FCSL: A Compiled Specification Language and Hardware Abstraction Layer for Simulation-to-Flight Continuity

Massimo Di Pierro*
University of California, Santa Cruz, California 95064, USA

2026

We describe FCSL, the Flight Computer Specification Language of the HiroSim simulation stack, and its companion hardware abstraction layer (HAL). A flight computer is a *graph of small algorithms* scheduled by an explicit *state machine*: each algorithm owns one subsystem, exposes typed input/output ports as its entire interface, and is written in a restricted Python dialect compiled to native code with *no dynamic memory allocation* on the control path. The compiler wires the algorithms and orders their execution from the dataflow, and runs each only in the states that schedule it. The *same* compiled controller runs unchanged in software-in-the-loop, hardware-in-the-loop, and on hardware, with only the HAL changing, and a Python harness, `AlgTwin`, lets ordinary pytest tests step it from one algorithm to the whole machine.

Keywords: flight software; domain-specific language; code generation; state machines; synchronous dataflow; modular software architecture; unit and integration testing; hardware abstraction layer; software/hardware-in-the-loop; real-time systems.

I. Introduction

A flight computer reads sensors, runs control law and guidance logic at fixed rates, and writes actuators, while sequencing through mission phases (for a lander, say: boot, ascent, coast, descent, safe). Such software is held to a higher standard than ordinary application code: it must be *deterministic* (the same inputs produce the same outputs at the same times), *analyzable* (its timing and resource use can be bounded ahead of flight), and *validated before it flies*. The dominant industrial answer is a combination of restrictive coding standards, model-based design with automatic code generation, and extensive in-the-loop testing [1, 2, 7].

FCSL is our take on this problem, implemented within the HiroSim stack [24]. It is deliberately small. A controller is not a program in a general-purpose language; it is a *graph of algorithms* whose execution is governed by an explicit

*Lecturer, School of Engineering; massimo.dipierro@gmail.com (Corresponding Author).

finite state machine, with algorithm bodies written in a restricted subset of Python and compiled to native Nim at build time. The runtime contains no Python interpreter and performs no heap allocation on the control path. The companion HAL gives the controller a single, stable interface to sensors, actuators, the clock, and logging; one implementation of that interface drives the simulator, another drives real Linux hardware, so the identical compiled controller is exercised in simulation and then flown.

Determinism and analyzability are necessary but not sufficient: flight software is also built, reviewed, and maintained by teams, and a second argument of this paper is a software-engineering one. It is easier for engineers to reason about a flight computer as a set of *modular algorithms, each in charge of one subsystem* — a sensor frontend, a navigation filter, a guidance law, an actuator mixer — than as one monolithic control program. In FCSL the algorithm is exactly that unit, and its typed input/output ports are its entire interface. The integration work that is traditionally hand-written and error-prone does not exist as source code: from the declared port connections the compiler wires the algorithms together, type-checks every connection, infers the dependency graph, and orders execution automatically, and the state machine determines which algorithms run, at what rate, in each mission phase — an algorithm runs only in the states that schedule it. The same modularity extends to test. Beyond full closed-loop simulation with the flight computer in the loop, HiroSim's Python client library (`simclient`) provides an `AlgTwin` harness that compiles and spawns a flight computer from its specification, so a plain pytest test can exercise a single algorithm in isolation, a connected group of algorithms, or the entire machine, stepping it tick by tick with synthetic inputs (Section XI).

This paper first situates FCSL within the HiroSim stack (Section II), then explains the rationale behind each of these decisions (Sections III–VI), the compilation process and HAL (Sections VII–IX), worked examples (Section X), the test workflow from a single algorithm to the whole machine (Section XI), why the same structure makes the design tractable for AI-assisted authoring (Section XII), and how FCSL relates to existing flight- and robotics-software systems (Section XIII). Where useful we cite the HiroSim sources so the description stays grounded in the implementation rather than the aspiration.

II. Overview: FCSL in the HiroSim stack

HiroSim [24] is a modular stack for closed-loop physics simulation and flight-software development, aimed at robotics, aerospace, and drone applications in which one controller must be validated in simulation before it goes on a vehicle. Its components are:

- a **deterministic, event-driven simulator** (written in Nim) with a nanosecond-precision integer clock, a selection of integrators (Euler, RK2, RK4, AB4, Velocity Verlet, adaptive RK45), and dynamically loaded model plugins that can be authored in any language with a C ABI (Nim, C/C++, Rust);

- a **UDP telemetry pipeline** — a receiver that compacts the packet stream into columnar NumPy windows, an HTTP supervisor service, and a live browser **visualizer** with time-series plots and 3D vehicle views;
- the **onboard SDK**, whose compiler `fcslc` turns an FCSL specification (machine JSON plus restricted-Python algorithms) into native code — the subject of this paper;
- the **FCSL-HAL**, the hardware abstraction layer that lets the compiled flight computer run against either the simulator or real Linux hardware (Section VIII);
- a **Zenoh communication bus** connecting the simulator, flight computers, and hardware-in-the-loop rigs, with bridges to ROS 2 (Section IX); and
- a **Trick-style variable server** and a Python client library, `simclient`, for live commanding, inspection, and scripted test — including the `AlgTwin` harness of Section XI, which compiles and spawns a flight computer so pytest tests can step it with synthetic inputs, no simulator required.

The stack's organizing principle is the one this paper develops in detail: the flight computer is specified once, compiled once, and exercised at every rung of the validation ladder — in-process simulation, cross-process simulation over the flight transport, virtual hardware-in-the-loop, and real hardware — with only the seam behind the controller changing between rungs. The simulator provides the deterministic plant and the telemetry pipeline provides the observability; FCSL and the HAL, described next, provide the controller and the seam.

III. The same code in simulation and on hardware

The central design goal is *simulation-to-flight continuity*: the artifact validated in simulation is bit-for-bit the artifact that flies. This is the single most effective defense against a class of failures that has destroyed real vehicles — the controller behaves correctly against the model used to develop it but differently against a separately written "flight build," or against subtly different units, frames, or sign conventions on the real bus.

Model-based design names a ladder of fidelities for exactly this reason: *model-in-the-loop* (MIL), *software-in-the-loop* (SIL), *processor-in-the-loop* (PIL), and *hardware-in-the-loop* (HIL) [1]. Open-source autopilots expose the same ladder: ArduPilot and PX4 both ship a SITL ("software in the loop") build of the *same* flight code linked against a simulated vehicle, and a HITL mode that runs the real firmware on the real board against a simulator [14, 13]. FCSL adopts this principle as a first-class property rather than a build variant.

Concretely, FCSL recognizes four run modes (Figure 1), which differ only in *how the controller's inputs and outputs are connected*, never in the controller code:

1. **In-process SIL.** The compiled controller is loaded as a simulator plugin and its inputs/outputs are wired directly to simulator variables. Fully deterministic; the fastest way to iterate.
2. **SIL over the flight transport.** The *same binary that flies* runs as its own process and exchanges per-channel values with the simulator over the Zenoh bus.
3. **Virtual HIL.** The controller runs as its own process and talks to the simulator through the HAL's typed sensor/actuator channels, fed synthetically over the wire.
4. **On hardware.** The controller runs as its own process and the HAL is backed by real Linux device drivers.

Figure 1 also shows a fifth, hybrid arrangement: the experimental composite HAL of Section VIII sources selected channels from real hardware and the remainder from the simulator, so a vehicle can be brought up one sensor at a time against otherwise simulated I/O.

Modes 3 and 4 share the HAL interface, so moving from virtual HIL to flight is a one-line construction change at startup — the loop is the same. As the HiroSim HAL documentation puts it, "the same alg loop runs against real Linux hardware or the simulator — only the HAL implementation behind the API changes" [24]. We return to the mechanism in Section VIII.

IV. Simplicity and no runtime dynamic allocation

Flight software is judged by what it *cannot* do as much as by what it does. The most consequential restriction in FCSL is that the generated control path performs *no dynamic memory allocation* after initialization.

The rationale is standard in safety-critical practice and is worth stating plainly. Dynamic allocation introduces (i) *unbounded or hard-to-bound latency* — an allocation can trigger a free-list refill, a page fault, heap compaction, or, under a tracing collector, a pause whose timing is data dependent; (ii) *failure modes with no good in-flight recovery* — fragmentation and out-of-memory cannot be sensibly handled mid-control-loop; and (iii) *loss of analyzability* — worst-case execution time and worst-case memory are no longer static properties. For these reasons NASA/JPL's "Power of Ten" rules for safety-critical code forbid dynamic memory allocation after initialization (rule 3), and the MISRA C guidelines and the practices behind DO-178C avionics certification discourage or ban it outright [2, 3, 1]. The same instinct drives hard-real-time language design: synchronous languages compile to code with statically bounded memory and time [5, 6, 7].

FCSL enforces the rule structurally, not by convention. The language has only scalars, *fixed-size* arrays, and flat structs with scalar fields (Section VI); there are no growable collections, no strings on the control path, no objects with dynamic lifetime. The code generator therefore emits only stack scalars, fixed-size array literals, and statically shaped records; it never emits a growable sequence, a string constructor, a hash table, a closure, or a heap allocation on

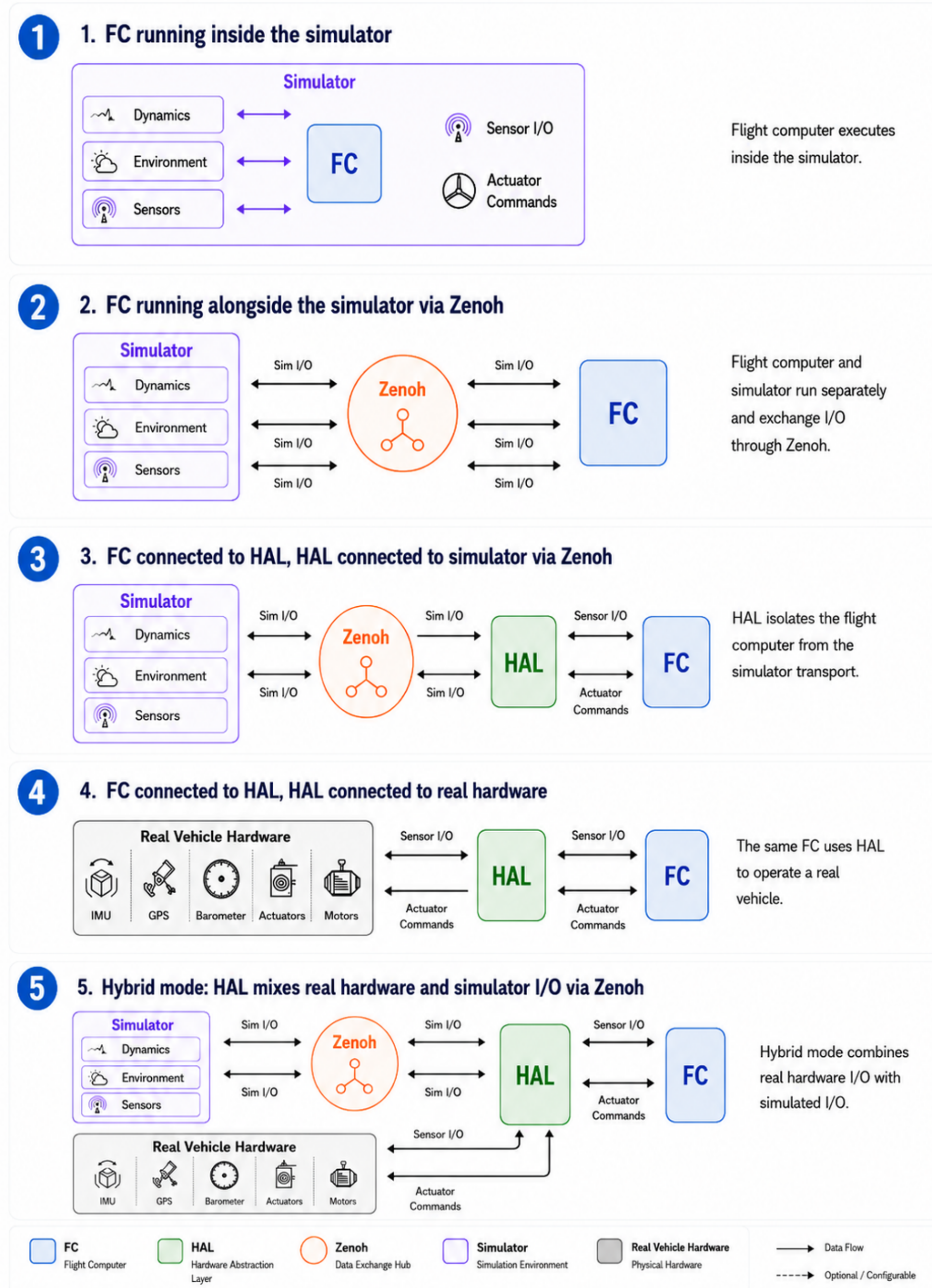

Fig. 1 The five FCSL modes of operation: in-process SIL, SIL over the flight transport, virtual HIL, flight, and a hybrid composite HAL that mixes real and simulated I/O.

the `execute` path. A unit test scans every generated source file for the tell-tale constructs (`seq[`, `newSeq`, `string(`, `Table[`, `new(`, `alloc()` and fails the build if any appear in control code [24]. State that must persist between ticks is declared up front in fixed-size fields and pre-allocated at initialization.

Simplicity is the same argument applied to control flow. Algorithm code cannot raise or catch exceptions; numerically singular operations (a singular matrix inverse, a non-positive-definite Cholesky factor) return NaN-filled results rather than raising, because there is no sensible place to catch an exception inside a 1 kHz loop. Floating point uses IEEE-754 defaults (quiet NaN/Inf, no trap), so a bad input degrades a channel rather than aborting the vehicle. The generated scheduler does, by default, wrap each instance's `execute` in a containment guard: an unhandled runtime fault (an out-of-bounds index, a divide-by-zero) marks that instance's `health` output `failed` and increments a per-instance fault counter instead of terminating the process — exceptions are contained at the instance boundary, never used as control flow (Section V). The bias, in the words of the HiroSim design notes, is to "prefer explicitness, static structure, and analyzability over convenience" [24].

V. A state machine over a graph of algorithms

A. The model

An FCSL *program* is one JSON file describing a finite state machine. Its elements are:

- **Algorithms** — the catalog of reusable computation blocks, each a single class (Section VI).
- **Instances** — named instantiations of algorithms with their parameters bound to constant values at compile time.
- **Connections** — explicit wiring of one instance's output to another's input, e.g. `sensors.theta` → `pilot.theta`. The data dependencies of the graph are *inferred* from this wiring, not declared separately.
- **States** — each state names a fixed set of enabled instances and a fixed rate (in Hz) for each, its *schedule*.
- **Transitions** — guarded edges `(from, request, to, priority)` between states, optionally gated by a health condition.

Nothing in this model is vehicle-specific or predefined. FCSL ships no built-in states, no fixed mission phases, and no canonical set of algorithms: the user declares however many states the mission needs, names them freely, and equips each with its own set of algorithm instances and rates. The four-state lander machine used as the running example in this paper is exactly that — an example, not a template the language imposes.

The execution model is a single global tick at `tick_hz`. In the active state, each enabled instance runs at its scheduled rate (every rate must divide `tick_hz` exactly, giving a predictable major cycle), and within a tick the

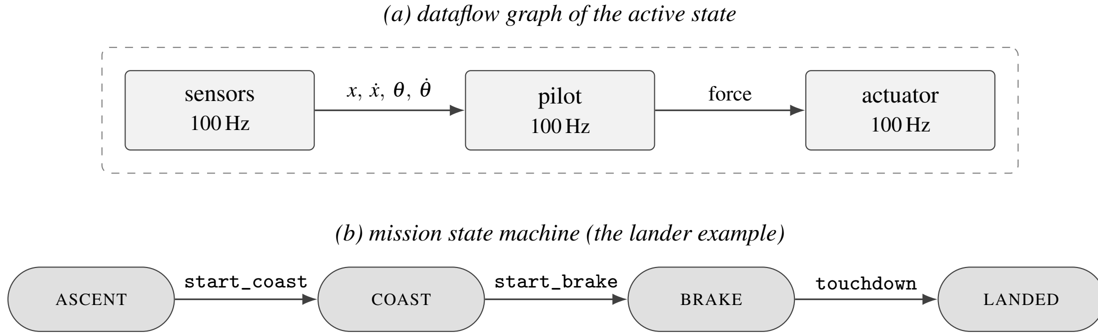

Fig. 2 An FCSL program is a graph of algorithms (a) scheduled by a state machine (b), shown here for the paper's four-state lander example.

enabled instances run *once, in topological order* of their connections. A cycle in a state's dataflow is rejected at compile time. The schedule is *static*: which blocks run, in what order, at what rate, is fixed per state and known before flight. This is the time-triggered, statically-scheduled discipline of the synchronous languages, not the dynamic, event-driven scheduling of a general RTOS task set [5].

Figure 2 shows a two-instance dataflow graph and a four-state mission machine (the HiroSim lander).

B. The software-engineering case: modular subsystems, automatic composition

Before the real-time properties, this model is a decomposition discipline. An engineer thinks of a flight computer as a set of subsystems — a sensor frontend, a navigation filter, a guidance law, a control law, an actuator mixer, a fault monitor — and FCSL's unit of code, the algorithm, is exactly one such subsystem. An algorithm's typed ports are its *entire* interface: it cannot reach into another block's state, call another block, or observe the machine except through its declared inputs. This buys the standard benefits of strong modularity — independent ownership, parallel development against port contracts, focused review, reuse of one algorithm across vehicles or as multiple instances with different parameters — with the boundaries enforced by the compiler rather than by convention.

The integration code that in a hand-written flight computer is the riskiest code — the main loop that calls subsystems in the right order, moves data between them, and remembers which ones run in which mode — does not exist as source code in FCSL. The engineer declares which outputs feed which inputs; the compiler wires the blocks together, type-checks every connection (including array shapes), infers the dependency graph, and orders execution topologically within each tick. The engineer declares which instances each state schedules and at what rate; the runtime runs an algorithm *only* in the states that schedule it, so phase logic stays out of algorithm bodies entirely. Adding a subsystem is adding an instance and its connections to the JSON; removing one is deleting them; the compiler re-derives the schedule and rejects at build time anything left dangling — a connection to a port that does not exist, a type or shape mismatch, a dataflow cycle, a transition request with no entry in the transition table. Integration errors therefore

surface as build errors in the one file that describes the architecture, rather than as flight anomalies traced back through glue code. That one file is also the review artifact: instances, wiring, per-state schedules, and the transition table, auditable in one place without reading any algorithm body.

The same modularity pays a third time in test: because a machine is just a list of instances and connections, any subset of algorithms is itself a valid machine, and Section XI shows how that turns the unit of design into the unit of test.

C. Why a state machine, and why this shape

Mission logic is naturally modal: a lander in its *ascent* phase and its *braking* phase obey different control laws, guards, and limits. Encoding modes as an explicit state machine — rather than as `if` ladders and boolean flags scattered across the control code — has a long pedigree. Harel's statecharts [4] established the visual, hierarchical state machine as the canonical way to specify reactive system behavior, and that model underlies MATLAB Stateflow and much of the model-based-design tooling used in aerospace. The synchronous languages Lustre, Esterel, and their industrial descendant SCADE compile mode logic and dataflow together into statically scheduled, statically bounded code that has been certified for use in commercial avionics [5, 6, 7]. The actor-oriented view of Ptolemy and the "coordination language" tradition make the same separation FCSL makes — between the *computation* inside a block and the *schedule* that composes blocks [10, 8]. Behavior trees are a more recent alternative for sequencing robot/agent behavior with similar goals [11]. At the autopilot level, ArduPilot's flight modes and PX4's commander are, in effect, hand-written state machines over a fixed set of controllers [13, 14].

FCSL makes three specific choices within this tradition:

1. **Algorithms never mutate machine state directly.** An algorithm requests a change by writing a reserved output, `transition_request`, with a transition name (or the empty string for "no request"). The runtime collects requests during the tick and applies *at most one* transition at the end, arbitrated by the priority declared in the JSON. This prevents hidden mode changes, makes arbitration explicit, and keeps the transition table auditable in one place rather than scattered through control code [24].

2. **Health is first class.** Every algorithm has an auto-injected `health` output (one of `nominal`, `degraded`, `failed`, or `stale`), and a transition may carry a *health guard*, so fault detection and the response to it (e.g. "on `failed` navigation, enter SAFE") live in the same declarative table as nominal sequencing. Two runtime monitors feed the same output. An optional per-instance worst-case-execution-time budget (`max_wcet_us`) times each `execute`; an overrun increments a counter and demotes the instance's `health` to `degraded`. And the default fault-containment guard (Section IV) sets `health` to `failed` on an unhandled runtime fault. The safety-net transitions in the table react identically whether the downgrade came from the algorithm's own logic, a timing

overrun, or a contained fault.

3. **The schedule is part of the state, not the code.** Because each state declares its instances and rates, the same algorithm can run at 100 Hz in one phase and be absent in another with no change to the algorithm itself — the phase *is* the state, not a variable inside a block [24].

VI. Why a Python dialect

Algorithm bodies are written in a restricted subset of Python. An algorithm is a class with four type-annotated dictionaries — `inputs`, `outputs`, `parameters`, `state` — and two methods, `start(self)` (run once per instance) and `execute(self)` (run each tick). Fields are read and written as `self.<name>`. Listing 1 shows the canonical shape.

Listing 1 An FCSL algorithm: a class with four typed dictionaries and two methods. `transition_request` **is the reserved output by which an algorithm asks the machine to change state.**

```
class Guidance:
    inputs = {"ax": "f64", "ay": "f64"}
    outputs = {"cmd_pitch": "f64", "cmd_yaw": "f64",
               "transition_request": "TransitionRequest"}
    parameters = {"gain": "f64"}
    state = {}

    def start(self):
        pass

    def execute(self):
        self.cmd_pitch = clamp(self.ax * self.gain, -1.0, 1.0)
        self.cmd_yaw = clamp(self.ay * self.gain, -1.0, 1.0)
        if abs(self.ax) > 50.0 or abs(self.ay) > 50.0:
            self.transition_request = "ENTER_SAFE"
        else:
            self.transition_request = ""
```

The choice of a Python *dialect* — as opposed to a graphical block language like Simulink/SCADE, or a hand-written C/C++ API — is pragmatic:

- **Familiar, expressive syntax.** Control engineers and students read and write Python. The body of a control law looks like the math, with no pointers, headers, or build scaffolding in the way. This is the same reason Python-like syntax appears in numerical tools that nonetheless compile to native code (Cython, Numba) and in MATLAB Function blocks.

- **A real parser for free.** FCSL parses the source with Python's own `ast` module, then lowers a *restricted* subset into its own intermediate representation; it does not run the Python and observe it. Anything outside the subset — imports, exceptions, dynamic objects, general strings/collections, arbitrary library calls — is rejected at compile time. The dialect is Python-shaped but is its own language with its own type system.

- **A narrow language is a safe language.** Restricting the surface is exactly what makes the no-allocation and static-schedule guarantees enforceable (Section IV). This mirrors the philosophy of domain-specific languages: a smaller language admits stronger analysis [12, 7].

The type system has scalar types (`bool`, `i32/i64`, `u32/u64`, `f32`, `f64`, plus `TransitionRequest` and `Health`) and *fixed-size* arrays, including nested ones: `f64[4]`, `u32[16]`, `f64[3][3]`. Every array dimension is a positive integer literal, so all shapes are known at compile time. User-defined *structs* are declared once at the top level of the machine JSON (a `structs` block mapping a type name, e.g. `Vec3`, to its fields) and may then appear in any of the four dictionaries, with struct parameters bound by inline JSON literals; in the current form struct fields must be scalars — no nested structs, no arrays inside structs — so every struct lowers to a flat, statically shaped record. The standard library available inside `execute` is deliberately aeronautical: elementwise math (`sin`, `cos`, `exp`, ...), an allocation-free linear-algebra kernel (`dot`, `norm`, `matmul`, `matvec`, `transpose`, `cross3`, `det`, `inv`, `solve`, `eye`, `diag`), scalar-first quaternions (`quat_mul`, `quat_rotate`), `cholesky`, and a composable discrete Kalman filter (`kf_predict_state`, `kf_predict_cov`, `kf_gain`, `kf_update_state`, `kf_update_cov`, and the numerically robust `kf_update_cov_joseph`). Dimensions are checked at compile time, so a shape mismatch is a build error, not a flight surprise.

Where the controller must call user-defined code — a device routine, an optimized kernel, an existing library — it declares an `@extern` stub outside the class with typed arguments. Only the stub's typed signature is checked at compile time; FCSL places *no* restrictions on the body of an extern function. The no-allocation discipline binds the algorithm dialect, not the user's own code: an extern is free to allocate, buffer, or call arbitrary libraries, with its real-time behavior the responsibility of its author (Section XIV). The default binding is a portable C ABI (the stub lowers to a call into a named C function in a named header), which is the only flavor that survives to a standalone embedded build; a Nim binding is also available for host-side helpers.

VII. Compilation

FCSL is *ahead-of-time compiled*, not interpreted. The compiler, `fcslc`, reads the program JSON and its algorithm `.py` files at *build* time and emits a single self-contained native (Nim) source file. In doing so it parses the restricted Python, resolves the wiring, type-checks every connection and array shape, topologically sorts each state (rejecting cycles and schedules whose rates do not divide `tick_hz`), validates that every `transition_request` literal appears in the transition table, and lowers `transition_request` to a generated enum so that transition handling does no string comparison and no allocation. The generated module exports a fixed contract: an enum of transition requests, an enum of machine states, a record per instance for its state/parameters/inputs/outputs, an `init` and `execute` proc per instance, and machine-level `initMachine` and `step` procedures. External code writes the source instance's outputs, calls `step`, and reads command outputs — a two-seam host interface.

The generated module is then compiled into a native shared library and loaded by the host (the simulator, or a standalone flight binary). The compiled artifact contains *no Python interpreter and never reads the JSON or the `.py` at runtime*; the JSON and Python are a build-time description only. A single, program-agnostic plugin target transpiles each FCSL program, generates an index that registers each compiled machine under its file-name stem, and links them into one shared object; adding a new flight computer is one more line in that list [24]. A direct consequence, noted prominently in the HiroSim developer docs, is that editing an algorithm, a port, a connection, or even a *parameter value* has no effect until the plugin is rebuilt and restaged — the running system reflects the compiled artifact, not the source on disk.

This "specify high, generate native, certify the generator" pattern is exactly how safety-critical control code is produced industrially: SCADE compiles a synchronous dataflow/state model to C qualified for DO-178; MATLAB Simulink with Embedded Coder generates C/C++ from block diagrams and Stateflow charts for production controllers [7, 4]. FCSL is a small instance of the same idea — the generated code is meant to be boring, readable, and obviously free of the constructs the domain forbids.

VIII. The hardware abstraction layer

The HAL is the seam that makes simulation-to-flight continuity (Section III) concrete. The controller is written against a single interface — read sensors, write actuators, ask the time, sleep until the next tick, log — and never names a device, a bus, or the simulator. Three deliberately separate APIs sit behind it: the *HAL* the controller calls, the *driver* interface that device backends implement, and the *wire* interface that simulator/HIL transports implement.

Two implementations satisfy the HAL. A *driver-backed* HAL composes per-sensor frontends over real device drivers; on Linux the default backends read inertial, magnetometer, and barometer data through the industrial-I/O (IIO) sysfs interface, drive motors and servos through PWM sysfs, read RC/joystick input through evdev, and read the battery through power-supply sysfs. A *wire-backed* HAL reads and writes the same logical channels as identifier-addressed frames over a transport, and, given a clock channel, advances on simulator time so a hardware-in-the-loop run is paced by the sim. A composite HAL can overlay selected real channels onto a wire-backed HAL for partial-hardware bring-up. Moving from simulation to flight is therefore a one-line change of which HAL is constructed at startup; the control loop is untouched. The per-sensor frontend structure intentionally mirrors ArduPilot's `AP_InertialSensor`, `AP_Baro`, `AP_Compass`, and friends — each owns several backends, tracks their health, and selects a primary — so the abstraction will be familiar to autopilot developers [13, 24].

For the lowest level, the language itself exposes a tiny, stable hardware-I/O ABI that an algorithm can call directly. These five builtins (Table 1) are backed by a board-support shim (`fcsl_hw.h` / `fcsl_hw.c`) that one replaces for a given board; in simulation they are stubbed.

Builtin	Signature	Purpose
`hw_read_gpio`	`u32 -> bool`	read a digital input pin
`hw_write_gpio`	`u32, bool -> ()`	write a digital output pin
`hw_read_adc`	`u32 -> u32`	read an analog-to-digital channel
`hw_read_u32`	`u32 -> u32`	read a board register/port
`hw_write_u32`	`u32, u32 -> ()`	write a board register/port

Table 1 The in-language hardware-I/O builtins: a small, stable ABI to a board-support shim. The richer typed sensor/actuator interface is the separate HAL library.

IX. Communication buses

FCSL separates three kinds of traffic, each with its own transport, rather than forcing everything through one bus.

Telemetry (one-way, observability). A standalone flight computer publishes its outputs over UDP using the same packet protocol the simulator uses: a configuration packet announces the set of channels (and is re-sent at exponentially spaced sequence numbers for loss resilience), followed by a data packet per tick. Array outputs are flattened row-major into scalar channels. This path is for recording and dashboards; it never closes a control loop.

Control (closed-loop) over Zenoh. The sensor→controller →actuator loop runs over the Zenoh publish/subscribe bus [16]. Subscriptions write named values into the controller's input instance before `step`; publications serialize selected outputs after it. Payloads are either raw little-endian scalars or CDR-wrapped (below). Endpoints are ordinary Zenoh locators (e.g. `tcp/127.0.0.1:7447`), and the controller can be a Zenoh *client* or *peer*. This is the transport over which the *same binary that flies* talks to the simulator (run mode 2).

Sensor/actuator HAL channels. In HIL and on hardware, the HAL moves typed channels — scalars, 3-vectors, quaternions, opaque byte blobs — addressed by a numeric 16-bit identifier (names live only in the configuration, never on the wire). A compact, deterministic little-endian codec serializes each channel as `(id, type, payload)`; sensors flow on one Zenoh key, actuators (plus an arm flag) on another, and a third key carries simulator time so the controller is paced by the sim. Bulk data (images, point clouds) is explicitly kept off this control path.

A serial-capable link layer. Below the channel codec sits an optional *port-multiplexed link layer* for serial-style transports (UDP today; UART, TCP, and Zenoh bindings are follow-ons). Each frame is versioned, little-endian, and protected by a CRC-16/CCITT-FALSE checksum; up to 16 ports are multiplexed onto one link, and each port carries a delivery policy chosen per stream: *latest* (the newest value overwrites; the reader sees only the most recent), *lossy-sequenced* (FIFO; sequence gaps are counted but not repaired), or *reliable* (go-back-N with cumulative acknowledgments and automatic fragmentation/reassembly of messages larger than one frame). Byte-stream transports COBS-encode each frame behind a zero delimiter. The layer follows the discipline of Section IV: every buffer is

a fixed-size array sized by compile-time constants, malformed input is dropped and counted rather than raised, and the multiplexer performs no I/O and owns no threads or timers — frames move and retransmissions fire only at explicit tick boundaries, so a hardware-in-the-loop run over a lossy serial link is reproducible under the simulator clock. On the simulator side a `serial_link` model bridges a link endpoint into the simulation and can inject seeded, deterministic impairments (drop and corruption percentages) for testing the policies under loss.

Interoperability. The Zenoh payloads can be wrapped in CDR, the DDS/ROS 2 wire format [17], so a HiroSim simulation or flight computer can appear to a ROS 2 graph as ordinary topics through the `zenoh-plugin-ros2dds` bridge; the simulator can even publish a `rosgraph_msgs/Clock` for `use_sim_time`. Finally, a Trick-style variable server [18] speaks a newline-delimited JSON command protocol (`get/set/eval/pause/resume/save/ restore`) shared by the simulator and the flight computer's optional command server, for interactive inspection and scripted test.

Compared with the autopilot world: where ArduPilot and PX4 standardize on MAVLink for command and telemetry, FCSL uses UDP for observability and Zenoh for the control loop and HAL channels; where ROS 2 uses DDS over a dynamically discovered graph of topics, FCSL uses a *static*, compile-time graph and can speak the same CDR encoding when bridged [13, 14, 15, 16].

X. Examples

A. A minimal two-block controller

The smallest complete program is a sensor block feeding a control block at a single rate with no mode changes. The machine JSON (Listing 2) names the algorithms, instantiates them with bound parameters, wires sensor outputs to controller inputs, and declares one state that runs both at `tick_hz`.

Listing 2 A minimal FCSL machine: two instances, one state, no transitions.

```json
{
  "tick_hz": 1000,
  "initial_state": "RUN",
  "algorithms": {
    "Sensors": { "source": "algorithms/sensors.py" },
    "Pilot": { "source": "algorithms/pilot.py" }
  },
  "instances": {
    "sensors": { "algorithm": "Sensors", "parameters": {} },
    "pilot": { "algorithm": "Pilot",
               "parameters": { "gain": 2.0, "limit": 1.0 } }
  },
  "connections": [
    { "from": "sensors.theta", "to": "pilot.theta" },
    { "from": "sensors.thetadot", "to": "pilot.thetadot" }
  ],
  "states": { "RUN": { "schedule": { "sensors": 1000, "pilot": 1000 } } },
  "transitions": []
}
```

Listing 3 **The `Pilot` algorithm referenced by Listing 2: a proportional-derivative law with a clamped output.**

```
class Pilot:
    inputs = {"theta": "f64", "thetadot": "f64"}
    outputs = {"force": "f64", "transition_request": "TransitionRequest"}
    parameters = {"gain": "f64", "limit": "f64"}
    state = {}

    def start(self):
        self.force = 0.0

    def execute(self):
        u = -self.gain * (self.theta + 0.1 * self.thetadot)
        self.force = clamp(u, -self.limit, self.limit)
        self.transition_request = ""
```

B. Lighting an LED on over-temperature

A common board-bring-up task is to assert a digital output when a monitored quantity crosses a threshold. Listing 4 reads CPU temperature through an `@extern` (on a Linux board this reads a thermal zone; in simulation the symbol is stubbed) and drives an LED through the GPIO builtin from Table 1. The threshold and pin are parameters, so the same algorithm serves any board by re-binding constants in the JSON. The state machine could additionally route a `transition_request` to a thermal-throttle state; here we keep it to the sensor-to-actuator path.

Listing 4 **An over-temperature monitor: light an LED when CPU temperature exceeds a limit. `cpu_temp_c` is an external C function; `hw_write_gpio` is the in-language GPIO builtin.**

```
from fcsl_builtins import extern, f64 # editor-only; fcslc ignores the import

@extern(header="board_sensors.h") # lang="c" (default): reads the thermal zone
def cpu_temp_c() -> f64: ...

class ThermalGuard:
    inputs = {}
    outputs = {"led_on": "bool", "temp_c": "f64",
               "transition_request": "TransitionRequest"}
    parameters = {"limit_c": "f64", "led_pin": "u32"}
    state = {}

    def start(self):
        self.led_on = False

    def execute(self):
        self.temp_c = cpu_temp_c()
        self.led_on = self.temp_c > self.limit_c
        hw_write_gpio(self.led_pin, self.led_on) # drive the LED pin
        self.transition_request = ""
```

With `limit_c` bound to `80.0` and `led_pin` to the board's LED line, the LED follows the over-temperature condition every tick. Because `temp_c` and `led_on` are declared outputs, both are visible on the telemetry stream and to the variable server with no extra code — the same introspection works identically in simulation and on hardware.

XI. Testing: from one algorithm to the whole machine

The run modes of Section III give system-level validation: closed-loop simulation with the flight computer in the loop, against a deterministic plant, up to hardware-in-the-loop. Day-to-day engineering also needs the lower rungs of the test pyramid — fast unit tests of one algorithm and integration tests of a subsystem chain — runnable from CI in seconds, with no simulator and no hardware. FCSL's modularity (Section B) makes those rungs cheap, and the `simclient` Python library provides the harness.

A. AlgTwin: stepping a compiled flight computer from Python

`AlgTwin.from_config(path)` takes an FCSL machine JSON, invokes the same compiler used for flight (`fcslc` plus the Nim backend), caches the resulting binary keyed by a content hash of the config and every referenced algorithm source, spawns it as a subprocess, and connects to its Trick-style command server. The test then drives the machine *synchronously*: `pause()` freezes the production loop so manual stepping cannot race it; `set(path, value)` writes any input, output, parameter-backed, or state field by its dotted path; `tick(n)` advances exactly n ticks; `get_value(path)` reads any field; `state()` and `set_state(name)` read and force the machine state; and `save()`/`restore()` checkpoint and roll back the full machine state. Listing 5 drives the lander flight computer of Figure 2 through its ASCENT→COAST transition with synthetic sensor values — a complete, runnable pytest-style test.

Listing 5 Stepping the compiled lander flight computer from Python with `AlgTwin`. The harness builds (and caches) the flight binary, spawns it, and exposes synchronous `set`/`tick`/`get_value` over the command protocol; the test injects synthetic sensor outputs and asserts on control outputs and state transitions.

```
from simclient import AlgTwin

with AlgTwin.from_config("examples/lander/fc/lander_control.json") as fc:
    fc.pause() # manual stepping from here on
    assert fc.state() == "ASCENT"

    # Synthetic sensors: low and slow -> the ascent law keeps climbing.
    fc.set("sensors.outputs.pos_z", 5.0)
    fc.set("sensors.outputs.vel_z", 4.0)
    fc.tick()
    assert fc.get_value("ascent.outputs.throttle") > 0.5

    # Cross the 40 m target altitude: ascent requests COAST ...
    fc.set("sensors.outputs.pos_z", 45.0)
    fc.tick()
    assert fc.get_value("ascent.outputs.transition_request") \
        == "tr_START_COAST"

    # ... and the machine consumes the request on the next tick.
    fc.tick()
    assert fc.state() == "COAST"
```

B. Isolation, grouping, and the full machine

Because a machine is just a JSON list of instances and connections (Section B), the natural unit-test fixture is a machine containing *only the algorithm under test*: one instance, one state that schedules it, no transitions. The test sets the instance's inputs, ticks, and asserts on its outputs — a plain Python function, no simulator, no hardware, no mocks. An integration test is the same pattern over a connected subgraph — say, sensor frontend → navigation → guidance — exercising the real compiled wiring under a synthetic sensor history. The full-machine config then tests sequencing: forcing phases with `set_state`, replaying input profiles, asserting that transitions fire when they should and that health guards trip on injected degradations. The unit of design is the unit of test at every level.

Two properties make this more than convenience. First, at every granularity the artifact under test is the *compiled native machine* — the same generated code, scheduler, and runtime that flies — not a Python re-interpretation of the algorithm, so there is no semantic gap between unit tests and flight even at the lowest rung of the pyramid. Second, the runtime's determinism (a fixed tick, a static schedule, no allocation, no hidden concurrency) makes every test exactly reproducible: `tick()` advances the machine and nothing else does. The content-hash build cache keeps the loop fast — the first run of a config compiles the binary in seconds, and subsequent runs reuse it, so these suites are cheap enough for CI and pre-commit hooks.

A companion `SimTwin` wraps the simulator behind the same synchronous API (`pause`/`step`/`set`/`get`), so plant-side tests — for example, asserting a sensor model's empirical noise statistics — are written in the same style; and the full closed-loop run modes of Section III remain the top of the pyramid. The result is one test vocabulary, in ordinary Python, spanning a single algorithm to a flown binary.

XII. The same design is AI-friendly

A substantial fraction of HiroSim — including parts of the FCSL compiler and several complete vehicles and their controllers — was authored in collaboration with a large language model (LLM). The observation that matters here is structural: the properties argued for on software-engineering grounds in Section B — locality, declarative composition, strong static contracts, determinism — are precisely the properties that make the design tractable for an AI agent to extend *correctly*. An LLM arrives with broad but shallow knowledge, has no persistent memory of the codebase between sessions, cannot run the system in its head, and pays a steep price for any ambiguity it must resolve by guessing; the framework's structure has to substitute for the situated knowledge a human contributor would accumulate.

Declarative composition bounds the task. Specifying a flight computer in FCSL means generating a JSON object that references existing, documented primitives, plus small, contract-bounded algorithm bodies. Both are constrained, well-specified tasks with a verifiable shape. The alternative — generating imperative code that mutates a shared object

graph, getting the locking, the update order, and the lifecycle right — is an open-ended task with many ways to be subtly wrong. By pushing composition into the declarative layer, FCSL converts most “add a subsystem” requests from program-synthesis problems into configuration problems, which current models handle far more reliably.

Locality keeps the context in a prompt. Because an algorithm’s typed ports are its entire interface (Section B), an agent asked to add or modify one needs to understand that algorithm’s contract and the names of the channels it touches — not the global control flow of the machine. The relevant context fits in a prompt. Non-local reasoning about interactions scattered across a codebase is exactly what exceeds an LLM’s working context and is hardest to verify after the fact.

Static contracts catch “plausible but wrong.” An LLM is good at producing code that is locally plausible; a tight static contract is what catches the cases where plausible is not correct, and it does so at the build step rather than three seconds into a flight. The type system is small; array dimensions are static, so a shape mismatch in a linear-algebra builtin is a compile error rather than a runtime surprise; transition requests lower to an enum, so a typo in a transition name fails the build; and the no-allocation property is enforced by a scan of the generated code (Section IV). The restriction list of Section VI is itself a feature: each forbidden construct is one fewer way for a generated controller to be non-deterministic or unbounded, and the compiler’s rejection message tells the agent precisely what to change.

Determinism makes verification cheap. A non-deterministic collaborator’s changes must be verified against a fixed baseline, and flaky verification is corrosive to an automated edit-test-review loop. Because runs are deterministic (a fixed tick, a static schedule, seeded noise in the simulator), an `AlgTwin` assertion of Section XI either holds every time or fails every time; the agent can run the suite after every change and read an unambiguous pass or fail. The content-hash build cache and a fast-compiling backend keep a compile step inside the agent’s correction loop cheap, which in aggregate is the difference between an agent that iterates freely toward a working controller and one throttled by its own toolchain.

The same experience also showed where machine authorship was most fragile: precisely where guarantees thinned to convention — the stale-artifact trap of Section VII (an edited source has no effect until the plugin is rebuilt), and silent NaN propagation on the numeric path. The lesson generalizes and is the same one safety-critical practice teaches: convert as many correctness obligations as possible from conventions into checks that fail loudly at build or test time.

XIII. Related systems and analogies

Autopilots: ArduPilot and PX4. FCSL is closest in spirit to the open-source autopilots. It shares their hardware-abstraction discipline (FCSL’s HAL deliberately mirrors ArduPilot’s `AP_HAL` and `AP_*` sensor frontends), their explicit

flight-mode state machines, and their "same code in SITL and on the board" methodology [13, 14]. It differs in being a *language and code generator* rather than a C++ framework: control logic is specified declaratively and compiled, the schedule is static and single-threaded by construction, and the no-allocation property is enforced by the generator rather than left to reviewer discipline.

Robotics middleware: ROS 2. ROS 2 composes nodes communicating over DDS topics with dynamic discovery and largely event-driven execution [15, 17]. FCSL's connections resemble topics, but the graph is *static* (fixed at compile time), the execution is a single deterministic tick rather than a set of concurrently scheduled callbacks, and there is no runtime discovery. FCSL can nonetheless interoperate with a ROS 2 graph by speaking CDR over a Zenoh-to-DDS bridge — notably, ROS 2 itself now offers a Zenoh middleware, so the transport choice is convergent.

Synchronous and model-based languages. The deepest lineage is the synchronous dataflow and statechart tradition: Lustre and Esterel and their certified industrial descendant SCADE, and the statechart formalism behind MATLAB Stateflow [5, 6, 7, 4]. These compile a combined dataflow-and-mode specification into statically scheduled, statically bounded code, and SCADE-generated code flies in certified avionics. Giotto and the time-triggered architecture contribute the static, rate-driven schedule; Ptolemy contributes the actor-oriented separation of computation from coordination [8, 9, 10]. FCSL is a small, Python-flavored member of this family; it is not itself a certified toolchain.

Certified flight software in aircraft and spacecraft. The principles FCSL adopts are the ones embodied, at vastly greater scale and rigor, in production flight computers. Commercial fly-by-wire systems are built to DO-178 software assurance and run on partitioned, time-deterministic avionics platforms (ARINC 653) on architectures such as the Boeing 777's triple-triple-redundant primary flight computer and Airbus's fly-by-wire suite, where determinism, bounded resources, and extensive in-the-loop validation are non-negotiable [1, 19, 20, 21]. Crewed spaceflight has relied on the same discipline since the Space Shuttle's Primary Avionics Software System, whose redundant design and exhaustive verification are a touchstone for the field [22]. Architecture description for such systems is itself standardized (AADL) [23]. FCSL does not claim this assurance level — it is a prototype — but its choices (static schedule, no dynamic allocation, explicit modes, generate-from-specification, simulation-to-flight continuity) are deliberately the *same choices*, in miniature, that the certified systems make.

XIV. Limitations

FCSL is an early prototype, and several aspects are explicitly subject to change. User-defined structs are currently flat: fields must be scalars, with no nested structs and no arrays inside structs. The WCET budgets of Section V measure and react at runtime but provide no static timing or worst-case execution-time *analysis*. The entire HAL layer — its

wire format, the simulator-side bus model, and the partial-hardware composite HAL — is experimental and expected to evolve, and the serial link layer of Section IX so far ships with in-process and UDP bindings only (UART and Zenoh bindings are planned). The no-allocation guarantee covers the FCSL-generated control path; `@extern` code can allocate internally, and the property is enforced by a source scan and by the narrowness of the language rather than by a formal proof about the final binary. None of these undercut the design thesis; they mark the boundary between what is implemented and what is intended.

XV. Conclusion

FCSL specifies a flight computer as a graph of small algorithms scheduled by an explicit state machine, written in a restricted Python dialect and compiled ahead of time to native code with no dynamic allocation on the control path. A single hardware-abstraction interface lets the identical compiled controller run in software-in-the-loop, in hardware-in-the-loop, and on real hardware, with only the implementation behind the interface changing. The same structure is an engineering workflow: each subsystem is an independent algorithm behind typed ports, composed by the compiler from declared connections rather than by hand-written glue, scheduled only in the states that need it, and testable — in isolation, in groups, or as the whole machine — from ordinary Python tests against the same compiled artifact that flies. These are not novel principles — they are the principles of synchronous languages, time-triggered scheduling, safety-critical coding standards, certified avionics, and plain modular software design — but FCSL packages them in a small, approachable form and wires them directly into a simulator so that the path from a Python control law to a flown binary is short and continuous.

Declaration on the use of AI

This paper was drafted, revised, and copy-edited with the assistance of Anthropic's Claude. The author reviewed and is responsible for all technical content.

References

[1] RTCA and EUROCAE, *DO-178C/ED-12C: Software Considerations in Airborne Systems and Equipment Certification*, RTCA, Washington, DC, 2011.

[2] Holzmann, G. J., "The Power of Ten: Rules for Developing Safety-Critical Code," *Computer*, Vol. 39, No. 6, 2006, pp. 95–99. https://doi.org/10.1109/MC.2006.212

[3] MISRA, *MISRA C:2012 — Guidelines for the Use of the C Language in Critical Systems*, HORIBA MIRA, Nuneaton, UK, 2013.

[4] Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, Vol. 8, No. 3, 1987, pp. 231–274. https://doi.org/10.1016/0167-6423(87)90035-9

[5] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D., "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, Vol. 79, No. 9, 1991, pp. 1305–1320. https://doi.org/10.1109/5.97300

[6] Berry, G., and Gonthier, G., "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," *Science of Computer Programming*, Vol. 19, No. 2, 1992, pp. 87–152. https://doi.org/10.1016/0167-6423(92)90005-V

[7] Colaço, J.-L., Pagano, B., and Pouzet, M., "SCADE 6: A Formal Language for Embedded Critical Software Development," *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, Institute of Electrical and Electronics Engineers, 2017, pp. 1–11. https://doi.org/10.1109/TASE.2017.8285623

[8] Henzinger, T. A., Horowitz, B., and Kirsch, C. M., "Giotto: A Time-Triggered Language for Embedded Programming," *Proceedings of the IEEE*, Vol. 91, No. 1, 2003, pp. 84–99. https://doi.org/10.1109/JPROC.2002.805825

[9] Kopetz, H., *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2nd ed., Springer, New York, 2011. https://doi.org/10.1007/978-1-4419-8237-7

[10] Lee, E. A., "The Problem with Threads," *Computer*, Vol. 39, No. 5, 2006, pp. 33–42. https://doi.org/10.1109/MC.2006.180

[11] Colledanchise, M., and Ögren, P., *Behavior Trees in Robotics and AI: An Introduction*, CRC Press, Boca Raton, FL, 2018. https://doi.org/10.1201/9780429489105

[12] Fowler, M., *Domain-Specific Languages*, Addison-Wesley, Boston, MA, 2010.

[13] ArduPilot Development Team, "ArduPilot Autopilot Suite," https://ardupilot.org [retrieved 12 June 2026].

[14] Meier, L., Honegger, D., and Pollefeys, M., "PX4: A Node-Based Multithreaded Open Source Robotics Framework for Deeply Embedded Platforms," *2015 IEEE International Conference on Robotics and Automation (ICRA)*, Institute of Electrical and Electronics Engineers, 2015, pp. 6235–6240. https://doi.org/10.1109/ICRA.2015.7140074

[15] Macenski, S., Foote, T., Gerkey, B., Lalancette, C., and Woodall, W., "Robot Operating System 2: Design, Architecture, and Uses in the Wild," *Science Robotics*, Vol. 7, No. 66, 2022, Paper eabm6074. https://doi.org/10.1126/scirobotics.abm6074

[16] Corsaro, A., Cominardi, L., Hécart, O., Baldoni, G., Enoch, J., Avital, P., Loudet, J., Guimarães, C., Ilyin, M., and Bannov, D., "Zenoh: Unifying Communication, Storage and Computation from the Cloud to the Microcontroller," *2023 26th Euromicro Conference on Digital System Design (DSD)*, Institute of Electrical and Electronics Engineers, 2023, pp. 422–428. https://doi.org/10.1109/DSD60849.2023.00065

[17] Object Management Group, *Data Distribution Service (DDS)*, Version 1.4, Document formal/2015-04-10, Object Management Group, Needham, MA, 2015.

[18] NASA Johnson Space Center, "Trick Simulation Environment," https://github.com/nasa/trick [retrieved 12 June 2026].

[19] ARINC, *ARINC Specification 653: Avionics Application Software Standard Interface*, Aeronautical Radio, Inc., Annapolis, MD, 2015.

[20] Yeh, Y. C., "Triple-Triple Redundant 777 Primary Flight Computer," *1996 IEEE Aerospace Applications Conference Proceedings*, Vol. 1, Institute of Electrical and Electronics Engineers, 1996, pp. 293–307. https://doi.org/10.1109/AERO.1996.495891

[21] Brière, D., and Traverse, P., "AIRBUS A320/A330/A340 Electrical Flight Controls: A Family of Fault-Tolerant Systems," *FTCS-23: The Twenty-Third International Symposium on Fault-Tolerant Computing*, Institute of Electrical and Electronics Engineers, 1993, pp. 616–623. https://doi.org/10.1109/FTCS.1993.627364

[22] Garman, J. R., "The 'Bug' Heard 'Round the World," *ACM SIGSOFT Software Engineering Notes*, Vol. 6, No. 5, 1981, pp. 3–10. https://doi.org/10.1145/1005928.1005929

[23] SAE International, *Architecture Analysis and Design Language (AADL)*, Standard AS5506C, SAE International, Warrendale, PA, 2017.

[24] HiroSim Simulation Stack, "FCSL, FCSL-HAL, and `simclient` Documentation," https://hirosim.com [retrieved 12 June 2026].