

AUTOMATIC GENERATION OF PHYSICAL SYSTEM MODELS FOR AIRCRAFT SYSTEMS BASED ON LLMS

Yuli REN[1], Lingfei YOU[2,*], Xinghai GAO[3], Zhuoqi WANG[4], Chuangye CHANG[5] & Zhiqi GUO[6]

[1]Institute of Unmanned System, Beihang University, Beijing, China
[2]Institute of Unmanned System, Beihang University, Beijing, China
[3]Institute of Unmanned System, Beihang University, Beijing, China
[4]Institute of Unmanned System, Beihang University, Beijing, China
[5]Institute of Unmanned System, Beihang University, Beijing, China
[6]Institute of Unmanned System, Beihang University, Beijing, China

Abstract

Modern aircraft systems increasingly exhibit multidisciplinary integration and strong coupling, creating an urgent need for MBSE-based unified modeling and virtual verification. Modelica has become an important standard for multidisciplinary collaborative simulation modeling. However, traditional modeling relies heavily on expert experience and faces challenges such as fragmented knowledge, difficulty in reuse, and hallucinations and syntax deviations generated by general-purpose large language models. To address these issues, this paper proposes a large-language-model-based method for automatic generation of Modelica models for aircraft systems. The method deeply integrates a RAG-based domain knowledge base and constructs an AI-agent collaborative workflow. By introducing a structured intermediate representation as a semantic bridge between natural language and code, the system establishes precise mapping rules from requirement descriptions to model declarations, parameter and variable definitions, and equation definitions. The method implements an end-to-end closed loop covering intent understanding, knowledge retrieval, high-fidelity code generation, and syntax checking. Experimental results show that the proposed method substantially improves the modeling accuracy and compilability of aircraft subsystems and provides a reusable technical route for rapid virtual verification of complex aerospace systems.

Keywords: large language model; aircraft systems; Modelica; RAG; AI agent

1. Introduction

As the coupling among components of modern aircraft systems continues to increase, multidisciplinary unified modeling that combines MBSE and Modelica is becoming an important direction for aircraft-system design and verification. However, conventional Modelica modeling relies on stepwise decomposition and manual coding by experts, resulting in a high entry threshold and long development cycles. General-purpose LLMs lack constraints from aeronautical domain knowledge and are prone to semantic misunderstanding, component misuse, and syntax errors.

In recent years, large language models have developed rapidly in the field of code generation. Studies on tools such as GitHub Copilot, Amazon CodeWhisperer, and StarCoder indicate that LLMs already show strong capabilities in boilerplate-code completion, standard algorithm implementation, and program generation for general-purpose software development [1-5]. Meanwhile, the model-driven engineering community has begun to combine LLMs, agents, and engineering-knowledge retrieval mechanisms to support model-transformation selection, engineering-rule invocation, and complex task decomposition [6-8]. For Modelica-oriented scenarios, existing studies have shown that although general-purpose models can generate local code fragments, they still have clear deficiencies in syntactic completeness, physical-semantic consistency, and simulability. Dedicated workflows, domain-knowledge enhancement, and feedback-optimization mechanisms are therefore required to significantly improve code-generation quality and simulation pass rates [7,9]. From the

https://doi.org/10.71945/icas2026_0479

perspective of modeling-language foundations, Modelica, as a unified object-oriented equation-based language, emphasizes component-based modeling, hierarchical composition, and equation-constraint representation for multidomain complex systems [10-13]. This provides a sound foundation for integrated modeling of aircraft airframe, propulsion, actuation, and control systems. It also means, however, that automatic generation must not only satisfy formal syntax correctness, but also ensure consistency among interfaces, parameters, connection relations, mechanism equations, and engineering constraints.

Therefore, for aircraft-system modeling scenarios, this paper proposes an LLM domain-specific architecture that integrates a structured knowledge base, hybrid retrieval strategy, specialized prompt engineering, and AI-agent collaborative workflow. The architecture progressively transforms natural-language requirements into executable Modelica code and implements an end-to-end closed loop from requirement understanding to model generation and checking.

2. Automatic Model Generation Method for Aircraft Systems

2.1 Overall Method Framework

The proposed method consists of five stages: domain-knowledge preparation, RAG-based retrieval augmentation, multi-agent collaborative generation, prompt-based constraints, and tool-side verification. Its objective is to stably transform natural-language requirements for aircraft systems into compilable and reusable Modelica models. The key to its implementation is to define generation boundaries using specialized knowledge, decompose complex tasks through agents, control output formats through prompts, and validate the results using the OpenModelica toolchain, thereby forming a closed loop of requirement, knowledge, generation, and checking. The overall framework of the proposed automatic model generation method is shown in Figure 1.

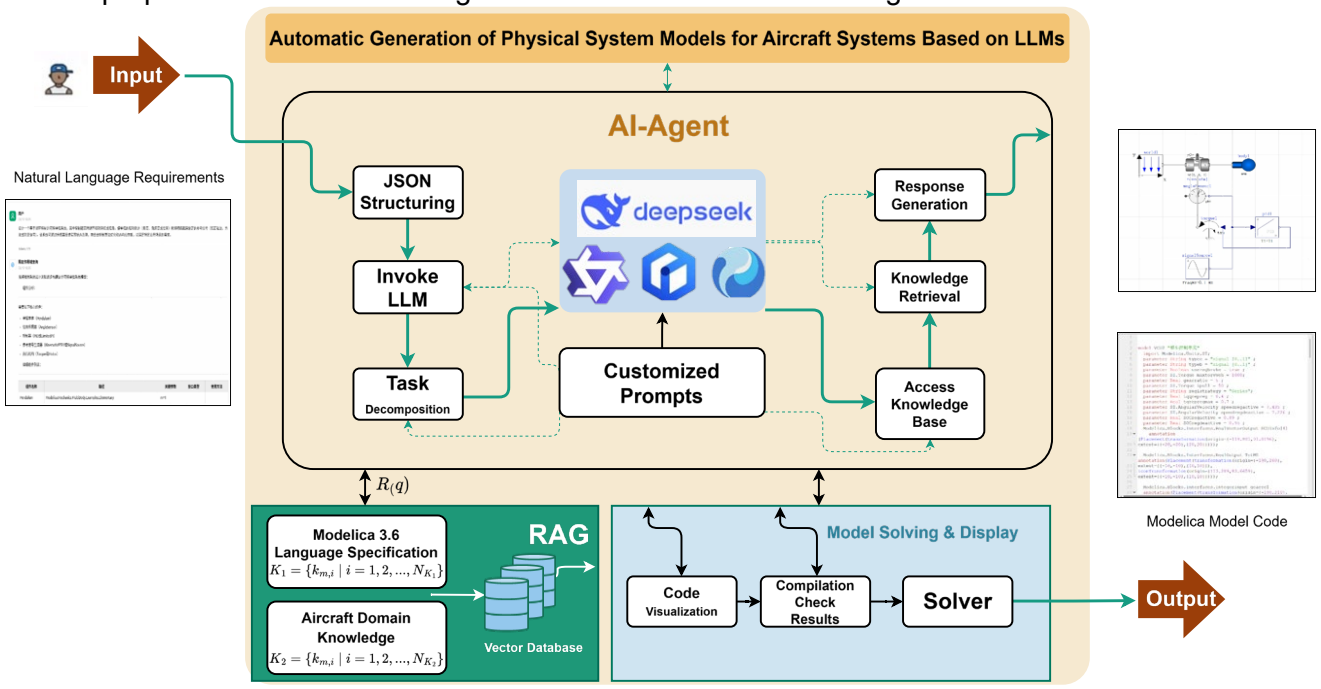


Figure 1 – Overall framework of the automatic model generation method for aircraft systems.

2.2 Construction of the Domain Knowledge Base

The domain knowledge base is constructed for automatic modeling of aircraft systems. Its inputs include the Modelica 3.6 language specification, aircraft-domain knowledge, academic papers, and typical subsystem cases. A structured knowledge-representation format for aircraft-domain knowledge is designed. Raw materials are uniformly parsed using tools such as Markdown; text, formulas, interface definitions, parameter descriptions, and constraint rules are extracted and transformed into editable structured entries. A single knowledge entry is defined as:

$$k_i = (\tau_i, n_i, e_i, V_i, s_i, r_i) \tag{1}$$

where the natural-language requirement is denoted as x, the domain knowledge base as

$K=\{k_i \mid i=1,2,...,N_K\}$, τ_i denotes the knowledge type, n_i denotes the knowledge name, e_i denotes the formula or rule expression, V_i denotes the variable set, s_i denotes the application scenario, and r_i denotes constraints and notes.

During knowledge organization, unified fields are established around automatic-generation requirements, such as knowledge type, knowledge name, formula expression, variable list, application scenario, and notes. A chunking strategy under semantic-completeness constraints is then used to segment long texts. The text chunks are vectorized using an embedding model and stored in a vector database, such as Elasticsearch. A hybrid index combining vector retrieval, keyword matching, and structured queries is constructed to improve the precise recall of formulas, component templates, and modeling rules.

The set of text chunks under semantic-completeness constraints is denoted as $C=\{c_j \mid j=1,2,...,N_C\}$, and the candidate knowledge-retrieval set for query q can be expressed as:

$$R(q)=\mathrm{TopK}\left(s_{ret}\left(q,c_j\right),c_j \in C\right) \tag{2}$$

where $s_{ret}\left(q,c_j\right)$ denotes the integrated retrieval score of knowledge chunk c_j with respect to query q, and $R(q)$ serves as the shared knowledge input for subsequent requirement understanding, code generation, and checking.

Through API interfaces and access protocols, knowledge entries that pass recall testing are connected to the generation system to form the Modelica domain knowledge base. This knowledge base covers the language-specification layer, physical-mechanism layer, and engineering-application layer. The former supports model declarations, inheritance, interfaces, and connection rules, while the latter two support component mechanisms for hydraulic, electrical, thermal, and control domains, as well as common naming conventions and boundary conditions in aircraft systems.

2.3 AI-Agent Design

On this basis, the system designs five types of agents: requirement understanding, component generation, system generation, code packaging, and code compilation. The requirement-understanding agent parses natural language into a unified JSON modeling requirement. The component-generation agent generates subsystem-level models for the airframe, propulsion, flight control, and other subsystems. The system-generation agent completes instance assembly, connection-relation expansion, and annotation completion. The code-packaging agent merges the previously generated component and system code into a package. The code-compilation agent performs static syntax checking, specification-consistency checking, and tool-side compilation checking.

All agents share a unified data protocol, while using differentiated prompts and parameter configurations according to their task characteristics. The knowledge-Q&A stage emphasizes retrieval recall and semantic understanding, whereas the code-generation stage emphasizes long-context preservation, structured output, and syntactic stability. Information is passed between stages through JSON, avoiding semantic drift caused by multi-round natural-language transfer. Local deployment, inference acceleration, and error-feedback mechanisms are also incorporated to improve system stability and engineering reproducibility.

The collaborative set composed of the five types of agents can be denoted uniformly as:

$$A=\left\{a_{req},a_{cmp},a_{sys},a_{asm},a_{chk}\right\} \tag{3}$$

where a_{req}, a_{cmp}, a_{sys}, a_{asm}, and a_{chk} correspond to the requirement-understanding, component-generation, system-generation, code-packaging, and code-compilation agents, respectively. Under the unified data protocol, the main information flow among the five types of agents can be written as:

$$m=a_{req}\left(x,R(q)\right),y_c=a_{cmp}\left(m,R(q)\right),y_s=a_{sys}\left(y_c,m\right),y=a_{asm}\left(y_c,y_s,m\right),v=a_{chk}\left(y\right) \tag{4}$$

where m denotes the structured modeling requirement, y_c denotes the component-level code result, y denotes the system-level Modelica code, and v denotes the checking-feedback result.

3. Structured Representation and Workflow Configuration

3.1 Logical Mapping of Structured Representation

To realize stable transformation from natural-language requirements to Modelica code, this paper introduces JSON as a unified intermediate structured representation. A template is designed to extract knowledge from domain materials and form structured information, including knowledge type, knowledge name, formula expression, formula description, variable list, and other elements. The system first normalizes user input into fields such as modelDefinition, imports, extends, interfaces, parameters, variables, subcomponents, equations, algorithms, and annotations, and then generates code according to deterministic rules. This mechanism avoids the structural drift that occurs when a large language model jumps directly from text to code, enabling modeling requirements to be transmitted layer by layer in a machine-processable and reviewable form.

In the mapping logic, the model name and description correspond to model or block declarations; interface fields correspond to connector declarations and their directions, types, and paths; parameter and variable fields correspond to parameter statements and internal states, respectively; subcomponents correspond to component instantiation; equations.standard and equations.connections correspond to mechanism equations and connect statements, respectively; and algorithms correspond to the algorithm section. This "structure-first, code-afterward" mechanism significantly improves the consistency, interpretability, and reusability of the output. The mapping relationships between JSON structured fields and Modelica code elements are summarized in Table 1.

Table 1 – schematic table of mapping relationships

JSON structured field	Field meaning	Corresponding Modelica code element	Mapping description
modelDefinition. name	Model name	model / block declaration	Directly generates the name of the Modelica model or module, e.g., model XXX or block XXX
modelDefinition. description	Model function description	annotation() comment	Used to describe the model function, applicable object, and modeling intent
imports / extends	External library references and inheritance	import / extends statements	Imports or inherits from the Modelica Standard Library, domain model libraries, or custom packages
interfaces.name	Interface name	Port or connector instance name	Generates interface variable names, such as u, y, port_a, and pin_p
equations. connections	Component connection relationships	connect(...) statements	Generates topological connections among ports or connectors of different components
interfaces.type	Interface type	Type declaration or connector type	Determines the interface data type or connector type, such as Real, Boolean, RealInput, RealOutput, or Pin
parameters	Model parameters	Parameter declarations	Defines configurable parameters, such as physical parameters and control parameters
variables	Internal variables or states	Variable declarations	Defines states, intermediate variables, or algebraic variables during model operation
subcomponents	Subcomponent information	Component declarations	Decomposes a complex system into reusable submodels or module instances
equations. standard	Standard mechanism equations	equation statements	Generates physical laws, constraint relations, and state-evolution equations
algorithms	Procedural computational logic	algorithm section	Generates conditionals, loops, event handling, and other non-pure-equation logic

3.2 Prompt and Workflow Design

In terms of workflow design, the system configures specialized prompt templates for different agents, specifying their roles, input formats, output boundaries, and error-avoidance requirements. The

prompts for the requirement-understanding agent emphasize modeling objectives, system boundaries, and constraint identification. The prompts for component generation, system generation, and code packaging emphasize complete path references, variable naming, unit systems, connection statements, and annotation formats. The prompt for the code-compilation agent focuses on parentheses, semicolons, type consistency, unit validity, and interpretation of tool-side error messages.

To balance semantic recall, keyword matching, and reranking correction, the integrated score in the knowledge-retrieval stage is uniformly written as:

$$s_{ret}(q,c_i) = \beta s_{rerank}(q,c_i) + (1-\beta)\left[(1-\alpha)s_{sem}(q,c_i) + \alpha s_{key}(q,c_i)\right] \tag{5}$$

where $s_{sem}(q,c_i)$ is the semantic-similarity score, $s_{key}(q,c_i)$ is the keyword-matching score, $s_{rerank}(q,c_i)$ is the reranking-model score, and α and β control the keyword weight and reranking weight, respectively.

The workflow is executed sequentially as a chain of requirement understanding, knowledge retrieval, JSON filling, component generation, system generation, code packaging, code compilation, and feedback correction. After the user submits natural-language requirements, the system first retrieves relevant rules, formulas, and templates, then generates structured JSON results, and subsequently generates component and system code accordingly. The code-compilation agent then performs compilation checks. Based on the ANTLR checking tool, it examines variables, interfaces, equations, and other elements. If problems such as undefined types, incorrect placement of connect statements, inconsistent units, or missing end statements are found, the error messages are returned to the corresponding agent for regeneration, forming a closed loop with a checking cycle.

Considering the closed-loop correction mechanism driven by checking feedback, the generation and correction process at the t-th iteration can be expressed as:

$$y(t+1) = G\left(y(t), v(t)\right), v(t) = a_{chk}\left(y(t)\right), t = 0,1,...,T-1 \tag{6}$$

where $G(\cdot)$ denotes the regeneration operator based on checking feedback. When the checking feedback is empty ($v(t)$) or compilation passes, the iteration terminates and outputs the final code y.

As illustrated in Figure 2, the proposed workflow follows a sequential and closed-loop generation process. First, the user's natural-language modeling requirement is parsed by the requirement-understanding agent and transformed into a structured JSON representation. Then, the system retrieves relevant Modelica syntax rules, aircraft-domain mechanisms, component templates, and engineering constraints from the domain knowledge base. Based on the retrieved knowledge and the structured JSON, component-level agents generate subsystem models, including flight control, aerodynamics, propulsion, and airframe dynamics. Subsequently, the system-architecture agent instantiates these components and constructs their connection relationships to form the top-level aircraft-system Modelica model.

After code generation, the code-packaging agent integrates all component models and system-level code into a complete Modelica package. The code-compilation agent then performs syntax checking, interface-consistency checking, unit checking, and tool-side compilation verification. If errors are detected, the checking feedback is returned to the corresponding generation stage for correction and regeneration. This iterative correction process continues until the feedback is empty or the generated code passes compilation. Therefore, the workflow forms an end-to-end closed loop from requirement understanding to executable Modelica code generation, which improves the structural consistency, syntactic correctness, and engineering usability of the generated model.

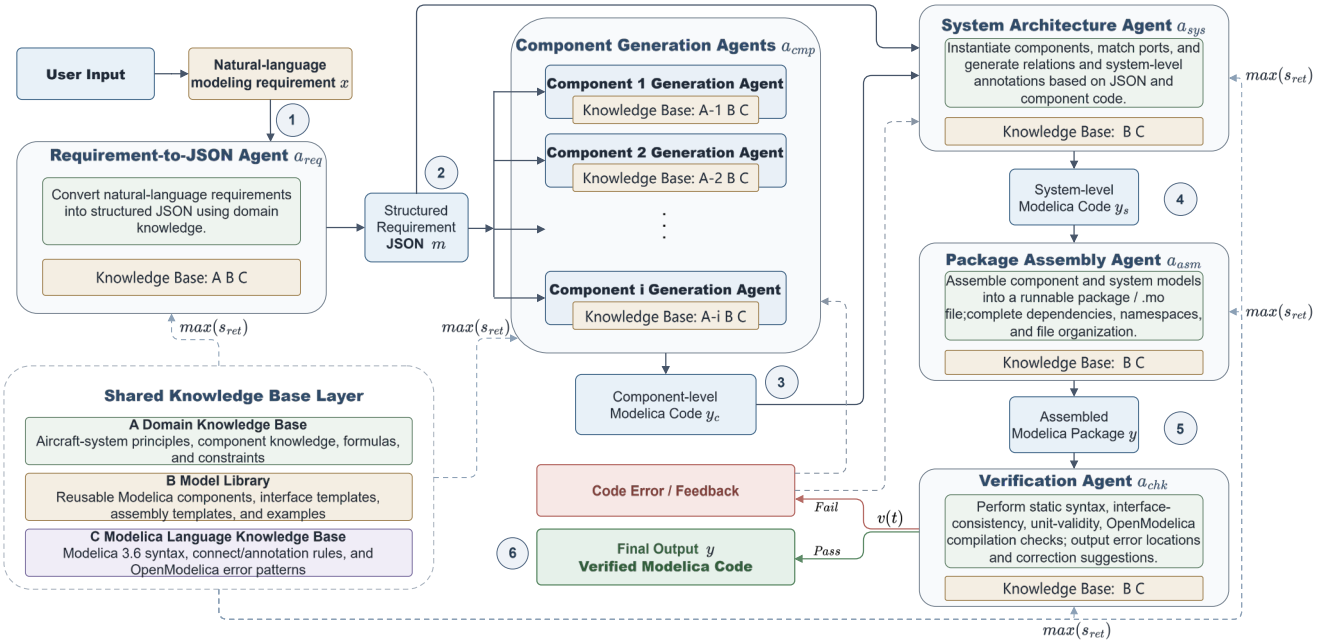


Figure 2 – Workflow of the LLM-based automatic generation method for aircraft-system Modelica models.

4. Case Verification

4.1 Case Overview

A typical aircraft longitudinal flight-control system is used for case verification. The case involves multiple coupled modules, including flight control, aerodynamics, propulsion, and airframe dynamics. The modeling requirements are described in natural language, covering the longitudinal motion-control objective of the aircraft, input/output variables, control commands, airframe-dynamic states, simulation time, and other basic information.

In the requirement-understanding stage, the system parses user inputs into fields such as model definitions, interface variables, parameter sets, internal state variables, component relations, and equation constraints.

In the component-generation stage, the component-code-generation agent generates flight-control, aerodynamic, propulsion, and airframe-dynamics components according to the structured requirements and the knowledge-base retrieval results. The system-architecture agent instantiates the components and generates the top-level system model. It completes the interface connections among the flight-control, aerodynamic, propulsion, and airframe-dynamics modules through connect(...) statements, and also complements the parameters, initial conditions, and annotation information required for simulation.

To further verify the rationality of the generated model's dynamic behavior, this paper sets up an altitude-tracking simulation task. Specifically, the target altitude smoothly climbs from 0 m to 300 m, remains at cruise, and then smoothly descends to 0 m. The velocity command is set to 100 m/s, the total simulation time is 42 min, and the simulation step size is 0.5 s. The main system outputs include flight altitude, velocity, angle of attack, pitch angle, pitch rate, horizontal position, thrust, lift, and drag, which are used to evaluate the dynamic-response rationality of the automatically generated model during flight-profile tracking. The following are partial prompt examples.

Prompt Example for the Flight-Control Component Code-Generation Agent

Role: *You are the Flight-Control Component Code-Generation Agent in an aircraft-system Modelica automatic modelinaaag workflow.*
Input: *The structured JSON produced by the Requirement-Understanding Agent; components.flightControl is the only modeling basis.*

Output: *Only one complete, compilable, and reusable FlightControl Modelica model.*

Notes:

Check task boundary: if flightControl is not required, return empty; otherwise generate.

Set model name: use modelName; default to FlightControl.

Read required fields: inputs, outputs, etc.

Merge global constraints into the generation context.

Infer interface intent from systemArchitecture.connections only; do not generate connect(...).

Retrieve domain knowledge by keywords and component fields.

Build ruleSet from retrieved knowledge, JSON constraints, etc.

Return an error flag if constraints are violated.

Generate one model skeleton according to the model name.

Generate RealInput/RealOutput interfaces and parameter Real(...) declarations with defaults.

Generate protected variables: errors, integrals, pitch commands, unsaturated commands, etc.

Generate only altitude-loop, pitch-loop, throttle-loop, and saturation equations.

Run consistency checks: single model, complete fields, etc.

Repair checkable issues; otherwise return an error flag.

Output the final FlightControl Modelica model.

4.2 Result Analysis

Based on proprietary workflow platform, this paper builds an agent workflow for automatic generation of Modelica system models, including a requirement-understanding agent, component-code-generation agents, a system-architecture agent, a code-assembly agent, and a model-validation agent. The embedded large language model is DeepSeek-V3, and it generated 91 lines of code with 14,180 characters. The configured agent workflow for the case study is illustrated in Figure 3. The requirement input, data-processing procedure, and output parameters of the case study are shown in Figure 4.

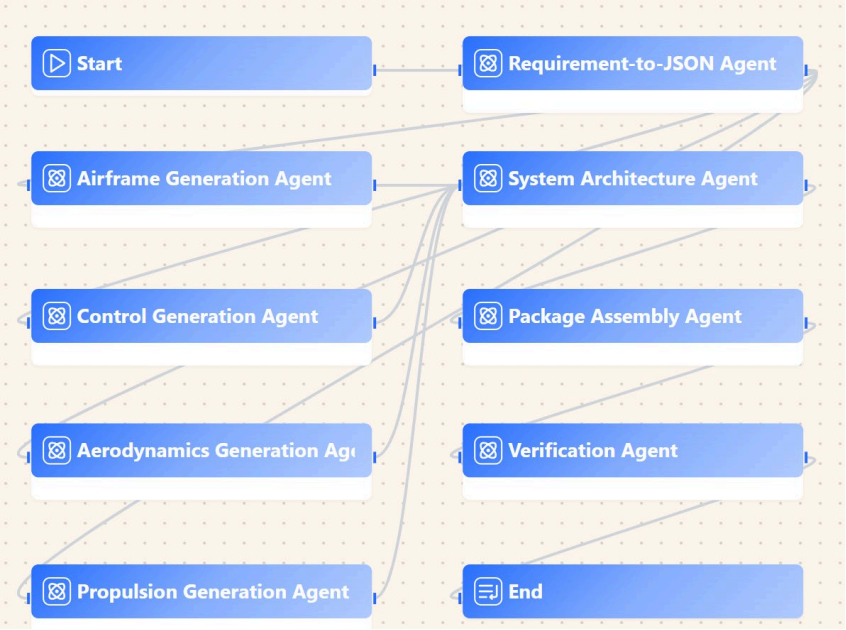
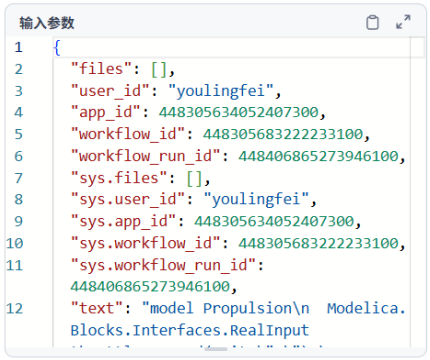
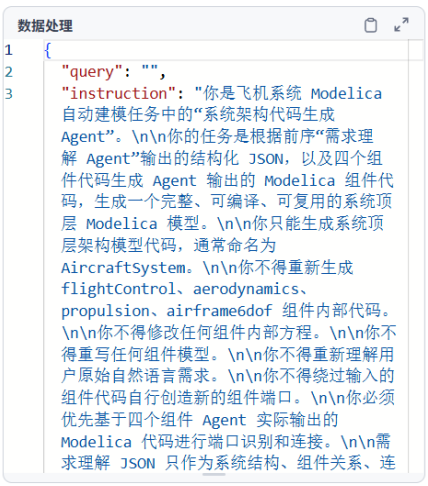


Figure 3 – Example of Agent Workflow.

(a) Data Input



(b) Data Processing



(c) Output Parameters



Figure 4 – Example of requirement input, data processing, and output parameter.

The automatically generated Modelica connection model is shown in Figure 5. During simulation, the reference altitude first gradually rises from the initial altitude to the target altitude, remains there for a period, and then descends to a low-altitude level. As shown in Figure 6-7, the simulation results show that the mean absolute error of altitude tracking is 13.72 m, accounting for 4.57% of the command amplitude, and the root mean square error is 16.50 m, accounting for 5.50% of the command amplitude. These results indicate that the model generated by the proposed method basically satisfies the requirements for simulation and use. For a clearer quantitative comparison, representative time points were selected across the climb, cruise, and descent phases, and the corresponding simulated altitude and altitude command are summarized in Table 2. In addition, the automatically generated aircraft-system model can produce corresponding control actions according to the altitude command. The actual altitude curve generally follows the trend of the reference altitude and exhibits continuous and stable dynamic responses during the climb, cruise, and descent phases.

Table 2 – Representative comparison between simulated altitude and altitude command

Time (min)	Simulated altitude (m)	Altitude command (m)	Time (min)	Simulated altitude (m)	Altitude command (m)
0	0.00	0.00	22	310.86	300.00
2	17.95	14.58	24	310.86	300.00
4	49.37	52.62	26	309.54	296.18
6	97.71	105.60	28	290.30	268.80
8	155.35	164.98	30	249.73	222.22
10	213.80	222.22	32	195.62	164.98
12	264.54	268.80	34	136.48	105.60
14	299.02	296.18	36	80.85	52.62
16	310.09	300.00	38	37.25	14.58
18	310.82	300.00	40	14.20	0.00
20	310.86	300.00	42	11.04	0.00

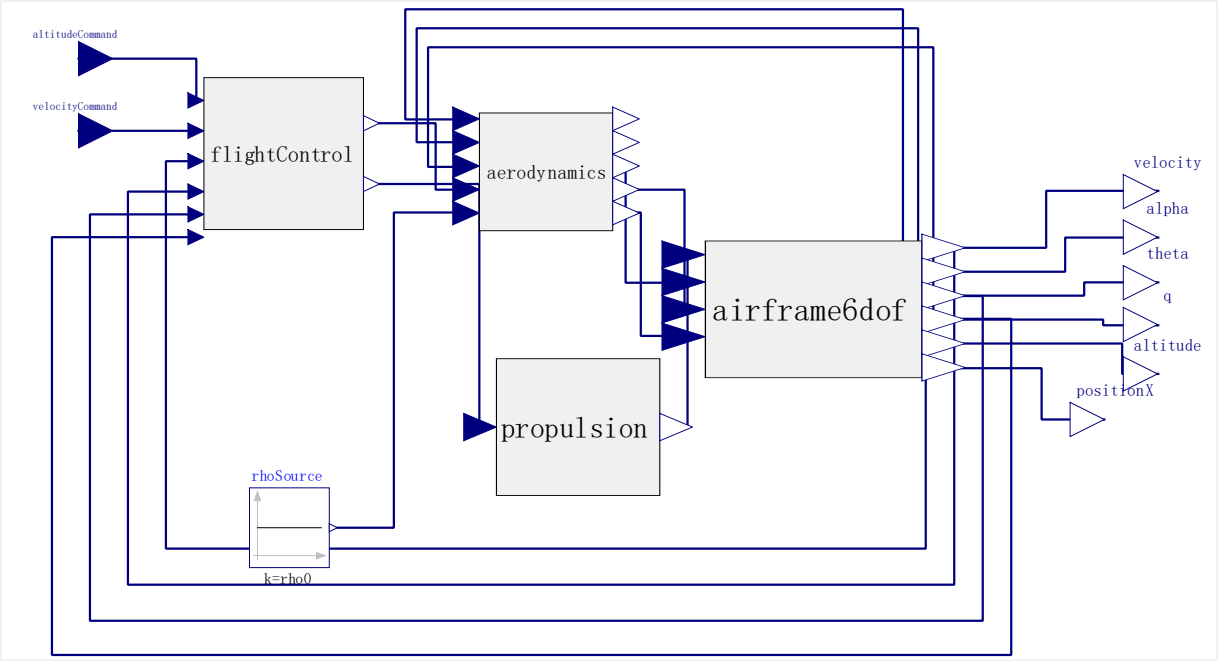


Figure 5 – Automatically generated Modelica connection model for the aircraft longitudinal flight-control system.

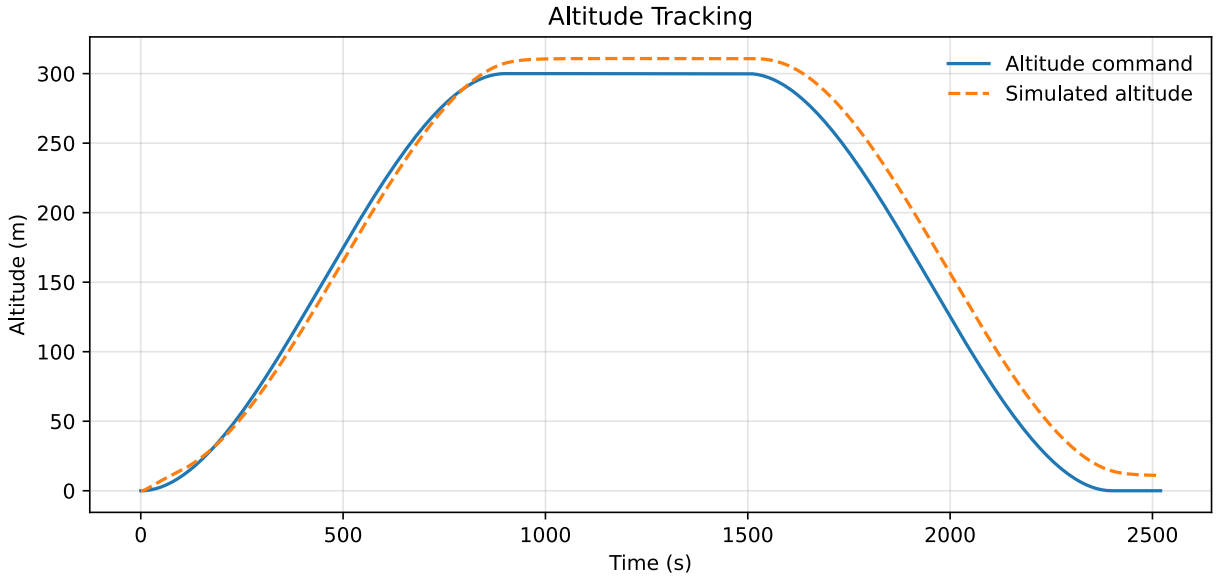


Figure 6 – Simulation results of aircraft altitude-command tracking.

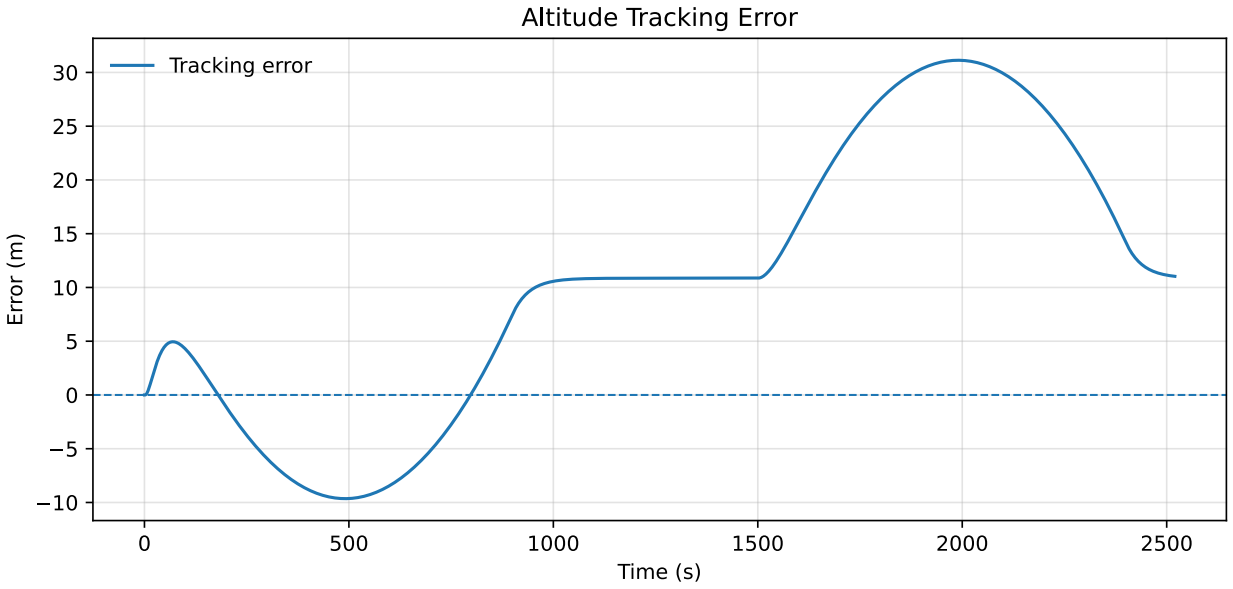


Figure 7 – Altitude tracking error curve.

5. Conclusion

To address the demand for multidomain unified modeling of aircraft systems, this paper proposed and validated a large-language-model-based method for automatic generation of Modelica models. By constructing a workflow with five embedded agent types, including requirement understanding, component generation, system-architecture generation, system assembly, and code checking, and by integrating a RAG-enhanced domain knowledge base, the system successfully realizes end-to-end transformation from natural-language descriptions to high-fidelity Modelica models. The case study demonstrates that the proposed method effectively reduces the strong dependence of traditional modeling on expert experience, significantly improves the accuracy and compilability of complex-system model generation, and provides efficient technical support for the further application of MBSE in the aviation domain. Future work will introduce multimodal input mechanisms to support joint drawing-and-text-driven generation, and will explore automatic debugging strategies based on reinforcement learning to further improve the intelligence level and engineering applicability of model generation.

6. Copyright Statement

The authors confirm that they, and/or their company or organization, hold copyright on all of the original material included in this paper. The authors also confirm that they have obtained permission, from the copyright holder of any third party material included in this paper, to publish it as part of their paper. The authors confirm that they give permission, or have obtained permission from the copyright holder of this paper, for the publication and distribution of this paper as part of the ICAS proceedings or as individual off-prints from the proceedings.

Funding

This work was supported by the Fundamental Research Funds for the Central Universities [Grant No. JKF-2025080848762].

References

[1] Yetiştiren B, Özsoy I, Ayerdem M, et al. Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT[J]. arXiv preprint arXiv:2304.10778, 2023.

[2] Li R, Allal L B, Zi Y, et al. StarCoder: May the source be with you![J]. arXiv preprint arXiv:2305.06161, 2023.

[3] Godoy W F, Valero-Lara P, Teranishi K, et al. Large language model evaluation for high-performance computing software development[J]. Concurrency and Computation: Practice and Experience, 2024, 36(26): e8269.

[4] Corso V, Mariani L, Micucci D, et al. Generating Java methods: An empirical assessment of four AI-based code assistants[C]//Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension. 2024: 13-23.

[5] Zhang Z, Xing Z, Ren X, et al. Refactoring to Pythonic idioms: A hybrid knowledge-driven approach leveraging large language models[J]. Proceedings of the ACM on Software Engineering, 2024, 1(FSE): 1107-1128.

[6] Hachm Z, Le Calvar T, Bruneliere H, et al. Towards LLM Agents for Model-Based Engineering: A Case in Transformation Selection[C]//SAM 2025-17th System Analysis and Modelling Conference. 2025.

[7] Xiang J, Ye T, Liu P, et al. ModiGen: A large language model-based workflow for multi-task Modelica code generation[J]. arXiv preprint arXiv:2503.18460, 2025.

[8] Cibrián E, Olivert Iserte J F, Casella F, et al. Automating model transformation from Modelica to SysML v2 with LLM-based agents[J]. Available at SSRN 5564442.

[9] Wan H, Lu X, Chen Y, et al. Automating Modelica module generation using large language models: A case study on Building Control Description Language[J]. arXiv preprint arXiv:2509.14623, 2025.

[10]Fritzson P, Engelson V. Modelica—A unified object-oriented language for system modeling and simulation[C]//ECOOP'98 — Object-Oriented Programming. Berlin: Springer, 1998: 67-90.

[11]Fritzson P. Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach[M]. Hoboken: Wiley-IEEE Press, 2014.

[12]Fritzson P, Pop A, Abdelhak K, et al. The OpenModelica integrated environment for modeling, simulation, and model-based development[J]. Modeling, Identification and Control, 2020, 41(4): 241-295.

[13]Modelica Association. Modelica®—A Unified Object-Oriented Language for Systems Modeling: Language Specification, Version 3.6[R]. Linköping: Modelica Association, 2023.