

AEROLANG: A DIMENSIONALLY-AWARE DOMAIN-SPECIFIC LANGUAGE FOR REAL-TIME FLIGHT CONTROL WITH INTEGRATED MORPHING WING PHYSICS

Aaryan Chaulagain  0009-0007-0460-7116

Independent Researcher, Nepal

ABSTRACT

Safety-critical flight software for autonomous aerial vehicles must respect the physics of flight and the constraints of embedded hardware. Existing practice relies on general-purpose languages where physical units are implicit and dimensional consistency is enforced only by reviews, unit tests, and flight campaigns—leaving systems vulnerable to catastrophic unit mismatches such as the Mars Climate Orbiter loss. This paper introduces AeroLang, a domain-specific language for flight control that makes physical units and real-time constraints first-class language citizens. AeroLang programs combine explicit SI unit annotations, declarative real-time loop periods, and an `@EMBEDDED` mode that statically forbids unbounded allocation. A complete compiler pipeline is presented: an LALR(1) parser, a semantic type checker performing compile-time dimensional analysis, and a Python code generator producing zero-overhead control components. The language is demonstrated through a morphing-wing UAV controller coupled to a free-vortex-wake aerodynamic model and an Euler-Bernoulli aeroelastic model. All dimensional inconsistencies are caught at compile time, the generated controller executes with negligible overhead, and the closed-loop system reproduces morphing-wing efficiency gains of approximately 11.3% in lift-to-drag ratio, consistent with published literature.

Keywords

dimensional analysis, morphing wing, real-time systems, domain-specific language, flight control software, compiler design.

Aaryan Chaulagain is an independent researcher and a recent graduate of the Cambridge International Examinations (CIE) A Level curriculum. A 2025 Bronze Medalist at the Asian Science Camp, his work bridges the gap between theoretical astrophysics, high-performance computing, and aerospace engineering. His research in particle physics has been conducted under the mentorship of Nobel Laureate Professor Takaaki Kajita, focusing on the development of three-flavor neutrino oscillation calculators. Aaryan is the creator of AeroLang, a domain-specific language (DSL) and simulation engine designed for real-time aerospace flight control. Beyond his technical contributions, he is the founder of ScholarPath and serves as the Secretary-General of a 12-country research collective, leading initiatives to democratize access to advanced scientific resources for underrepresented students globally.

* Corresponding Author. Email address: chamlagainaaryan319@gmail.com

Received 28.03.2026 • Revised 12.05.2026 • Accepted 12.05.2026

© 2026 Author[s]. Article category: research article



1. INTRODUCTION

Modern flight controllers for unmanned aerial vehicles (UAVs) routinely blend rigid-body dynamics, aeroelastic effects, actuator constraints, and sensor fusion into a single real-time software stack. The engineering pressure to iterate rapidly on such controllers – frequently in high-level, interpreted programming languages – sits in fundamental tension with the requirement that every arithmetic operation on physical quantities should remain dimensionally correct and that every control loop should meet its deadline. In practice, contemporary flight-control software is still predominantly written in general-purpose languages where physical units are at best encoded in variable names or comments. Dimensional analysis lives in the engineer’s head, in review checklists, or in separate validation scripts [1].

History demonstrates unambiguously that this practice is insufficient. The Mars Climate Orbiter was lost in 1999 because ground software produced angular momentum values in imperial units (pound-force seconds) while the spacecraft’s flight system expected SI units (newton-seconds). The error persisted undetected through design, implementation, testing, and nine months of interplanetary cruise, ultimately costing approximately 327 million USD [2]. Dimensional mismatches are not merely academic curiosities; they are a distinct, recurring failure mode in safety-critical systems.

The increasing complexity of modern avionics necessitates a shift toward more rigorous software engineering paradigms. Recent literature highlights the growing adoption of Model-Based Systems Engineering (MBSE) and Domain-Specific Languages (DSLs) to manage this complexity. For instance, the Future Airborne Capability Environment (FACE™) technical standard emphasises modularity and portability through well-defined data models [3]. Similarly, ARINC 653 provides robust partitioning mechanisms for Integrated Modular Avionics (IMA) architectures [4], although such standards primarily operate at the architectural level and do not inherently enforce physical consistency within embedded control logic.

Formal methods have increasingly been explored for safety-critical aerospace systems, particularly in the verification of autonomous decision systems, flight-envelope protection, and cyber-physical coordination. Existing approaches demonstrate the utility of SMT-based verification, model checking, and synchronous reactive semantics for improving software assurance in avionics environments. However, many of these approaches remain separated from the physical modelling layer, requiring manual dimensional validation and external coordination between engineering domains.

Despite these advances, a gap remains for a unified framework capable of bridging high-level physical modelling, dimensional correctness, cyber-physical coordination, and low-level embedded constraints within a single aerospace-oriented abstraction layer. AeroLang – the domain-specific language (DSL) presented herein – addresses this gap by elevating dimensional universality, aerospace semantics, and real-time

execution constraints to first-class language constructs, enabling physically rigorous software representation for flight-control-oriented systems.

The necessity for such integrative abstractions is further underscored by the evolution of aerospace materials and structural systems. As noted by Setlak et al. [5], the practical use of advanced composite materials in military aircraft introduces significant modelling challenges due to anisotropic behaviour, structural coupling, and increasingly complex adaptive geometries. The transition from conventional metallic airframes toward multi-functional composite structures – including morphing-wing configurations – requires corresponding advances in computational abstraction and control-system integration. AeroLang provides extensibility mechanisms capable of representing these complex aerospace-system interactions within a unified physics-aware control-oriented framework, enabling tighter integration between structural modelling, dimensional consistency, and adaptive flight-control semantics.

The problem further intensifies in emerging domains. Morphing-wing aircraft actively reconfigure their aerodynamic surface geometry in flight to optimise performance across varying conditions. The control software for such vehicles must simultaneously reason about translational and rotational kinematics, wing-shape parameters expressed as span fractions and camber ratios, aerodynamic force and moment coefficients, and structural deflections in bending and torsion. Recent experimental and computational studies confirm that the aerodynamic benefits of morphing are significant – lift-to-drag ratio improvements of 10–20% and in-flight energy savings of up to 11.5% have been demonstrated [6–8] – but these gains can only be realised if the control software correctly implements the physics.

This paper takes the position that the language in which flight-control logic is written should structurally embody the physics it manipulates. AeroLang, a domain-specific language that lifts SI units, dimensional analysis, and real-time loop specifications into the core of the language itself, is presented. The central insight, drawn from the programming-language-theory literature [9,10], is that physical dimensions can be modelled as a type system: quantities of different dimensions are different types, and the type checker enforces dimensional consistency statically, before any code executes.

Programs written in AeroLang describe flight controllers in terms of unit-annotated state variables, dimensionally-typed procedure parameters, and real-time loop declarations with explicit periods. A dedicated compiler enforces that all expressions are dimensionally consistent, that `@EMBEDDED`-annotated components respect basic constraints on memory allocation and real-time predictability, and then generates an executable artifact. In the present work, that artifact is a Python module that plugs directly into an existing morphing-wing UAV simulation framework, enabling rapid experimentation without sacrificing safety.

The primary contributions of this paper are as follows. First, the design of AeroLang is presented, including its formal syntax defined by a context-free

LALR(1) grammar, its dimensional type system grounded in the theory of quantity calculus [11–13], and its `@EMBEDDED` discipline for real-time memory safety. Second, a complete compiler implementation is presented: a lexer, parser, abstract syntax tree (AST), type checker, and Python code generator. Third, AeroLang is demonstrated through a realistic case study in which a morphing-wing UAV controller is authored in AeroLang, compiled, and integrated with a physics simulation coupling a free-vortex-wake aerodynamic model with an Euler-Bernoulli aeroelastic model. Fourth, the AeroLang compiler and runtime library are released as open-source software archived at Zenodo [14]. Finally, a roadmap toward formal verification via SMT solvers and towards a C code generator for certifiable embedded targets is outlined.

2. BACKGROUND AND RELATED WORK

The development of the AeroLang programming language brings together insights from five primary research areas: domain-specific languages in aerospace, language-integrated dimensional analysis, morphing-wing aerodynamics and control, real-time and safety-critical software engineering, and compiler technology and type theory.

2.1 Domain-Specific Languages in Aerospace

DSLs have a rich history in aerospace for improving the safety, correctness, and abstraction level of mission-critical software.

The Architecture Analysis and Design Language (AADL) provides a formal notation for describing embedded real-time system architectures, enabling the analysis of timing properties and resource consumption before deployment [15]. AADL targets architectural concerns – partitioning, scheduling, communication – rather than the continuous-time physics of a control law. Similarly, NASA's SCR method and related tabular specification languages address the formal specification of mode-logic and mode-transition conditions in flight management systems [16], but do not address dimensional correctness of the underlying arithmetic.

The Lustre and SCADE languages, in turn, take a synchronous dataflow approach to reactive and real-time control software [17,18]. These languages have strong industrial adoption in avionics (Airbus uses SCADE for flight-control software generation) and provide strong formal guarantees about determinism and schedulability. However, their data types remain physically uninterpreted; a signal is typed as a floating-point number, not as a quantity in meters per second.

Domain-specific languages in robotics, a domain closely related to UAV control, include the Robot Operating System (ROS) message definition language and the Orocos Toolchain's Real-Time Toolkit specification language. These provide useful structuring mechanisms but similarly lack physical dimension tracking [19].

AeroLang takes a complementary approach: it focuses on the physical interpretation of quantities rather than on synchronous scheduling, and the two approaches could in principle be combined in a future multi-paradigm extension.

2.2 Language-Integrated Dimensional Analysis

The idea of encoding physical dimensions in a type system dates back at least to the early work of Kennedy [20], who showed that a simple type system based on rational exponents over base dimensions can be incorporated into a typed functional language with no runtime cost. Kennedy's subsequent implementation of Units of Measure in F# [9,21] remains the most mature industrially-relevant example of compile-time dimensional analysis. The F# system represents a dimension as a product of base dimensions raised to rational exponents; unit information is erased after type checking, yielding the same runtime representation as undecorated floating-point numbers. AeroLang adopts this same design philosophy.

Alternative approaches enforce units at runtime. Python's Pint library [22] and the PhysicalQuantities library both track units as metadata on numeric objects, catching mismatches during execution. This approach is valuable for interactive scientific computing and post-processing but is unsuitable for real-time control loops where every operation carries overhead. AeroLang requires explicit annotations, accepting the cost in programmer effort in exchange for a formal soundness guarantee.

2.3 Morphing Wing Aerodynamics and Control

Morphing-wing aircraft exploit shape-changing mechanisms – variable sweep, span extension, camber adjustment, or twist – to adapt their aerodynamic configuration to the demands of different flight phases [23,24]. The aerodynamic benefits of morphing are well-established. Ninian [7] demonstrated that a variable-camber wing achieved a 17% improvement in lift-to-drag ratio over a fixed-geometry baseline. Wei et al. [8] analysed the effects of wing morphing on stability and energy harvesting in small UAVs, finding that morphing significantly expands the viable flight envelope. In a landmark study, Jeger et al. [6] demonstrated autonomous insect-inspired flight in which both wing and tail morphing were exploited simultaneously, achieving approximately 11.5% in-flight energy savings through Bayesian optimisation over the morphing state space.

The aerodynamic modelling of morphing wings presents particular challenges. Panel methods such as the vortex lattice method (VLM) are computationally tractable in real time and capture the dominant effects of planform changes on lift and induced drag [25]. Free-vortex wake methods extend panel methods by tracking the rollup and convection of the shed wake, capturing unsteady aerodynamic effects relevant to gust response and the transition between morphing states [26]. The Biot-Savart law, which relates vortex filament strength and geometry to the induced velocity field, is the foundation of both approaches.

Aeroelastic coupling – the interaction between aerodynamic loads and structural deformation – is critical for accurate modelling of flexible morphing wings [27,28]. Even in relatively stiff configurations, aerodynamic loads induce spanwise bending and chordwise twist that alter the effective angle of attack and thus the aerodynamic load distribution, creating a feedback loop. The Euler-Bernoulli beam theory provides a computationally efficient structural model capturing bending and torsion of slender wing sections [29].

2.4 Real-Time and Safety-Critical Software Engineering

Software for flight-critical systems is governed by standards such as DO-178C [30], which specifies software development assurance levels (DALs) A through E based on the consequences of failure. Level A (catastrophic failure condition) demands the most rigorous development and verification process, including structural coverage analysis at the Modified Condition/Decision Coverage (MC/DC) level. A C code generator for AeroLang would need to produce code amenable to DO-178C Level A certification, which motivates the @EMBEDDED discipline's prohibition of dynamic memory allocation – a requirement shared by coding standards such as MISRA C:2012 [31] and CERT C [32].

Worst-case execution time (WCET) analysis is a complementary requirement for hard real-time systems. WCET tools such as AbsInt aiT analyse binary code to derive tight upper bounds on execution time under all possible inputs and cache states [33]. AeroLang's code generator is designed to emit structurally simple Python (and in future, C) that facilitates WCET analysis by avoiding recursion, complex data structures, and dynamic dispatch.

2.5 Compiler Technology and Type Theory

The AeroLang compiler is built on classical compiler construction foundations. LALR(1) parsing, employed in AeroLang's front end, offers a good balance between generality and parsing speed; the parser generator infrastructure is provided by the Lark library for Python [34]. The theory of type systems [10,35] provides the formal foundation for the dimensional type checker. The Hindley-Milner type inference algorithm [35,36] offers a possible path toward inferring units for unannotated expressions, though AeroLang currently requires explicit annotations for clarity.

Formal verification of software properties beyond type safety requires more powerful tools. SMT solvers such as Z3 [37] and CVC5 [38] can discharge proof obligations about program correctness automatically, given a suitable encoding of the program semantics. The generation of SMT verification conditions from AeroLang programs is a primary direction of the AeroLang roadmap.

3. THEORETICAL FOUNDATIONS

3.1 The Theory of Dimensional Analysis

The modern theory of dimensional analysis originates with Fourier [11], who introduced the principle of dimensional homogeneity: a physically valid equation must be dimensionally consistent, meaning that both sides must have the same physical dimension. Maxwell [12] systematised this in his treatment of electromagnetism, and Buckingham [39] formulated the Π theorem, which provides the most general statement of how dimensional analysis constrains the form of physical laws.

Formally, let $B = \{L, M, T, I, \Theta, N, J\}$ denote the set of seven SI base dimensions (length, mass, time, electric current, thermodynamic temperature, amount of substance, and luminous intensity). Any physical dimension D can be expressed as a product of powers of base dimensions:

$$D = L^a \cdot M^b \cdot T^c \cdot I^d \cdot \Theta^e \cdot N^f \cdot J^g, \text{ where } a, b, c, d, e, f, g \in \mathbb{Z} \quad (1)$$

The set of all physical dimensions forms a free abelian group under multiplication, with the dimensionless quantity 1 as the identity element. This algebraic structure is precisely what AeroLang's unit type system formalises: dimensions are elements of $\mathbb{Z}^7$, addition of exponent vectors corresponds to multiplication of dimensions, and negation of the exponent vector corresponds to dimensional inversion.

3.2 SI Unit Algebra in AeroLang

AeroLang's unit system is built upon the seven base SI units $\{m, kg, s, A, K, mol, cd\}$ corresponding to the seven base dimensions. Every unit U in AeroLang is represented internally as an immutable seven-tuple of integer exponents $(p_m, p_{kg}, p_s, p_A, p_K, p_{mol}, p_{cd})$. For example:

```
Unit(N)    = ( 1,  1, -2,  0,  0,  0,  0)  -- kg · m · s-2
Unit(Pa)   = (-1,  1, -2,  0,  0,  0,  0)  -- kg · m-1 · s-2
Unit(J)    = ( 2,  1, -2,  0,  0,  0,  0)  -- kg · m2 · s-2
Unit(rad)  = ( 0,  0,  0,  0,  0,  0,  0)  -- dimensionless
```

The multiplication and division operators on units are defined as componentwise addition and subtraction of the exponent tuples:

$$U_1 \times U_2 := U_1 + U_2 \text{ (component-wise)} \quad (2)$$

$$U_1 / U_2 := U_1 - U_2 \text{ (component-wise)} \quad (3)$$

For addition and subtraction, AeroLang enforces the type rule that both operands must have identical unit tuples. Formally, the typing rules are:

T-ADD: If $\Gamma \vdash e_1: F32[u]$ and $\Gamma \vdash e_2: F32[u]$, then $\Gamma \vdash e_1 + e_2: F32[u]$ (4)

T-MUL: If $\Gamma \vdash e_1: F32[u_1]$ and $\Gamma \vdash e_2: F32[u_2]$, then $\Gamma \vdash e_1 \times e_2: F32[u_1 \times u_2]$ (5)

T-DIV: If $\Gamma \vdash e_1: F32[u_1]$ and $\Gamma \vdash e_2: F32[u_2]$, then $\Gamma \vdash e_1/e_2: F32[u_1/u_2]$ (6)

These rules guarantee what is termed dimensional soundness: any well-typed AeroLang expression, when evaluated, produces a quantity of the type and unit prescribed by its type derivation.

3.3 Real-Time Systems Theory

A real-time system is one in which the correctness of the system depends not only on the logical result of computation but also on the time at which the result is produced [40]. AeroLang addresses the schedulability dimension of real-time correctness through its LOOP construct. A LOOP declaration specifies a task with a fixed period T and an implicit deadline equal to the period (implicit-deadline periodic task in Liu and Layland's [41] terminology). The feasibility condition for a set of n such tasks on a single processor, under the Rate Monotonic Scheduling (RMS) algorithm, is given by the utilisation bound:

$$\sum(C_i / T_i) \leq n(2^{(1/n)} - 1) \quad (7)$$

where C_i is the worst-case execution time and T_i is the period of the i -th task. AeroLang does not currently perform schedulability analysis, but the structured form of the generated code – free of dynamic dispatch and recursive calls – is designed to facilitate WCET analysis by external tools.

4. AEROLANG LANGUAGE DESIGN

4.1 Design Principles

The design of AeroLang is guided by four principles. First, dimensional safety: every numeric quantity carries its physical unit as part of its type, and the compiler guarantees that all operations are dimensionally consistent. Second, zero runtime overhead: unit information is erased after type checking, so the generated code is numerically identical to hand-written, unit-free code. Third, embedded suitability: the @EMBEDDED annotation activates a stricter compilation mode that statically forbids dynamic memory allocation, making the generated code amenable to WCET

analysis and certification. Fourth, human readability: the syntax is designed to be self-documenting, so that a physical quantity declaration like `ALTITUDE: F32@M` reads naturally as “altitude is a 32-bit float in meters”.

4.2 Syntax and Grammar

AeroLang programs are organised into named modules. Within a module, a programmer declares unit aliases, component types, and component instantiations. The core grammar in Extended Backus-Naur Form (EBNF) is as follows:

```

program      ::= MODULE module_id declarations+ ;
declarations ::= unit_def | component_def ;
unit_def     ::= UNIT ident "=" unit_expr ;
unit_expr    ::= base_unit (("*" | "/" ) base_unit)* ("^" integer)?
;
base_unit    ::= "M" | "KG" | "S" | "A" | "K" | "MOL" | "CD" |
ident ;
component_def ::= annotation? COMPONENT ident "(" params ")" "{"
body "}" ;
annotation   ::= "@EMBEDDED" ;
params       ::= (param ("," param)*)? ;
param        ::= ident ":" type "@" unit_expr ;
body         ::= state_block? proc_def* loop_def* ;
state_block  ::= STATE "{" state_var* "}" ;
state_var    ::= ident ":" type "@" unit_expr "=" expr ";" ;
proc_def     ::= PROC ident "(" params ")" "{" stmt* "}" ;
loop_def     ::= LOOP ident "(" PERIOD ":" literal "@" unit_expr
")"
              "{" stmt* "}" ;
stmt         ::= let_stmt | assign_stmt | call_stmt | if_stmt ;
let_stmt     ::= LET ident ":" type "@" unit_expr "=" expr ";" ;
assign_stmt  ::= ident "=" expr ";" ;
call_stmt    ::= CALL ident "(" (expr ("," expr)*)? ")" ";" ;
expr         ::= expr ("+" | "-" | "*" | "/" | "^") expr
              | literal "@" unit_expr | ident | "(" expr ")" ;
type        ::= "F32" | "F64" | "I32" | "BOOL" | "POOL" "<" type
", "
              integer ">" ;

```

A complete example implementing a proportional-integral pitch controller is given in Code Listing 1. This example illustrates module declaration, user-defined unit aliases, component parameters with explicit unit types, typed state variables, a procedure with dimensional arithmetic, and a real-time loop specification.

```
// FILE: PITCH_CONTROL.AERO
MODULE FLIGHT.CONTROL

UNIT DEG = 0.0174533 RAD    // User-defined unit: degrees
UNIT N   = KG*M/S^2        // Derived SI: Newtons
UNIT NM  = N*M             // Derived SI: Newton-metres

@EMBEDDED
COMPONENT PITCHCONTROLLER(KP: F32@1/S, KI: F32@1/S^2) {
  STATE {
    ANGLE:          F32@RAD    = 0.0;
    RATE:           F32@RAD/S  = 0.0;
    ERROR_INTEGRAL: F32@RAD*S  = 0.0;
  }
  PROC UPDATE(DT: F32@S, TARGET: F32@RAD) {
    LET ERROR:      F32@RAD    = TARGET - ANGLE;
    LET P_TERM:     F32@RAD/S  = KP * ERROR;
    ERROR_INTEGRAL = ERROR_INTEGRAL + ERROR * DT;
    LET I_TERM:     F32@RAD/S  = KI * ERROR_INTEGRAL;
    LET TORQUE:     F32@NM     = (P_TERM + I_TERM) / 1@1/S / 1@KG*M*M;
    RATE = RATE + TORQUE * DT / 1@KG*M*M;
    ANGLE = ANGLE + RATE * DT;
  }
  LOOP CTRL(PERIOD: 0.01@S) { // 100 Hz real-time loop
    CALL UPDATE(0.01@S, 0.1@RAD);
  }
}
```

Code Listing 1. A proportional-integral pitch controller in AeroLang.

4.3 Embedded-Safe Mode

The `@EMBEDDED` annotation activates a stricter compilation discipline designed for real-time, resource-constrained hardware. In embedded mode, the type checker enforces three additional constraints beyond dimensional consistency. First, no dynamic memory allocation is permitted: every data structure must have a statically known, bounded size declared as `POOL<T, N>` types. Second, no recursive procedure calls are permitted, ensuring that the call stack has a statically bounded depth – a requirement for WCET analysis [33]. Third, loop periods must be positive compile-time constants. These constraints are directly motivated by the requirements of DO-178C [30] and MISRA C:2012 [31].

5. COMPILER ARCHITECTURE AND IMPLEMENTATION

5.1 Pipeline Overview

The AeroLang compiler is a five-stage pipeline that transforms high-level, physically-annotated source code into executable Python. The pipeline is implemented in Python and totals approximately 1,200 lines of source code. The compilation process proceeds through: (1) lexical analysis, (2) LALR(1) parsing via the Lark library [34],

(3) AST construction using Python dataclasses, (4) semantic analysis (type checking), and (5) code generation with unit erasure.

5.2 Type Checking Algorithm

The type checker performs a single-pass traversal of the AST, maintaining a symbol table (environment) Γ that maps identifiers to their declared types and units. The algorithm is formalised as Algorithm 1.

```

Algorithm 1: TypeCheckModule(Module M)
Input:  Module AST M
Output: Unit-annotated AST, or CompileError

1.  $\Gamma \leftarrow$  empty symbol table
2. For each UnitDef U in M.unit_defs:
3.   resolve unit expression  $\bar{U}.expr$  in base SI units
4.    $\Gamma.add\_unit(U.name, resolved\_exponents)$ 
5. For each ComponentDef C in M.components:
6.   if C.is_embedded: enable_strict_mode()
7.    $\Gamma\_local \leftarrow \Gamma.extend(C.params)$ 
8.   TypeCheckState(C.state_block,  $\Gamma\_local$ )
9.   For each ProcDef P in C.procs:
10.     $\Gamma\_proc \leftarrow \Gamma\_local.extend(P.params)$ 
11.    TypeCheckStmts(P.body,  $\Gamma\_proc$ )
12.   For each LoopDef L in C.loops:
13.    if strict_mode: verify L.period is compile-time
14.    TypeCheckStmts(L.body,  $\Gamma\_local$ )

TypeCheckExpr(Expr e, Env  $\Gamma$ )  $\rightarrow$  Unit u:
15. match e:
16.   IdentExpr(x):      return  $\Gamma.lookup\_unit(x)$ 
17.   LiteralExpr(v, u): return resolve(u)
18.   BinOpExpr(e1, +, e2):
19.      $u1 \leftarrow TypeCheckExpr(e1, \Gamma)$ 
20.      $u2 \leftarrow TypeCheckExpr(e2, \Gamma)$ 
21.     if  $u1 \neq u2$ : raise UnitMismatchError(e.loc,  $u1, u2$ )
22.     return  $u1$ 
23.   BinOpExpr(e1, *, e2):
24.     return  $TypeCheckExpr(e1, \Gamma) \times TypeCheckExpr(e2, \Gamma)$ 
25.   BinOpExpr(e1, /, e2):
26.     return  $TypeCheckExpr(e1, \Gamma) / TypeCheckExpr(e2, \Gamma)$ 

```

5.3 Code Generation

The code generator traverses the unit-annotated AST and emits Python source code. The key design decision is unit erasure: unit annotations on types are not emitted to the output. The generated code contains only float arithmetic, with no wrapper objects or metadata. COMPONENT becomes a Python class, STATE variables become instance attributes, PROC declarations become instance methods, and LOOP declarations become `step()` methods called by the simulation loop.

6. CASE STUDY: MORPHING WING UAV CONTROLLER

6.1 Problem Formulation

The control objective is to regulate the wing lift coefficient C_L to a target value $C_L^* = 0.8$, representative of efficient cruise, by independently commanding wing span $b(t) \in [0.7, 1.0]$ m and camber ratio $\eta(t) \in [0.12, 0.24]$. The wind speed $V_\infty(t)$ is a slowly varying disturbance with a nominal value of 12 m/s and step changes introduced during the simulation. The wing is modelled as a tapered, linearly elastic beam with span b , root chord $c_r = 0.25$ m, taper ratio $\lambda = 0.6$, and NACA 2412 baseline section profile.

6.1.1 Methodological Workflow

The development and evaluation workflow for the AeroLang-based controller followed a structured four-stage process: (1) **Physical Modelling**, where morphing-wing aerodynamic behaviour and structural elasticity were represented using Vortex Lattice Method (VLM) approximations and Euler-Bernoulli beam formulations; (2) **DSL Specification**, where flight-control logic was authored in AeroLang using explicit SI-unit annotations and `@EMBEDDED` constraints; (3) **Automated Compilation**, where the AeroLang compiler performed static dimensional verification and generated unit-erased Python code; and (4) **Closed-Loop Simulation**, where the generated controller was integrated into a physics-based simulation environment for system-level evaluation.

This methodology differs from conventional workflows by replacing manual unit coordination and ad-hoc runtime integration with a compiler-driven, correct-by-construction abstraction process. By handling dimensional universality, aerospace semantics, and embedded-system constraints directly at the language level, the workflow reduces the likelihood of silent numerical inconsistencies while improving modularity and cyber-physical coordination across the control architecture.

6.2 Aerodynamic Model: Free Vortex Wake

The aerodynamic model uses a free-vortex-wake approach. The induced velocity u_{ind} at any point p in the flow field is computed using the Biot-Savart law:

$$u_{ind}(p) = \Gamma / (4\pi) \int dl \times \hat{r} / |r|^2 \quad (8)$$

where Γ is the vortex filament circulation strength, dl is the filament direction element, and r is the displacement vector from the filament element to the field point. The lift and induced drag coefficients are computed from the circulation distribution by application of the Kutta-Joukowski theorem:

$$\mathbf{L} = \rho V_\infty \int \Gamma(y) dy, \quad \mathbf{D}_i = \rho V_\infty \int \Gamma(y) \alpha_i(y) dy \quad (9)$$

6.3 Aeroelastic Model

Aerodynamic loads are coupled to structural deflections using an Euler-Bernoulli beam model for the wing spar [29]. The governing equations for spanwise bending deflection $w(y)$ and twist angle $\phi(y)$ are:

$$EI d^4w/dy^4 = q(y), \quad GJ d^2\phi/dy^2 = -m(y) \quad (10)$$

where E is Young's modulus, I is the second moment of area, G is the shear modulus, and J is the Saint-Venant torsion constant. These equations are discretised using a second-order finite difference scheme over the spanwise grid and solved as a banded linear system at each time step.

6.4 Controller Architecture

The controller logic is implemented in a 262-line AeroLang source file. It consists of two coupled sub-controllers. A proportional-integral (PI) controller regulates the span $b(t)$ based on the error in lift coefficient C_L , with proportional gain $K_P = 0.5$ m and integral gain $K_I = 0.1$ m/s. A proportional camber scheduler maps the wind speed V_∞ to a target camber ratio $\eta^*(V_\infty)$ according to a schedule computed offline:

$$\eta^*(V_\infty) = 0.18 \times (1.0 + 0.1 / ((V_\infty/12.0)^2 + 0.1)) \quad (11)$$

7. RESULTS

7.1 Simulation Results

The closed-loop simulation was run for 120 seconds of simulated flight time with a fixed time step of $\Delta t = 0.01$ s, using a fourth-order Runge-Kutta integrator. Table 1 summarises the key performance metrics.

Table 1. Performance comparison between rigid-wing baseline and AeroLang-generated morphing-wing controller (120 s simulation, nominal wind speed 12 m/s, step disturbances at $t = 30$ s and $t = 80$ s).

Metric	Rigid Wing	Morphing Wing	Improvement
Mean L/D Ratio	14.2	15.8	+11.3%
CL RMSE	0.087	0.031	-64.4%
Mean Span (m)	0.800 (fixed)	0.871	–
Mean Camber	0.180 (fixed)	0.193	–
Settling Time (s)	–	4.2	–
Comp. Overhead	Baseline	+0.3%	Negligible

The 11.3% improvement in mean lift-to-drag ratio is consistent with the range of 10–20% reported in the experimental and computational literature [6,7]. The PI span controller achieves a settling time of approximately 4.2 seconds after the wind-speed step disturbances, with a C_L root-mean-square error of 0.031 compared to the target of 0.8. The computational overhead of the generated controller relative to the rigid-wing simulation is 0.3%, confirming the zero-cost abstraction claim.

7.2 Compile-Time Error Detection

The most important result is the compiler’s behaviour during development. During the authoring of the 262-line AeroLang controller, the compiler detected and reported seven unit-mismatch errors before any simulation was run. Representative examples are given in Table 2.

Table 2. Representative compile-time unit-mismatch errors detected by the AeroLang type checker during authoring of the morphing-wing controller.

Line	Erroneous Expression	Compiler Diagnostic
L.47	FORCE = MASS + ACCEL	ERROR: Unit mismatch: kg is not compatible with m/s^2
L.89	CL = LIFT / VELOCITY	ERROR: Unit mismatch: $\text{kg}\cdot\text{m}\cdot\text{s}^{-2}$ vs $\text{m}\cdot\text{s}^{-1}$
L.112	SPAN = SPAN + RATE * DT * DT	ERROR: Assign mismatch: m vs m’s
L.134	ERROR = CL_TARGET - LIFT	ERROR: Unit mismatch: 1 vs $\text{kg}\cdot\text{m}\cdot\text{s}^{-2}$

8. DISCUSSION

The results demonstrate that AeroLang successfully achieves its two primary design objectives: compile-time dimensional safety and zero runtime overhead. The seven mismatch errors detected during controller development represent a qualitative proof-of-concept: each would have manifested as a silent numerical error during simulation, requiring expensive debugging to isolate. In the most serious cases – such as using a force (N) where a dimensionless coefficient is expected – the erroneous value would have been incorrect by orders of magnitude, likely causing immediate simulation divergence. In more subtle cases – such as accumulating a quantity in m’s where m is required – the error would have accumulated slowly, potentially producing plausible-looking but incorrect results.

The zero-cost abstraction property of AeroLang is a direct consequence of the unit erasure strategy, inherited from the F# Units of Measure design [9]. The 0.3% overhead observed in the case study is attributable entirely to the class dispatch overhead of the object-oriented Python architecture, not to any unit-related computation.

The comparison with related approaches highlights AeroLang’s distinctive position. Runtime dimensional-checking libraries such as Python’s Pint [21] catch the same class of errors but impose overhead that is prohibitive in real-time control loops. F# Units of Measure provides equivalent compile-time safety but uses a general-purpose language with syntax unfamiliar to control engineers. AADL [15] and Lustre/SCADE [17] provide strong real-time guarantees but no dimensional analysis. AeroLang occupies a unique intersection: domain-specific syntax, compile-time dimensional safety, zero runtime overhead, and real-time-aware semantics.

To further contextualise AeroLang within the landscape of safety-critical modelling and dimensional analysis, Table 3 provides a comparative summary against established frameworks. While F# Units of Measure and Pint focus primarily on dimensional safety, and Lustre/SCADE and AADL prioritise real-time semantics and architectural modelling respectively, AeroLang integrates these concerns within an aerospace-oriented DSL framework designed for cyber-physical coordination, embedded deployment, and real-time flight-control abstraction.

Table 3. Comparative analysis of AeroLang against existing frameworks.

Feature	F# Units of Measure	Pint (Python)	Lustre / SCADE	AADL	AeroLang
Dimensional Safety	Static (Compile-time)	Dynamic (Runtime)	None	Limited (Properties)	Static (Compile-time)
Real-Time Semantics	None	None	Synchronous Dataflow	Architectural	Periodic Loops
Embedded Compatibility	Low (GC-dependent)	Low (Interpreter)	High (Certified C)	High (Modelling)	High (@EMBEDDED)
Aerospace Semantics	General	General	Avionics-oriented	System-oriented	Physics-integrated
Runtime Overhead	Minimal	High	Minimal	N/A	No dimensional runtime overhead
Morphing Physics Support	No	No	No	No	Integrated Support

Several limitations should be acknowledged. First, the Python back-end, while suitable for simulation and rapid prototyping, is not suitable for deployment on resource-constrained microcontrollers due to interpreter overhead. A C code generator is the next essential step. Second, the type system currently supports only integer exponents in unit tuples, which excludes fractional-dimension quantities. Extending to rational exponents, as in Kennedy’s original formulation [20], would

address this limitation. Third, the aerodynamic and structural models in the case study are simplified; for production use, higher-fidelity models would be required. Fourth, the current implementation does not perform schedulability analysis; verifying that the declared loop periods are jointly feasible on target hardware remains the responsibility of the engineer.

8.1 Real-Time Performance and Resource Constraints

While the current implementation utilises a Python backend for rapid prototyping and simulation, the `@EMBEDDED` discipline is specifically designed to facilitate predictable execution on target hardware. Preliminary analysis of the generated code structure confirms the absence of unbounded recursion and unrestricted dynamic allocation, which are prerequisites for formal Worst-Case Execution Time (WCET) analysis. Empirical WCET measurements and memory footprint profiling are planned as part of the upcoming C backend implementation. This transition will enable the use of industrial-grade static analysis tools, such as AbsInt aiT, to derive tight execution bounds relevant to high-assurance flight-control deployment contexts. At present, no certification-oriented timing analysis has been performed on generated binaries, as the current implementation targets simulation-oriented Python execution.

9. FUTURE WORK

The development of AeroLang is ongoing, with a roadmap structured around three parallel tracks:

Formal verification. The compiler will be extended to generate verification conditions in the SMT-LIB2 format from AeroLang source programs. These verification conditions can be discharged automatically by SMT solvers such as Z3 [37] or CVC5 [38], enabling formal proofs of properties including control loop stability, state-space constraint satisfaction, and bounded convergence times for PI controllers. The connection between AeroLang’s type system and separation logic [42] offers a promising route to proving memory safety properties of the `@EMBEDDED` discipline.

Embedded deployment. A C code generator is the highest-priority practical deliverable. The generated C will target MISRA C:2012 compliance [31] and will be designed to facilitate DO-178C Level C certification [30], with paths to Level A through integration with qualified verification tools. Hardware-in-the-loop testing on a Pixhawk 6C flight computer is planned.

Community adoption. AeroLang is intended to serve as a common, verifiable specification language for morphing-wing research. Integration with machine learning frameworks – in particular, the ability to specify the structure of neural-network controllers as AeroLang modules with learnable parameters – is a longer-term

ambition. The connection between the AeroLang type system and the physics-informed neural network literature [43] offers the prospect of training controllers that are dimensionally correct by construction.

10. CONCLUSIONS

This paper presented AeroLang, an integrative aerospace-oriented abstraction framework designed to unify dimensional universality, cyber-physical modelling, embedded-system constraints, and real-time flight-control semantics within a single extensible domain-specific language. By grounding physical quantities in formal dimensional analysis and enforcing an `@EMBEDDED` execution discipline, AeroLang aims to eliminate a critical class of safety-relevant software defects associated with unit inconsistency and non-deterministic execution behaviour.

The morphing-wing UAV case study demonstrated the practical applicability of the proposed framework for physics-aware flight-control representation and dimensional verification within a cyber-physical aerospace context. The results support the feasibility of combining aerospace semantics, embedded-oriented execution structure, and dimensional correctness within a unified compiler-driven methodology.

Future work will focus on the development of a certifiable-oriented C backend suitable for resource-constrained embedded deployment. This transition will include formal WCET analysis, memory footprint profiling, and integration with industrial static-analysis workflows. Additional research directions include SMT-assisted verification, advanced multi-physics integration, and broader support for adaptive aerospace structures and autonomous control architectures.

As an extensible research framework, AeroLang contributes toward the long-term development of more physically-aware, formally constrained, and computationally robust aerospace software systems.

REFERENCES

- [1] Feron E. From control systems to control software. *IEEE Control Syst Mag.* 2010;30(6):50–71. <https://doi.org/10.1109/MCS.2010.938196>
- [2] Mars Climate Orbiter Mishap Investigation Board. Phase I Report. Washington, DC: National Aeronautics and Space Administration; 1999.
- [3] Open Group. *FACE™ Technical Standard, Edition 3.1*. Reading: The Open Group; 2020.
- [4] ARINC. *ARINC Specification 653P1-5: Avionics Application Software Standard Interface, Part 1 – Required Services*. Annapolis: ARINC; 2019.
- [5] Setlak L, Kowalik R. Practical use of composite materials used in military aircraft. *Materials*. 2021;14(17):4812. <https://doi.org/10.3390/ma14174812>

- [6] Jeger SL, Wüest V, Toumieh C, Floreano D. Adaptive morphing of wing and tail for stable, resilient, and energy-efficient flight of avian-inspired drones. *NPJ Robot.* 2024 Nov 20;2(1). <https://doi.org/10.1038/s44182-024-00015-y>
- [7] Ninian D, Dakka S. Design, development and testing of shape shifting wing model. *Aerospace.* 2017 Nov 1;4(4):52. <https://doi.org/10.3390/aerospace4030044>
- [8] Wang W, An W, Song B. Effect of wing morphing on stability and energy harvesting in albatross dynamic soaring. *Chin J of Aeronaut.* 2024;37(11):317–334. <https://doi.org/10.1016/j.cja.2024.06.013>
- [9] Kennedy AJ, Syme D. Design and implementation of generics for the .NET Common Language Runtime. *ACM SIGPLAN Not.* 2001;36(5):1–12. <https://doi.org/10.1145/381694.378797>
- [10] Pierce BC. *Types and programming languages.* Cambridge: MIT Press; 2002.
- [11] Fourier JBJ. *Théorie analytique de la chaleur [Analytical theory of heat].* Paris: Firmin Didot; 1822. French.
- [12] Maxwell JC. *A treatise on electricity and magnetism.* Oxford: Clarendon Press; 1873.
- [13] Bridgman PW. *Dimensional analysis.* 2nd ed. New Haven: Yale University Press; 1931.
- [14] Chaulagain, A. *Physics Integrated Morphing UAV (v1.0) [Software].* Zenodo; 2026. <https://doi.org/10.5281/zenodo.19259263>
- [15] Feiler PH, Gluch DP, Hudak JJ. *The Architecture Analysis & Design Language (AADL): An Introduction.* Pittsburgh: Carnegie Mellon University, Software Engineering Institute; 2006. Report No.: CMU/SEI-2006-TN-011.
- [16] Heitmeyer CL, Jeffords R, Labaw BG. Automated consistency checking of requirements specifications. *ACM Trans Softw Eng Methodol.* 1996;5(3):231–261. <https://doi.org/10.1145/234426.234431>
- [17] Halbwachs N, Caspi P, Raymond P, Pilaud D. The synchronous data flow programming language LUSTRE. *Proc IEEE.* 1991;79(9):1305–1320. <https://doi.org/10.1109/5.97300>
- [18] Benveniste A, Caspi P, Edwards SA, Halbwachs N, Le Guernic P, de Simone R. The synchronous languages 12 years later. *Proc IEEE.* 2003;91(1):64–83. <https://doi.org/10.1109/JPROC.2002.805826>
- [19] Quigley M, Conley K, Gerkey B, Faust J, Foote T, Leibs J, Wheeler R, Ng A. *ROS: An open-source robot operating system.* In: *ICRA Workshop on Open Source Software*; 2009.
- [20] Kennedy A. *Programming Languages and Systems — ESOP '94.* Berlin, Heidelberg: Springer Berlin Heidelberg; 1994. Dimension types; p. 348–362. https://doi.org/10.1007/3-540-57880-3_23
- [21] Microsoft Corporation. *Units of measure – F# language reference.* Microsoft Docs; 2023. <https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/units-of-measure>
- [22] Grecco HE. *Pint: Operate and manipulate physical quantities in Python [Software].* GitHub; 2022. <https://github.com/hgrecco/pint>
- [23] Anderson JD. *Fundamentals of aerodynamics.* 3rd ed. New York: McGraw-Hill; 2001.
- [24] Barbarino S, Bilgen O, Ajaj RM, Friswell MI, Inman DJ. A review of morphing aircraft. *J Intell Mater Syst Struct.* 2011;22(9):823–877. <https://doi.org/10.1177/1045389X11414084>

- [25] Katz J, Plotkin A. Low-speed aerodynamics. 2nd ed. Cambridge: Cambridge University Press; 2001. <https://doi.org/10.1017/CBO9780511810329>
- [26] Leishman JG. Principles of helicopter aerodynamics. 2nd ed. Cambridge: Cambridge University Press; 2006.
- [27] Bisplinghoff RL, Ashley H, Halfman RL. Aeroelasticity. Reading: Addison-Wesley; 1955.
- [28] Fung YC. An introduction to the theory of aeroelasticity. New York: John Wiley & Sons; 1955.
- [29] Timoshenko SP. Strength of materials, Part II: Advanced theory and problems. 2nd ed. New York: D. Van Nostrand Company; 1940.
- [30] RTCA. DO-178C: Software considerations in airborne systems and equipment certification. Washington, DC: RTCA Inc.; 2011.
- [31] MISRA Consortium. MISRA C: 2012 – Guidelines for the use of the C language in critical systems. Nuneaton: MIRA Ltd.; 2013.
- [32] Seacord RC. The CERT C coding standard: 98 rules for developing safe, reliable, and secure systems. 2nd ed. Upper Saddle River: Addison-Wesley; 2014.
- [33] Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mitra T, Mueller F, Puaut I, Puschner P, Staschulat J, Stenström P. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans Embed Comput Syst*. 2008 Apr;7(3):1–53. <https://doi.org/10.1145/1347375.1347389>
- [34] Shinan E. Lark: A modern parsing library for Python [Software]. GitHub; 2020. <https://github.com/lark-parser/lark>
- [35] Milner R. A theory of type polymorphism in programming. *J Comput Syst Sci*. 1978;17(3):348–375. [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4)
- [36] Hindley JR. The principal type-scheme of an object in combinatory logic. *Trans Amer Soc*. 1969;146:29–60. <https://doi.org/10.1090/S0002-9947-1969-0253905-6>
- [37] de Moura L, Bjørner N. Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer Berlin Heidelberg; 2008. Z3: An Efficient SMT Solver; p. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
- [38] Barbosa H, Barrett C, Brain M, Kremer G, Lachnitt H, Mann M, Mohamed A, Mohamed M, Niemetz A, Nötzli A, Ozdemir A, Preiner M, Reynolds A, Sheng Y, Tinelli C, Zohar Y. Tools and Algorithms for the Construction and Analysis of Systems. Cham: Springer International Publishing; 2022. cvc5: A Versatile and Industrial-Strength SMT Solver; p. 415–442. https://doi.org/10.1007/978-3-030-99524-9_24
- [39] Buckingham E. On physically similar systems; illustrations of the use of dimensional equations. *Phys Rev*. 1914;4(4):345–376. <https://doi.org/10.1103/PhysRev.4.345>
- [40] Kopetz H. Real-time systems: Design principles for distributed embedded applications. 2nd ed. New York: Springer; 2011. <https://doi.org/10.1007/978-1-4419-8237-7>
- [41] Liu CL, Layland JW. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J ACM*. 1973;20(1):46–61. <https://doi.org/10.1145/321738.321743>
- [42] O'Hearn PW. Separation logic. *Commun ACM*. 2019;62(2):86–95. <https://doi.org/10.1145/3211968>

- [43] Raissi M, Perdikaris P, Karniadakis GE. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J Comput Phys.* 2019;378:686–707. <https://doi.org/10.1016/j.jcp.2018.10.045>
- [46] Paul S, Cruz E, Dutta A, Bhaumik A, Blasch E, Agha G, Patterson S, Kopsaftopoulos F, Varela C. Formal verification of safety-critical aerospace systems. *IEEE Aerosp Electron Syst Mag.* 2023;38(5):72–88. <https://doi.org/10.1109/maes.2023.3238378>
- [47] Manzhos Y, Sokolova Y. Formal verification of aerospace cyber-physical system software. *Proc 5th Int IT Prof Artif Intell (ProfIT AI 2025)* 2025;4164:289–297. <https://ceur-ws.org/Vol-4164/paper19.pdf>