time. This ensures predictable worst-case cycle time.

Confluence: Controllable event processing order does not affect final state. When several controllable events are enabled, execution in any order produces identical results. This eliminates race conditions and ensures deterministic behavior. Category 8 requirements (Determinism and Priority Enforcement) explicitly prevent problematic event combinations.

The verified specification guarantees:
1. Real-time determinism (bounded 2-iteration worst-case)
2. Liveness (no deadlocks under the event-state semantics)
3. Consistency (order-independent behavior)
4. Safety (46 requirements enforced by construction)

These formal guarantees eliminate logic error classes that manual design cannot achieve.

These formal guarantees eradicate logical errors that would otherwise be exceedingly challenging, labor-intensive to avert solely through manual design. By mathematically enforcing correctness properties at the system level, they offer a degree of reliability, consistency, and verification that conventional hand-crafted methods cannot realistically assure, particularly as the complexity of the system increases.

4.5 Results

This section presents the deployment validation of the synthesized supervisor of our HVAC application .

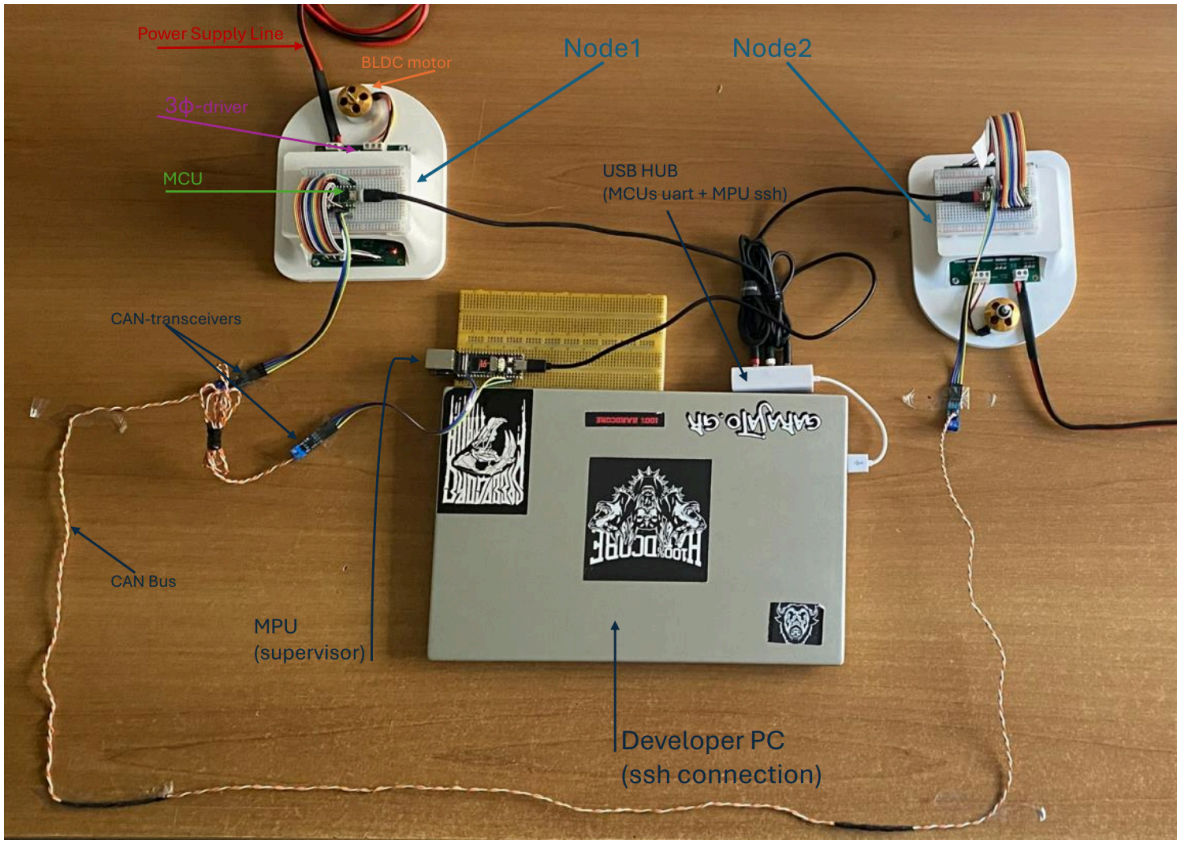


Figure 22: Physical test bench. Supervisor running in MPU, two DRV8302 motor drive nodes with BLDC motor, shared 12 V power supply, and CAN bus interconnect.

4.5.1 Code Generation and Integration

The synthesized and verified CIF specification of our supervisor called `output_NET_checked.cif` was compiled to C99 using ESCET's code generator, producing the supervisor library. The command `cifcodegen("Synth/output_NET_checked.cif -o gen/ -l c99 -p NET")` generated five files totalling approximately 2,400 lines: the model-specific engine (`NET_engine.c`, 1870 lines containing 49 edge functions), the runtime library (`NET_library.c`, 522 lines), and the associated headers. The generated engine was cross-compiled with `arm-linux-gnueabihf-gcc` in C99 mode and linked with three hand-written integration modules: the CAN interface library (`can_if.c`, 262 lines), the terminal user interface (`net_tui.c`, 348 lines), and the main supervisor loop (`main.c`, 204 lines). The resulting statically linked ARM binary occupies approximately 43 KB of memory at runtime (29.5 KB code, 960 bytes initialized data, 12.9 KB BSS) The integration effort is modest: approximately 814 lines of hand-written C bridge the gap between the generated supervisor and the physical hardware. The generated code requires no modification, all platform-specific behavior is confined to the three callback functions specified by the ESCET code generation interface.

4.5.2 CAN Interface Library

The CAN interface library (`can_if.c / can_if.h`) abstracts the Linux SocketCAN subsystem into the two logical channels required by the CIF event model.During initialization, two separate raw CAN sockets are created. The first is a non-blocking socket configured with filters to receive only packet type `0x4` frames, corresponding to uncontrollable node events. The second is a blocking transmit socket used to issue register commands through packet type `0x2` frames, which represent controllable events.

An bounded (128 entries) event queue implements the receive path. At the start of each supervisor cycle, the function `can_read_events()` drains all pending CAN frames from the kernel socket buffer into the queue. Each received frame is decoded by extracting the node address, event code, and sequence number from the 29-bit extended identifier. The function `can_event_triggered(node_id, event)` performs a linear scan of the queue to check whether a specific event has been received, and `can_event_queue_clear()` resets the queue after the supervisor cycle completes. Kernel-level nanosecond timestamps are extracted from ancillary socket data (SO_TIMESTAMPNS) and stored alongside each event for diagnostic logging.

The transmit path provides two functions(Table 12):

- 1. `can_cmd_simple(node_id, register)` sends a zero-payload CAN write request for state-transition commands (calibrate, enable, stop, recover, reboot).
- 2. `can_cmd_velocity(node_id, velocity)` sends a 4-byte IEEE 754 float payload to the velocity target register (0xF6). Both functions assemble the 29-bit extended CAN identifier from the node address, packet type, register address, and motor index fields defined by the CANCommander protocol.

The hardware acceptance filter configured on the receive socket ensures that only event notifications reach the supervisor, register responses and command frames are discarded at the kernel level, reducing processing overhead.

4.5.3 Supervisor Execution Loop

The main application runs a simple loop that repeats every 10 ms, meaning we have a control cycle of $\frac{1}{10ms} = 100$ Hz. In each tick it does the same things in the same order :

1. Check whether the operator typed anything
2. Write the current velocity into the CIF model variable
3. Call the `NET_EngineTimeStep(0.01)` — which is the ESCET execution scheme —
4. Then refresh the terminal.

We measured how long the actual control work takes inside that loop. That includes flushing the CAN event queue, assigning the input variables, running the full `PerformEdges()` pass with both event loops, and sending out CAN commands through the `InfoEvent()` callbacks. On our deployment hardware the whole sequence took between 25 µs and 125 µs, comfortably within the 10 ms budget. This leaves over 98% of the 10 ms tick period idle providing substantial margin for additional nodes, more complex requirements, or higher-priority tasks sharing the same processor. The supervisor could operate at cycle periods well below 10 ms if the application demanded faster response, though the 10 ms period (100Hz) is already well below the thermal and mechanical time constants of the HVAC fan application. The two ESCET callback functions connect the generated supervisor to the physical system:

- `NET_AssignInputVariables()` is called at the start of each engine step. It clears the event queue, reads all pending CAN frames, and maps each node's event flags to the corresponding CIF input variables. HMI flags (start, stop, technician reset) are consumed and cleared after each cycle. This implements the single-sample-per-cycle semantics required by the ESCET execution scheme.
- `NET_InfoEvent()` is called when each controllable event fires. It maps the CIF event identifier to the corresponding CAN register write via a switch statement, for example, when the supervisor fires `Net_NODE1___c_calibrate_`, the callback transmits a write to register 0xF1 on node 1 , thus a `can_cmd_simple(1, 0xF1)` call. Each event is also timestamped and appended to the TUI event log.

On shutdown, the system continues running for up to 30 additional cycles, each separated by a delay, allowing pending commands to propagate and giving the coordinator time to settle into an idle state. If the coordinator becomes idle at any point during these cycles, the shutdown process exits early.

4.5.4 Deployment Platform

The supervisor runs on a Luckfox Lyra Plus single-board computer based on the Rockchip RK3506G2 SoC [18] (triple-core ARM Cortex-A7 at 1.2 GHz, 128 MB DDR3L) running Linux buildroot OS. The board connects to the CAN bus via an SN65HVD230 transceiver, accessed through the Linux SocketCAN driver (can0 interface at 1 Mbit/s). The operator/ technician interacts with the supervisor remotely via SSH over the board's 10/100 Mbps Ethernet port. We deliberately picked a platform that is not particularly powerful to see whether the synthesized supervisor could keep up on hardware you might actually find in a building automation cabinet. The fact that the cycle time stayed under 150 µs on that board tells us the computational overhead is negligible, and there is no reason this approach could not run on similar embedded targets.

4.5.5 Terminal User Interface

A text-based operator interface exposes our supervisor which renders ANSI escape sequences over the SSH session. The TUI operates in non-blocking raw terminal mode and redraws the full screen on every supervisor cycle.

The screen is split into four parts. A header at the top with the system name, uptime, and cycle time. Under that, a line showing the coordinator's current phase and velocity setpoint. Then a panel for each node — you can tell at a glance how things are going because the state is color-coded green, yellow, or red. Each panel also lists the velocity, any fault reason, and the five counters, which turn red when they hit their limit. At the bottom there is a small event log and a reminder of the keyboard controls.

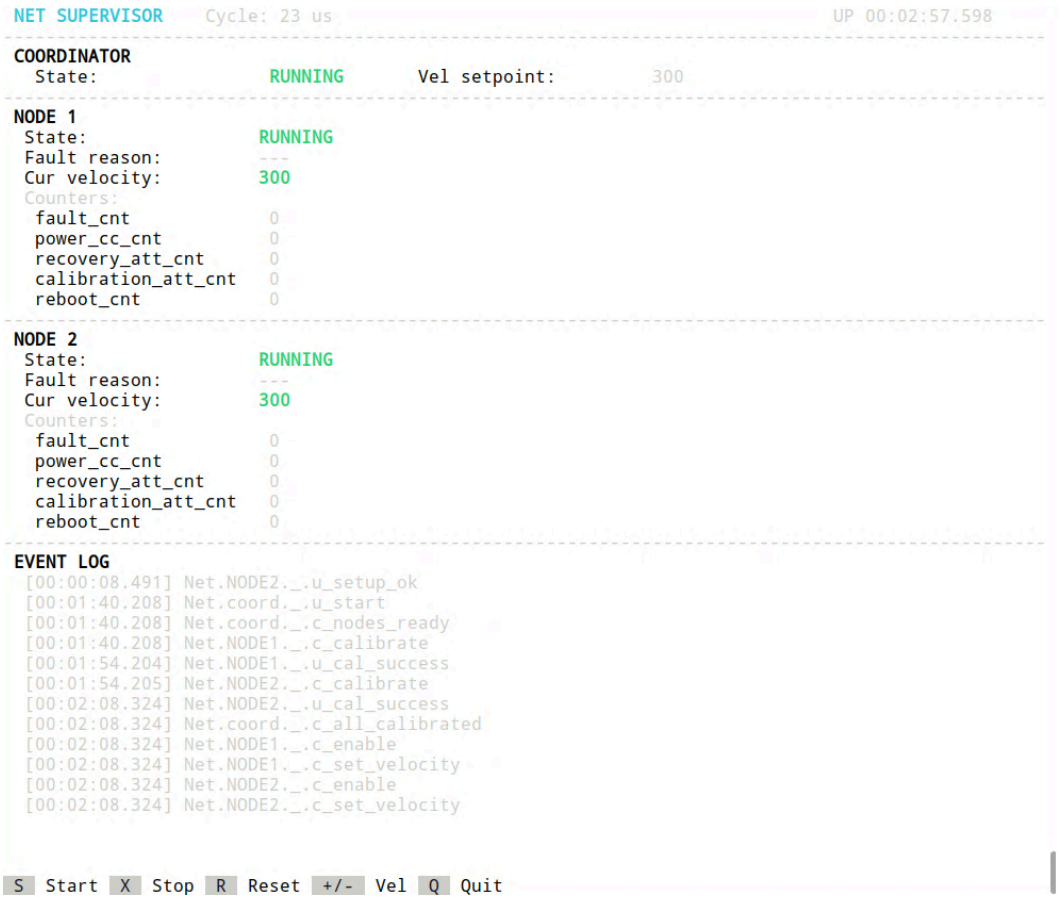


Figure 23: Supervisor TUI during operation, accessed via SSH connection. The display shows coordinator and node states with diagnostic counters, velocity setpoint, real-time event log, and operator key inputs

4.5.6 Testing on Real Hardware

We put the full system through its paces on the actual hardware to make sure everything worked as designed. Here's what happened in our key tests.

Normal Startup and Running

When we powered everything up, both motor nodes booted smoothly and signaled they were ready (`u_setup_ok`). The supervisor kicked off the full sequence: it went from IDLE to BOOTING, then CALIBRATING. NODE1 calibrated first (as required by R19–R20), then NODE2. Once both were good, the supervisor switched to RUNNING mode and enabled both motors at the same time. Operator velocity commands went to both nodes equally. Hitting stop sent brake commands to both, and the system safely dropped back to IDLE.

One-at-a-Time Calibration

The supervisor enforced the required order—NODE1 first, no exceptions. Even when both nodes needed recalibration (`NEEDS_RECAL`), it only sent the calibrate command (`c_calibrate`) to NODE1. NODE2 waited until NODE1 was done.

Clean Coordinated Stops

Operator stop commands put the supervisor into `STOPPING`, both nodes got stop signals and smoothly decelerated (exponential velocity ramp-down). The supervisor waited for both to confirm u_stopped and went back to IDLE.

Synchronized Motor Running



Figure 24: Phase currents from both nodes during synced operation. UART data from the Teensy boards shows both motors spinning at the exact same operator-commanded speed, with classic sinusoidal waveforms from Space Vector PWM under control.

In RUNNING mode, both motors chased the same velocity target from the TUI operator interface. The supervisor broadcast a single velocity setpoint to keep them perfectly in sync. Each node streamed its three-phase currents (from DRV8302 sense amps) over UART—smooth, same amplitude, frequency and approximately same phase sine waves confirmed healthy commutation on both sides.

Fault Testing (Breaking Things on Purpose)

To prove bidirectional fault handling, we injected faults from Table 4 by grounding DRV8302 pins **while motors were spinning**. Tested each fault type on NODE1 (watching NODE2 react), then swapped. Both stayed running continuously otherwise.

NODE1 Faults (NODE2 behaved symmetrically, just reversed order)

- **Electrical fault** (`nFAULT` low, `nOCTW` high): NODE1 shut down its motor locally and flagged `EVT_FAULT`. Supervisor stopped NODE2, recalibrated NODE1, and both went back to `RUNNING`.
- **Thermal fault** (`nFAULT` low, `nOCTW` low): NODE1 flagged `THERMAL` and broadcasts `EVT_OVERHEATED`.NODE2 stopped, recovery failed (`EVT_RECOVER_FAILED`) while pins stayed grounded. Released pins NODE1 saw them clear, reset EN_GATE, flagged `EVT_COOLED`. Recalibrated NODE1, and both went back to `RUNNING`.
- **Power fault** (`PWRGD` low): NODE1 went `NO_POWER`. Supervisor stopped NODE2. Released pin, NODE1 cleared its latch in `FAULT` state, normal recovery.

Counter Escalation (both nodes)

We made every counter type hit its threshold and triggered EMERGENCY:

- **Recovery fails** (`recovery_att_cnt`): Kept pins grounded during attempts.

- **Calibration fails** (`calibration_att_cnt`): Unplugged motor phases bad `characteriseMotor()` params.
- **Setup fails** (`reboot_cnt`): Floated ADC/driver pins → `setup()` failed every time, `EVT_SETUP_ERROR`.
- **Fault cycles** (`fault_cnt`): Repeated injections over recovery loops.
- **Power cycles** (`power_cc_cnt`): Pulsed `PWRGD` low repeatedly.

The supervisor locked out all auto-operations until manual tech reset. Local safeties kicked in instantly and all responses matched specs perfectly. No deadlocks, no unexpected states observed.

4.5.7 Quantitative Summary

The below table summarizes the key metrics of the implementation.

Model Metrics	
CIF model size (plants, requirements)	844 lines
Requirements count	46
Synthesis Time	$\sim 2.5\,\text{sec}$
Code Generation Metrics	
Generated C99 code (engine + library)	2,392 lines
Hand-written integration code (CAN, TUI, main)	814 lines
Generated edge functions	49
Verification Results	
Bounded response (uncontrollable loop)	≤ 2 iterations
Bounded response (controllable loop)	≤ 2 iterations
Confluence	Verified
Non-blocking under control	Verified
Runtime Metrics	
Supervisor runtime footprint (text + data + bss)	≈ 43 KB
Application supervisor cycle	10 ms (100 Hz)
Measured cycle execution time (engine)	< 150 µs
Communication & Platform	
CAN bus type	2.0B
CAN bus bitrate	1 Mbit/s
Node platform	Teensy 4.0 (MCU)
Supervisor deployment platform	Luckfox Lyra Plus (Cortex-A7, 1.2 GHz

5 Conclusions

5.1 Summary

This thesis presented a two-layer embedded control architecture that combines formal supervisory synthesis with classical real-time control. The **Supervisor** upper layer was modeled in CIF and synthesized using the Eclipse ESCET toolkit, giving us a coordinator whose discrete logic comes with mathematical correctness guarantees. The **Resource Controllers** lower layer which is made up of the individual nodes, each running field-oriented motor control locally through the SimpleFOC library and governed by a state machine designed around functional safety principles: fail-safe defaults, defensive programming, and clearly defined fault states. The two layers talk to each other over CAN 2.0B bus, nodes report what happened (uncontrollable events), and the supervisor tells them what to do next (controllable events).

We validated the approach on a small but representative setup: two CAN-connected nodes driving BLDC motors, coordinated by the synthesized supervisor, in a balanced ventilation (HRV) scenario. The CIF models , plants, coordinator, and all 46 formal requirements came out to just 844 lines. From those, synthesis produced a verified supervisor that generated 2,392 lines of C99 code. We combined that with 814 lines of hand-written platform code covering the CAN bus communication layer and the terminal interface described earlier, and deployed everything on a single-board computer. The controller properties checker confirmed bounded response within 2 iterations,confluence, and non-blocking under control, which together mean the supervisor is deterministic, deadlock-free, and behaves the same regardless of the order events arrive in.

Testing covered normal operation cycles, sequential calibration, coordinated shutdown, and physical fault injection. We induced electrical faults, thermal faults, and power supply loss by grounding the DRV8302 diagnostic pins while the motors were running. Every time, the system did what it was supposed to: the nodes caught the fault locally and shut down the motor immediately, and the supervisor on top handled coordinated recovery, tracked the counters, and escalated to emergency when needed — all exactly matching what the 46 requirements prescribed.

5.2 Contributions

This thesis presents a two-layer embedded control architecture in which a synthesized supervisor coordinates hand-written node firmware while preserving formal correctness across their boundary. Preservation of guarantees is achieved by a tightly specified interface – input variables, plant invariants, and event callbacks – so that synthesis assumptions map directly to the deployed software.

The CIF components models and the 46 formalized requirements were co-developed from a Heat Recovery Ventilation unit use case. Co-specifying models and requirements increased the fidelity of the synthesis input, enabling the synthesized supervisor to reason about realistic system behavior.

Nodes implement local fail-safe behavior: each node stops its motor immediately when a hardware fault is detected, independent of the supervisor's state. This defense in depth,

combined with supervisor coordination, follows layered-protection thinking from IEC 61508-3 [9] and preserves safety during supervisor latency or temporary loss.

The work documents and end-to-end process from CIF requirements to deployed code on CAN-equipped hardware: plant modeling, requirement formalization, synthesis, interactive simulation, property checking, code generation, cross-compilation, and hardware integration. A practical contribution is a two-phase synthesis method: first synthesize an abstract supervisor for simulation (no hardware bindings), the re-synthesize with concrete bindings for deployment.This separation addresses simulator limitations with input variables and simplifies platform integration.

A set of transferable modelling patterns emerged from this work. Structuring counters (cumulative versus consecutive and when to reset them based on state) repeatedly affected design choices. Deciding which events are controllable versus uncontrollable clarified where observation and authority reside in the system. The one-to-one mapping between CIF events and CAN frames provided a clean, debuggable interface that other distributed setups can reuse.By combining correct-by-construction synthesis with co-developed component models and real, product-derived requirements, the approach offers a practical path for shipping more correct embedded software and accelerating development for mass-produced products.

5.3 Limitations

There are a few things this work does not cover, and being upfront about them matters for anyone who might try to build on it.

The most obvious one is scale. The current implementation demonstrates $N = 2$ nodes. To put this in perspective, consider the BDD footprint of our models. The ESCET documentation explains that each CIF variable is internally represented using boolean BDD variables: a `bool` uses 1 bit, an `int[0..k]` uses $\log_2(k+1)$ bits, and each automaton with multiple locations gets a location pointer variable [2]. In our two-node model this adds up to roughly 86 BDD variables across bot nodes and the coordinator. Synthesis completed in 2.5 seconds because ESCET's symbolic BDD-based approach efficiently compresses the reachable state space without explicit enumeration [2]. However, each additional node would add approximately 37 BDD variable ordering plus new cross-node requirements and the synchronous composition grows combinatorially with the number of components [6]. The ESCET documentation notes that BDD variable ordering "can significantly influence the performance of synthesis" and that BDD operation cache can become "a common cause of out-of-memory errors" when they exceed available CPU cache capacity [2]. Whether a 10-node or 50-node variant remains tractable under monolithic synthesis is an open empirical question we did not investigate.

The motor control itself is open-loop. SimpleFOC is running in velocity mode without current feedback, which is fine for fans but would not cut it for anything requiring precise torque control. The good news is that switching to closed-loop current control is a configuration change inside SimpleFOC and does not touch the supervisor or the CAN layer at all. You would want current sensors and possibly an encoder on the hardware side, and you could then do a proper calibration routine that measures the motor's KV rating and pole pairs automatically. We did not implement any of that, but nothing in the architecture prevents it.

The fault model has gaps too, we modeled what the DRV8302 actually reports, thermal shutdown, electrical faults, power loss, but we did not model CAN bus failures. No message loss, no bus-off detection, no timeout handling at the supervisor level. The nodes are safe regardless, because the local fault reaction does not depend on the network, but the supervisor's picture of what is happening relies on CAN messages actually arriving. In a noisier electrical environment that assumption would need revisiting.

And last, the generated C99 code prioritizes correctnesses and portability over compactness. The entire engine is large (1870 lines): each automaton location is flattened into a single enum and every edge uses the same repetitive guard-update pattern, producing a long procedural sequence with no structs or organised data layouts. It runs well on our MPU (Cortex-A7) and meets cycle-time requirements, but on smaller mcus with strict memory limits you may want to post-process or hand optimize the output. That is a limitation of the code generator, not of the modeling aproach.

5.4 Future Work

The most immediate next step would be closing the motor control loop. SimpleFOC supports `TorqueControlType::foc_current` out of the box, and the DRV8302 boards already have the current sensing hardware wired up. Switching to closed-loop is essentially a one-line configuration change in the firmware, and it would not affect the supervisor or communication layers at all.

Modeling CAN bus failures would be a bigger effort but a valuable one. You would extend the CIF plant models to include message loss and bus-off conditions, add timeout-based detection logic at the supervisor level, and possibly model redundant communication paths or heartbeat protocols as additional plant automata with their own requirements. The result would be a supervisor that handles network faults with the same rigor it currently handles motor faults.

Scaling to networks with significantly more than tow nodes is the open question we find most interesting. As discussed in the limitations, each additional node adds roughly 37 BDD variables and a proportional increase in the cross-node requirements, and monolithic synthesis may eventually become impractical. The ESCET toolkit already provides a path forward through the CIF multi-level splitter [2], which groups plant components and requirements into clusters and synthesizes a separate supervisor for each group. Moormann et al. [13] have demonstrated synthesis and implementation of distributed supervisory controllers with communication delays, showing that the decomposition approach is viable for network systems. This compositional strategy aligns naturally with both the hierarchical control architecture described in Figure 5 of Chapter 1, which already isolates per-node behavior inot identical, independently instantiated CIF definitions. Investigating whether multi-level synthesis maintains tractable synthesis times for larger networks while preserving the formal guarantees established in this work is a natural next step.

5.5 Closing Remarks

What synthesis-based engineering really buys you is a different trade-off in where you spend your effort. Instead of writing coordination logic by hand and then trying to test every corner case, you write a formal specification and let the toolchain produce code that is correct by construction. Deadlocks, unsafe interlocks, race conditions, the kinds of bugs that slip

through testing because they only show up under specific timing or event orderings, are eliminated structurally rather than caught after the fact.

The two-layer split we used keeps that advantage contained where it belongs. The real-time motor control code, which is well understood and has been validated on its own, stays exactly as it was. Only the coordination logic on top gets replaced with synthesized code. You do not have to formally model everything to get value from formal methods, just the parts where the complexity of interactions between components makes manual reasoning unreliable.

In practical terms, the modeling effort was not enormous. The 844 lines of CIF specification we wrote produced a supervisor that handles 46 requirements across two nodes, with fault recovery, counter tracking, sequential resource sharing, and emergency escalation. Writing that by hand, with equivalent coverage, would take considerably more code, and you still would not have the formal guarantee that the logic is deadlock-free and deterministic. The 814 lines of platform code we wrote to glue it all together were mostly mechanical: mapping CIF events to CAN frames and drawing the terminal display.

The ventilation system we built is not a factory floor or a semiconductor fab, which is where supervisory control theory has mostly lived until now. But building automation has real coordination requirements and real safety consequences when those requirements are violated. If the methodology and patterns from this thesis make it a little easier for an embedded engineer to pick up CIF and ESCET and try synthesis on their own system, then the work was worth doing.

Bibliography

[1] N. Sridhar, "Power Electronics in Motor Drives: Where is it?," White paper SLYY78A, Oct. 2019.

[2] Eclipse Foundation, "Eclipse ESCET – The Eclipse Supervisory Control Engineering Toolkit." 2023.

[3] W. M. Wonham and K. Cai, *Supervisory Control of Discrete-Event Systems*. Springer, 2019. doi: 10.1007/978-3-319-77452-7.

[4] H. Awad, "Supervisory Control Systems: Theory and Industrial Applications," *Petri Nets in Science and Engineering*, pp. 93–109, 2018, doi: 10.5772/intechopen.75166.

[5] J. C. Kaandorp, "Documenting and Controlling the Behavior of the ASML Lot Operations Expose Wafer," Master's thesis, 2023.

[6] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, 3rd ed. Springer, 2021. doi: 10.1007/978-3-030-72274-6.

[7] P. J. Ramadge and W. M. Wonham, "Supervisory Control of a Class of Discrete Event Processes," *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp. 206–230, 1987, doi: 10.1137/0325013.

[8] "IEC 61508-1:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 1: General requirements," no. IEC61508–1. 2010.

[9] "IEC 61508-3:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements," no. IEC61508–3. 2010.

[10] "IEC 61508-4:2010 Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 4: Definitions and abbreviations," no. IEC61508–4. 2010.

[11] M. Sköldstam, K. Åkesson, and M. Fabian, "Supervisory Control Applied to Automata Extended with Variables – Revised," Göteborg, Sweden, technical report R3/2007, 2007.

[12] L. Ouedraogo, R. Kumar, R. Malik, and K. Åkesson, "Nonblocking and Safe Control of Discrete-Event Systems Modeled as Extended Finite Automata," *IEEE Transactions on Automation Science and Engineering*, vol. 8, no. 3, pp. 560–569, 2011, doi: 10.1109/TASE.2011.2124457.

[13] L. Moormann, R. H. J. Schouten, J. M. van de Mortel-Fronczak, W. J. Fokkink, and J. E. Rooda, "Synthesis and Implementation of Distributed Supervisory Controllers With Communication Delays," *IEEE Transactions on Automation Science and Engineering*, vol. 20, no. 3, pp. 1591–1606, 2023, doi: 10.1109/TASE.2023.3260442.

[14] U.S. Department of Energy, "Heat Recovery Ventilators: Technology and Installation Guide." 2021. [Online]. Available: https://www.energy.gov/energysaver/heat-and-energy-recovery-ventilators

[15] ASHRAE, "ASHRAE Standard 62.2-2019: Ventilation and Acceptable Indoor Air Quality in Low-Rise Residential Buildings," technical report, 2019.

[16] Texas Instruments, “DRV8302 Three Phase Gate Driver With Dual Current Shunt Amplifiers and Buck Regulator – Hardware Controlled.” Mar. 2016. [Online]. Available: https://www.ti.com/lit/ds/symlink/drv8302.pdf

[17] A. Skuric, H. S. Bank, R. Unger, O. Williams, and D. González-Reyes, “SimpleFOC: A Field Oriented Control (FOC) Library for Controlling Brushless Direct Current (BLDC) and Stepper Motors,” *Journal of Open Source Software*, vol. 7, no. 74, p. 4232, 2022, doi: 10.21105/joss.04232.

[18] Rockchip Electronics Co., Ltd., “RK3506G2 Datasheet, Revision 1.3.” Aug. 2025. [Online]. Available: https://opensource.rock-chips.com/images/5/51/Rockchip_RK3506G2_Datasheet_V1.3-20250811.pdf

6 Appendices

6.1 Appendix A: Node Plant Model

```
// NODE.cif
// Author: Stavros Martini 11/3/26
// Thesis: Model Based Synthesis and Validation of High Performance
//         Supervisory Controllers for Embedded Systems
// DRV8302 BLDC Motor Node -
//================================================================================
//
// HARDWARE: 7-pole BLDC, DRV8302 driver, Teensy 4.0, SimpleFOC
// FAULTS: ELECTRICAL (nFAULT=0,nOCTW=1), THERMAL (nFAULT=0,nOCTW=0), NO_POWER (PWRGD=0)
// RECOVERY: Electrical EN_GATE reset, Thermal -> passive cooling, Power -> restore+recover


enum FaultReason = NONE, ELECTRICAL, THERMAL, POWER_RESTORE;

plant def DRV8302_motor_node(uncontrollable u_technician_reset; alg int [0..300] tar_vel):

    const int MAX_RANGE = 10;

    //================================================================================
    // COUNTERS (Supervisor-visible safety metrics)
    //================================================================================
    // Increment: min(counter+1, MAX_RANGE) - saturating arithmetic
    // Reset: Successful calibration (all operational) or reboot (all)

    disc int[0..MAX_RANGE] fault_cnt = 0,              // u_fault, u_overheated (total faults)
                           power_cc_cnt = 0,         // u_no_power        (total power cycles)
                           recovery_att_cnt = 0,   // u_recover_failed  (consecutive recoveries)
                           calibration_att_cnt = 0,// u_cal_failed      (consecutive calibrations)
                           reboot_cnt = 0;          // c_reboot          (consecutive reboots)
    //================================================================================
    // RUNTIME STATE
    //================================================================================

    disc int[0..300] cur_vel = 0;  // Velocity setpoint and actual
    disc bool rec_flag = true , delay_passed= true;                 // Recovery protocol (alternating cmd/rsp)
    disc FaultReason fault_reason = NONE;      // Why we're in FAULT (determines recovery path)

    //================================================================================
    // EVENTS
    //================================================================================

    // Controllable (supervisor commands)
    controllable c_calibrate, c_enable, c_stop, c_recover, c_cal_reject, c_reboot, c_set_velocity;

    // Uncontrollable (hardware events)
    uncontrollable u_setup_ok, u_setup_error;
    uncontrollable u_cal_success, u_cal_failed;
    uncontrollable u_fault, u_overheated, u_cooled, u_fault_cleared, u_recover_failed;
    uncontrollable u_no_power, u_power_restore;
    uncontrollable u_stopped;

    //================================================================================
    // STATE MACHINE
    //================================================================================
    // INIT: Hardware initialization (driver, motor, current sense)
    location INIT:
        initial; marked;
        edge u_setup_ok do reboot_cnt := 0 goto NEEDS_RECAL;
        edge u_setup_error goto SETUP_ERROR;
        edge u_fault do
            fault_cnt := min(fault_cnt + 1, MAX_RANGE),
            fault_reason := ELECTRICAL
            goto FAULT;
        edge u_overheated do
            fault_cnt := min(fault_cnt + 1, MAX_RANGE),
            fault_reason := THERMAL
            goto FAULT;
        edge u_no_power do
```

```
            power_cc_cnt := min(power_cc_cnt + 1, MAX_RANGE)
            goto NO_POWER;

        // Technician reset from INIT - go to safe state
        edge u_technician_reset ;

    // SETUP_ERROR: Init failed - wait for reboot
    location SETUP_ERROR:
        marked;
        edge c_reboot do
            reboot_cnt := min(reboot_cnt + 1, MAX_RANGE)
            goto INIT;

        // Technician reset from SETUP_ERROR - clear counters and recover
        edge u_technician_reset do
            fault_cnt := 0,
            recovery_att_cnt := 0,
            calibration_att_cnt := 0,
            power_cc_cnt := 0,
            reboot_cnt := 0;

    // NEEDS_RECAL: Motor uncalibrated - parameters unknown
    location NEEDS_RECAL:
        marked;
        edge c_calibrate goto CALIBRATING;
        edge u_fault do
            fault_cnt := min(fault_cnt + 1, MAX_RANGE),
            fault_reason := ELECTRICAL
            goto FAULT;
        edge u_overheated do
            fault_cnt := min(fault_cnt + 1, MAX_RANGE),
            fault_reason := THERMAL
            goto FAULT;
        edge u_no_power do
            power_cc_cnt := min(power_cc_cnt + 1, MAX_RANGE)
            goto NO_POWER;
        // Technician reset from NEEDS_RECAL - no-op (already safe state)
        edge u_technician_reset do
            fault_cnt := 0,
            recovery_att_cnt := 0,
            calibration_att_cnt := 0,
            power_cc_cnt := 0,
            reboot_cnt := 0;
    // CALIBRATING: Measuring R, L parameters (~10-15sec)
    location CALIBRATING:
        edge u_cal_success do
            calibration_att_cnt := 0
            goto IDLE;
        edge u_cal_failed do
            calibration_att_cnt := min(calibration_att_cnt + 1, MAX_RANGE)
            goto CAL_FAILED;
        edge u_fault do
            fault_cnt := min(fault_cnt + 1, MAX_RANGE),
            fault_reason := ELECTRICAL
            goto FAULT;
        edge u_overheated do
            fault_cnt := min(fault_cnt + 1, MAX_RANGE),
            fault_reason := THERMAL
            goto FAULT;
        edge u_no_power do
            power_cc_cnt := min(power_cc_cnt + 1, MAX_RANGE)
            goto NO_POWER;
        edge u_technician_reset;

    // CAL_FAILED: Calibration failed - retry or abandon
    location CAL_FAILED:
        marked;
        edge c_calibrate goto CALIBRATING;
        edge c_cal_reject goto NEEDS_RECAL;
        edge u_fault do
            fault_cnt := min(fault_cnt + 1, MAX_RANGE),
            fault_reason := ELECTRICAL
            goto FAULT;
```

```
        edge u_overheated do
            fault_cnt := min(fault_cnt + 1, MAX_RANGE),
            fault_reason := THERMAL
            goto FAULT;
        edge u_no_power do
            power_cc_cnt := min(power_cc_cnt + 1, MAX_RANGE)
            goto NO_POWER;

        // Technician reset from CAL_FAILED - clear counters and recover
        edge u_technician_reset do
            fault_cnt := 0,
            recovery_att_cnt := 0,
            calibration_att_cnt := 0,
            power_cc_cnt := 0,
            reboot_cnt := 0;
    // IDLE: Calibrated, motor disabled, ready for operation
    location IDLE:
        marked;
        edge c_enable goto RUNNING;
        edge u_fault do
            fault_cnt := min(fault_cnt + 1, MAX_RANGE),
            fault_reason := ELECTRICAL
            goto FAULT;
        edge u_overheated do
            fault_cnt := min(fault_cnt + 1, MAX_RANGE),
            fault_reason := THERMAL
            goto FAULT;
        edge u_no_power do
            power_cc_cnt := min(power_cc_cnt + 1, MAX_RANGE)
            goto NO_POWER;

        // Technician reset from IDLE - no-op (already safe state)
        edge u_technician_reset;

    // RUNNING: FOC active, motor spinning
    location RUNNING:
        edge c_set_velocity when cur_vel != tar_vel do cur_vel := tar_vel;
        edge c_stop goto STOPPING;
        edge u_fault do
            fault_cnt := min(fault_cnt + 1, MAX_RANGE),
            cur_vel := 0,
            fault_reason := ELECTRICAL
            goto FAULT;
        edge u_overheated do
            fault_cnt := min(fault_cnt + 1, MAX_RANGE),
            cur_vel := 0,
            fault_reason := THERMAL
            goto FAULT;
        edge u_no_power do
            power_cc_cnt := min(power_cc_cnt + 1, MAX_RANGE),
            cur_vel := 0
            goto NO_POWER;

        // Technician reset from RUNNING - stop motor, reset counters
        edge u_technician_reset;

    // STOPPING: Controlled deceleration (exponential decay)
    location STOPPING:
        edge u_stopped do cur_vel := 0 goto IDLE;
        edge c_enable goto RUNNING;
        edge u_fault do
            fault_cnt := min(fault_cnt + 1, MAX_RANGE),
            cur_vel := 0,
            fault_reason := ELECTRICAL
            goto FAULT;
        edge u_overheated do
            fault_cnt := min(fault_cnt + 1, MAX_RANGE),
            cur_vel := 0,
            fault_reason := THERMAL
            goto FAULT;
        edge u_no_power do
            power_cc_cnt := min(power_cc_cnt + 1, MAX_RANGE),
            cur_vel := 0
```

```
            goto NO_POWER;

        // Technician reset from STOPPING - abort stop, reset counters
        edge u_technician_reset ;

    // FAULT: Hardware fault - type-specific recovery required
    // rec_flag alternates: c_recover sets false, responses set true
    location FAULT:
        marked;

        edge c_recover when rec_flag and delay_passed do
            rec_flag := not rec_flag
            goto FAULT;

        edge u_recover_failed when not rec_flag do
            recovery_att_cnt := min(recovery_att_cnt + 1, MAX_RANGE),
            rec_flag := not rec_flag
            goto FAULT;

        // Thermal fault cleared (only fires when fault_reason = THERMAL)
        edge u_cooled when fault_reason = THERMAL do
            recovery_att_cnt := 0,
            fault_reason := NONE,
            rec_flag := true
            goto NEEDS_RECAL;

        // Electrical fault cleared (fires when ELECTRICAL or POWER_RESTORE)
        edge u_fault_cleared when (fault_reason = ELECTRICAL or fault_reason = POWER_RESTORE) do
            recovery_att_cnt := 0,
            fault_reason := NONE,
            rec_flag := true
            goto NEEDS_RECAL;

        edge u_no_power do
            power_cc_cnt := min(power_cc_cnt + 1, MAX_RANGE),
            rec_flag := true,
            fault_reason := NONE
            goto NO_POWER;

        // Technician reset from FAULT - full counter reset and recover
        edge u_technician_reset do
            fault_cnt := 0,
            recovery_att_cnt := 0,
            calibration_att_cnt := 0,
            power_cc_cnt := 0,
            reboot_cnt := 0;

    // NO_POWER: Power supply lost - wait for restore
    location NO_POWER:
        marked;
        edge u_power_restore do
            fault_reason := POWER_RESTORE
            goto FAULT;
        edge u_technician_reset;
end

//================================================================================
// INPUT CAN GROUP
//================================================================================

 // Input group definition
 group def CAN_IF():
     // Event flags from CAN node
     input bool Evt_setup_ok      , Evt_setup_error;
     input bool Evt_cal_success   , Evt_cal_failed;
     input bool Evt_stopped       , Evt_fault;
     input bool Evt_no_power      , Evt_power_restored;
     input bool Evt_overheated    , Evt_cooled;
     input bool Evt_fault_cleared , Evt_recover_failed;
 end

 //================================================================================
// NODE GROUP DEFINITION
```

```
//============================================================================
//
// Wraps the motor node plant for network instantiation.
// node_id is algebraic parameter (compile-time constant) for CAN addressing.
//

group  def  NODE(alg int [1..254] node_id ; uncontrollable u_technician_reset ; alg int [0..300] tar_vel):
       _: DRV8302_motor_node(u_technician_reset, tar_vel);
     CAN: CAN_IF();
        // Plant invariants linking events to input flags
     plant invariant _.u_setup_ok         needs CAN.Evt_setup_ok;
     plant invariant _.u_setup_error      needs CAN.Evt_setup_error;
     plant invariant _.u_cal_success      needs CAN.Evt_cal_success;
     plant invariant _.u_cal_failed       needs CAN.Evt_cal_failed;
     plant invariant _.u_stopped          needs CAN.Evt_stopped;
     plant invariant _.u_fault            needs CAN.Evt_fault;
     plant invariant _.u_no_power         needs CAN.Evt_no_power;
     plant invariant _.u_power_restore    needs (CAN.Evt_power_restored and not CAN.Evt_no_power);
     plant invariant _.u_overheated       needs CAN.Evt_overheated;
     plant invariant _.u_cooled           needs CAN.Evt_cooled          and not CAN.Evt_overheated;
     plant invariant _.u_fault_cleared    needs CAN.Evt_fault_cleared   and not CAN.Evt_fault;
     plant invariant _.u_recover_failed   needs CAN.Evt_recover_failed;

end
```

6.2 Appendix B: Coordinator Model

```
//Coordinator.cif
// Author: Stavros Martini 11/3/26
// Thesis: Model Based Synthesis and Validation of High Performance
//         Supervisory Controllers for Embedded Systems
   /*
DISTRIBUTED BLDC MOTOR CONTROL NETWORK

GENERAL SYSTEM:
  - Scalable architecture (N motors on CAN bus)
  - Coordinated control with safety supervision
  - Autonomous fault recovery
  - Counter-based safety limits

EXPERIMENTAL DEMONSTRATION (N=2):
  - Application: Balanced Ventilation System (HRV/ERV)
  - NODE1: Supply Fan (fresh air intake)
  - NODE2: Exhaust Fan (stale air exhaust)
  - Critical requirement: Balanced airflow (pressure stability)

EXTENSIBILITY:
  - N=3: Triple-fan system (supply + exhaust + recirculation)
  - N=4: Dual-zone HVAC (two independent HRV units)
  - N=6: Multi-zone building automation
  - Other: Conveyor systems, pump arrays, multi-axis machines

*/
//============================================================================
// System Coordinator - Distributed Motor Control Network
//============================================================================
//
// GENERAL: Orchestrates N motor nodes for coordinated operation
// EXPERIMENTAL CASE (N=2): Balanced Ventilation System (HRV/ERV)
//   - NODE1: Supply fan (fresh air intake)
//   - NODE2: Exhaust fan (stale air exhaust)
//
// SCALABILITY: Architecture supports arbitrary N nodes on CAN bus
//   - Requirements adapt to application (balance, sync, independence)
//   - Coordinator phases remain consistent across applications
//
//============================================================================

plant def sys_coordinator():

    // OPERATOR INTERFACE (Application-specific)
    //============================================================================
```

```
// General: System start/stop commands
// HVAC case: Building automation or wall controller

uncontrollable u_start;  // Operator: Start coordinated operation
uncontrollable u_stop;   // Operator: Stop coordinated operation
uncontrollable u_technician_reset;

// AUTONOMOUS TRANSITIONS (Supervisor-determined)
//============================================================================
// General: System-level state transitions based on node health
// Requirements define exact conditions per application

controllable c_nodes_ready;      // All nodes initialized
controllable c_all_calibrated;   // All nodes characterized
controllable c_all_stopped;      // All nodes stopped
controllable c_node_unhealthy;   // Any node in fault/error state
controllable c_all_healthy;      // All nodes recovered
controllable c_limits_exceeded;  // Safety thresholds exceeded
controllable c_all_power_lost;   // Power failure (all nodes)
controllable c_power_restored;   // Power restored

disc int [0..300] velocity=1;
//============================================================================
// SYSTEM LIFECYCLE (Application-independent phases)
//============================================================================

// IDLE: System off, nodes should be idle or in error states
location IDLE:
    initial; marked;
    edge u_start goto BOOTING;
    edge c_all_power_lost goto NO_POWER;

// BOOTING: Nodes transitioning from INIT to operational readiness
location BOOTING:
    edge c_nodes_ready goto CALIBRATING;
    edge c_node_unhealthy goto RECOVERING;
    edge c_limits_exceeded goto EMERGENCY;
    edge u_stop goto IDLE;
    edge c_all_power_lost goto NO_POWER;

// CALIBRATING: Parameter characterization (sequential per hardware constraints)
location CALIBRATING:
    edge c_all_calibrated goto RUNNING;
    edge c_node_unhealthy goto RECOVERING;
    edge c_limits_exceeded goto EMERGENCY;
    edge u_stop goto STOPPING;
    edge c_all_power_lost goto NO_POWER;

// RUNNING: Normal coordinated operation
location RUNNING:
    marked;
    edge c_node_unhealthy goto RECOVERING;
    edge u_stop goto STOPPING;
    edge c_all_power_lost goto NO_POWER;

// STOPPING: Coordinated shutdown
location STOPPING:
    edge c_all_stopped goto IDLE;
    edge c_node_unhealthy goto RECOVERING;
    edge c_all_power_lost goto NO_POWER;

// RECOVERING: Fault recovery (sequential per power supply constraints)
location RECOVERING:
    marked;
    edge c_all_healthy goto BOOTING;
    edge c_limits_exceeded goto EMERGENCY;
    edge c_all_power_lost goto NO_POWER;

// EMERGENCY: Safety limits exceeded - operator intervention required
location EMERGENCY:
    marked;
    edge u_technician_reset goto BOOTING;  // Reset before restart
    edge c_all_power_lost goto NO_POWER;
```

```
    // NO_POWER: Power supply failure
    location NO_POWER:
        marked;
        edge c_power_restored goto RECOVERING;
end


group def HMI_IF():
    input bool _u_start, _u_stop, _u_technician_reset;
end

group def System_Coordinator():
     _    : sys_coordinator();
// linking u events to Human Machine Interface
     HMI : HMI_IF();
     plant invariant _.u_start                      needs HMI._u_start and not HMI._u_stop;
     plant invariant _.u_stop                       needs HMI._u_stop and not HMI._u_start;
     plant invariant _.u_technician_reset           needs HMI._u_technician_reset;
end
```

6.3 Appendix C: Net Definition and Requirements

```
// Author: Stavros Martini 11/3/26
// Thesis: Model Based Synthesis and Validation of High Performance
//         Supervisory Controllers for Embedded Systems
import "NODE.cif";
import "Coordinator.cif";
//
// Network Definition
//
group Net:
    coord: System_Coordinator();
    NODE1: NODE(1,coord._.u_technician_reset, coord._.velocity);
    NODE2: NODE(2,coord._.u_technician_reset, coord._.velocity);
end
```

```
//=============================================================================
// Requirements.cif
// Distributed Motor Control Network -- System Requirements
//
// Author: Stavros Martini 11/3/26
// Thesis: Model Based Synthesis and Validation of High Performance
//         Supervisory Controllers for Embedded Systems
//
// Two-node balanced ventilation (HRV) setup:
//   NODE1 = supply fan  (outdoor air in)
//   NODE2 = exhaust fan (indoor air out)
//
// Main rule: both fans run together or neither runs.
//=============================================================================

import "NET.cif";

// Tunable limits -- change these to match your hardware/application
const int MAX_FAULTS   = 5;
const int MAX_POWER    = 3;
const int MAX_CAL      = 3;
const int MAX_RECOVERY = 3;
const int MAX_REBOOT   = 3;

// True when any counter on any node has hit its limit
alg bool LIMIT_EXHAUSTED = (
    Net.NODE1._.fault_cnt >= MAX_FAULTS or
    Net.NODE1._.power_cc_cnt >= MAX_POWER or
    Net.NODE1._.recovery_att_cnt >= MAX_RECOVERY or
    Net.NODE1._.calibration_att_cnt >= MAX_CAL or
    Net.NODE1._.reboot_cnt >= MAX_REBOOT or
    Net.NODE2._.fault_cnt >= MAX_FAULTS or
    Net.NODE2._.power_cc_cnt >= MAX_POWER or
```

```
    Net.NODE2._.recovery_att_cnt >= MAX_RECOVERY or
    Net.NODE2._.calibration_att_cnt >= MAX_CAL or
    Net.NODE2._.reboot_cnt >= MAX_REBOOT
);


//=============================================================================
// Category 1: Coordinator transitions  (R1-R8)
// When can the coordinator move between phases?
//=============================================================================

// R1: both nodes done with init, ready to calibrate
requirement Net.coord._.c_nodes_ready needs
    (Net.NODE1._.NEEDS_RECAL or Net.NODE1._.IDLE or Net.NODE1._.CAL_FAILED) and
    (Net.NODE2._.NEEDS_RECAL or Net.NODE2._.IDLE or Net.NODE2._.CAL_FAILED);

// R2: both calibrated and counters are fine, we can run
requirement Net.coord._.c_all_calibrated needs
    (Net.NODE1._.IDLE and Net.NODE2._.IDLE) and (not LIMIT_EXHAUSTED);

// R3: both fans confirmed stopped, safe to go back to idle
requirement Net.coord._.c_all_stopped needs
    Net.coord._.STOPPING and
    (Net.NODE1._.IDLE or Net.NODE1._.NEEDS_RECAL) and
    (Net.NODE2._.IDLE or Net.NODE2._.NEEDS_RECAL);

// R4: something went wrong on at least one node
// also catches the case where only one node lost power (the other is fine)
requirement Net.coord._.c_node_unhealthy needs
    (Net.NODE1._.FAULT or Net.NODE1._.SETUP_ERROR) or
    (Net.NODE2._.FAULT or Net.NODE2._.SETUP_ERROR) or
    (Net.NODE1._.NO_POWER and not Net.NODE2._.NO_POWER) or
    (Net.NODE2._.NO_POWER and not Net.NODE1._.NO_POWER);

// R5: both nodes are back on their feet after a fault
requirement Net.coord._.c_all_healthy needs
    Net.coord._.RECOVERING and
    (Net.NODE1._.IDLE or Net.NODE1._.NEEDS_RECAL) and
    (Net.NODE2._.IDLE or Net.NODE2._.NEEDS_RECAL);

// R6: we ran out of retries, time to call the technician
requirement Net.coord._.c_limits_exceeded needs LIMIT_EXHAUSTED;

// R7: both nodes lost power at the same time (building outage)
requirement Net.coord._.c_all_power_lost needs
    Net.NODE1._.NO_POWER and Net.NODE2._.NO_POWER;

// R8: power is back on both nodes
requirement Net.coord._.c_power_restored needs
    Net.coord._.NO_POWER and
    not Net.NODE1._.NO_POWER and not Net.NODE2._.NO_POWER;


//=============================================================================
// Category 2: Phase gating  (R9-R18)
// Each command is only allowed during the right phase.
//=============================================================================

// R9-R10: calibrate only during CALIBRATING
requirement Net.NODE1._.c_calibrate needs Net.coord._.CALIBRATING;
requirement Net.NODE2._.c_calibrate needs Net.coord._.CALIBRATING;

// R11-R12: motors spin only during RUNNING
requirement Net.NODE1._.c_enable needs Net.coord._.RUNNING;
requirement Net.NODE2._.c_enable needs Net.coord._.RUNNING;

// R13-R14: stop is allowed during shutdown, recovery, or emergency
requirement Net.NODE1._.c_stop needs
    Net.coord._.STOPPING or Net.coord._.RECOVERING or Net.coord._.EMERGENCY;
requirement Net.NODE2._.c_stop needs
    Net.coord._.STOPPING or Net.coord._.RECOVERING or Net.coord._.EMERGENCY;

// R15-R16: recovery commands only during RECOVERING
```

```
requirement Net.NODE1._.c_recover needs Net.coord._.RECOVERING;
requirement Net.NODE2._.c_recover needs Net.coord._.RECOVERING;

// R17-R18: speed changes only while running
requirement Net.NODE1._.c_set_velocity needs Net.coord._.RUNNING;
requirement Net.NODE2._.c_set_velocity needs Net.coord._.RUNNING;


//============================================================================
// Category 3: Sequential calibration  (R19-R20)
// One at a time -- the PSU can't handle both calibrating together.
//============================================================================

// R19: NODE1 waits for NODE2 to finish calibrating
// R20: NODE2 waits for NODE1 to finish calibrating
requirement Net.NODE1._.c_calibrate needs not Net.NODE2._.CALIBRATING;
requirement Net.NODE2._.c_calibrate needs not Net.NODE1._.CALIBRATING;


//============================================================================
// Category 4: Sequential recovery  (R21)
// Supply fan goes first -- positive pressure matters more.
//============================================================================

// R21: NODE2 can't recover while NODE1 is still faulted
requirement Net.NODE2._.c_recover needs not Net.NODE1._.FAULT;


//============================================================================
// Category 5: Counter limits  (R22-R27)
// Stop retrying after too many failures. If it didn't work N times,
// it's not going to work the N+1th time either.
//============================================================================

// R22-R23: calibration attempts
requirement Net.NODE1._.c_calibrate needs Net.NODE1._.calibration_att_cnt < MAX_CAL;
requirement Net.NODE2._.c_calibrate needs Net.NODE2._.calibration_att_cnt < MAX_CAL;

// R24-R25: recovery attempts
requirement Net.NODE1._.c_recover needs Net.NODE1._.recovery_att_cnt < MAX_RECOVERY;
requirement Net.NODE2._.c_recover needs Net.NODE2._.recovery_att_cnt < MAX_RECOVERY;

// R26-R27: reboot attempts
requirement Net.NODE1._.c_reboot needs Net.NODE1._.reboot_cnt < MAX_REBOOT;
requirement Net.NODE2._.c_reboot needs Net.NODE2._.reboot_cnt < MAX_REBOOT;


//============================================================================
// Category 6: Both-or-nothing  (R28-R29)
// If one fan is unhealthy, the other one doesn't start.
// Running only one fan messes up building pressure.
//============================================================================

// R28: NODE1 won't start if NODE2 is in trouble
requirement Net.NODE1._.c_enable needs
    not Net.NODE2._.FAULT and
    not Net.NODE2._.NO_POWER and
    not Net.NODE2._.SETUP_ERROR and
    not Net.NODE2._.CAL_FAILED;

// R29: NODE2 won't start if NODE1 is in trouble
requirement Net.NODE2._.c_enable needs
    not Net.NODE1._.FAULT and
    not Net.NODE1._.NO_POWER and
    not Net.NODE1._.SETUP_ERROR and
    not Net.NODE1._.CAL_FAILED;


//============================================================================
// Category 7: Preventive safety  (R30-R33)
// Don't attempt risky operations when the system is already degraded.
//============================================================================
```

```
// R30-R31: don't start the fans if faults keep happening
requirement Net.NODE1._.c_enable needs
    Net.NODE1._.fault_cnt < MAX_FAULTS and
    Net.NODE2._.fault_cnt < MAX_FAULTS;
requirement Net.NODE2._.c_enable needs
    Net.NODE1._.fault_cnt < MAX_FAULTS and
    Net.NODE2._.fault_cnt < MAX_FAULTS;

// R32-R33: don't calibrate if the power supply keeps cycling
requirement Net.NODE1._.c_calibrate needs Net.NODE1._.power_cc_cnt < MAX_POWER;
requirement Net.NODE2._.c_calibrate needs Net.NODE2._.power_cc_cnt < MAX_POWER;

// We never use cal_reject in this setup -- failed cals just get retried.
// Killing it here so synthesis doesn't have to worry about it.
requirement Net.NODE1._.c_cal_reject needs false;
requirement Net.NODE2._.c_cal_reject needs false;


//============================================================================
// Category 8: Determinism and priority  (R34-R46)
// Without these, multiple events could fire in different orders and
// end up in different states. These pin down the execution order
// so the supervisor has confluence.
//============================================================================

// R34-R35: reboots only happen during recovery
requirement Net.NODE1._.c_reboot needs Net.coord._.RECOVERING;
requirement Net.NODE2._.c_reboot needs Net.coord._.RECOVERING;

// R36-R37: don't calibrate a node while the other one is broken
//          (unless we're already in recovery mode, then it's fine)
requirement Net.NODE1._.c_calibrate needs
    not (Net.NODE2._.FAULT or Net.NODE2._.SETUP_ERROR or Net.NODE2._.NO_POWER)
    or Net.coord._.RECOVERING;
requirement Net.NODE2._.c_calibrate needs
    not (Net.NODE1._.FAULT or Net.NODE1._.SETUP_ERROR or Net.NODE1._.NO_POWER)
    or Net.coord._.RECOVERING;

// R38-R39: no speed changes if the other node is down
requirement Net.NODE1._.c_set_velocity needs
    not (Net.NODE2._.FAULT or Net.NODE2._.SETUP_ERROR or Net.NODE2._.NO_POWER);
requirement Net.NODE2._.c_set_velocity needs
    not (Net.NODE1._.FAULT or Net.NODE1._.SETUP_ERROR or Net.NODE1._.NO_POWER);

// R40: NODE1 reboot needs clean conditions
requirement Net.NODE1._.c_reboot needs
    not LIMIT_EXHAUSTED and not Net.NODE2._.RUNNING and Net.NODE2._.IDLE;

// R41: NODE1 calibration blocked if any limit is hit
requirement Net.NODE1._.c_calibrate needs not LIMIT_EXHAUSTED;

// R42: NODE1 recovery blocked if any limit is hit
requirement Net.NODE1._.c_recover needs not LIMIT_EXHAUSTED;

// R43: NODE2 reboot needs clean conditions
requirement Net.NODE2._.c_reboot needs
    not LIMIT_EXHAUSTED and not Net.NODE1._.RUNNING and Net.NODE1._.IDLE;

// R44: NODE2 calibration blocked if any limit is hit
requirement Net.NODE2._.c_calibrate needs not LIMIT_EXHAUSTED;

// R45: NODE2 recovery blocked if any limit is hit
requirement Net.NODE2._.c_recover needs not LIMIT_EXHAUSTED;

// R46: NODE2 always calibrates after NODE1 (supply fan first)
requirement Net.NODE2._.c_calibrate needs
    not Net.NODE1._.NEEDS_RECAL and
    not Net.NODE1._.CAL_FAILED;
```