




Article

Methodology to Develop a Discrete-Event Supervisory Controller for an Autonomous Helicopter Flight

James Horner¹, Tanner Trautrim¹, Cristina Ruiz Martin^{1,*}, Iryna Borshchova^{2,*} and Gabriel Wainer^{1,*}

¹ Department of Systems and Computer Engineering, Carleton University, Ottawa, ON K1S 5B6, Canada; jameshorner@cmail.carleton.ca (J.H.); tannertrautrim@cmail.carleton.ca (T.T.)

² Aerospace Research Centre, National Research Council of Canada, Ottawa, ON K1A 0R6, Canada

* Correspondence: cristinaruizmartin@sce.carleton.ca (C.R.M.); iryna.borshchova@nrc-cnrc.gc.ca (I.B.); gwainer@sce.carleton.ca (G.W.)

Abstract

The National Research Council Canada (NRC) is actively engaged in the development of an advanced autonomy system for the Bell 412 helicopter. This system's capabilities extend to the execution of complex missions, such as arctic resupply missions. In an arctic resupply mission, the helicopter autonomously delivers supplies to a remote arctic base. During the mission it performs tasks such as takeoff, navigation, obstacle avoidance, and precise landing at its destination, all while minimizing the need for pilot intervention. The complexity of this autonomy system necessitates the inclusion of a high-level supervisory controller. This controller plays a critical role in monitoring mission progress, interacting with system components, and efficiently allocating resources. Conventionally, supervisory controllers are embedded within monolithic programs, lacking transparent state flows. This causes system modification and testing to be a significant challenge. In our research, we present an innovative approach and methodology to develop supervisory controllers for autonomous aircraft on the example of the NRC Bell 412. Using the Discrete Event System Specification (DEVS) formalism and the Cadmium simulation engine, we effectively address the challenges above. We discuss the entire development process for a state-based, event-driven supervisory controller for autonomous rotorcraft using the NRC's Bell-412 autonomy system as a comprehensive case study. This process includes modeling, implementation, verification, validation, testing, and deployment. It incorporates a simulation phase, in which the supervisor integrates with components within a Digital Twin of the Bell 412, and a real-time operations phase, where the supervisor becomes an integral part of the actual Bell 412 helicopter. Our method outlines the smooth transition between these phases, ensuring a seamless and efficient process.



Received: 19 June 2025

Revised: 22 September 2025

Accepted: 2 October 2025

Published: 10 October 2025

Citation: Horner, J.; Trautrim, T.; Ruiz Martin, C.; Borshchova, I.; Wainer, G. Methodology to Develop a Discrete-Event Supervisory Controller for an Autonomous Helicopter Flight. *Aerospace* **2025**, *12*, 912. <https://doi.org/10.3390/aerospace12100912>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

Keywords: DEVS; autonomous flight; supervisory controller

1. Introduction

The Canadian Vertical Lift Autonomy Demonstration (CVLAD) project aimed at developing an autonomy system for the National Research Council Canada's (NRC's) Bell-412 Advanced Systems Research Aircraft (ASRA), equipped with advanced fly-by-wire capabilities [1]. This project's main goal was to conduct an arctic resupply mission where the helicopter would autonomously transport supplies, navigate while avoiding obstacles, and land at its destination, all while minimizing the need for pilot intervention.

Figure 1 offers a high-level component diagram of ASRA which is, composed of several distinct software modules, each playing an important role in the execution of autonomous

missions. The modules are as follows: the Mission Planning Software (Element 1), the Autonomy Core (Element 2), the Autonomy Guidance (Element 3), and the LIDAR-based Landing Zone Evaluation module (Element 4).

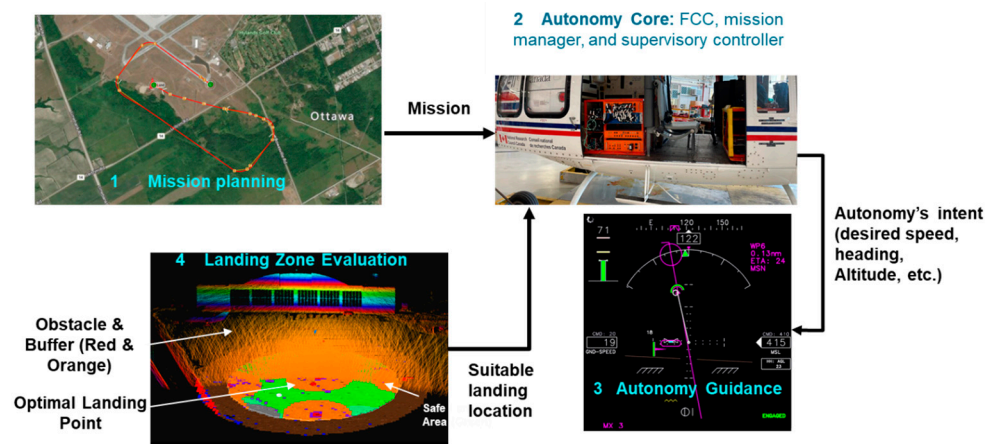


Figure 1. CVLAD autonomy system overview.

The Mission Planning Software serves as the launchpad for setting up autonomous helicopter missions. It allows a user to specify the takeoff location, create flight plans, and define desired speeds and altitudes. After the mission is planned, it is transferred to the Autonomy Core, which serves as the central hub and is the core of CVLAD's autonomy system.

The Autonomy Core encompasses a Flight Control Computer (FCC), a Mission Manager, and the Supervisory Controller. The FCC takes on the role of communicating digital control commands to the helicopter's flight control system and is essentially an inner-loop controller and an advanced autopilot system. Simultaneously, the Mission Manager is entrusted with overseeing mission execution including desired speed, heading, altitude, and more.

Once the helicopter reaches the landing leg, the Landing Zone Evaluation module selects the safest landing location. After determining the most suitable position, the Autonomy Core executes an autonomous landing.

Given the complexity of the Bell 412 autonomous system, which comprises numerous subsystems, we need a high-level supervisory controller, referred to as the "Supervisor". In this research, we integrated a Supervisor as a key component of the Autonomy Core that plays a crucial role in ensuring the desired system's state flow from takeoff to landing, overseeing the status of all the components, and managing resources within the helicopter. This is essential since multiple subsystems may contend for shared resources, such as control authority—e.g., both the FCC and the pilot may seek to control the aircraft simultaneously.

There are numerous methods for the design and synthesis of supervisory controllers in industrial control systems. Such traditional techniques suffer from several drawbacks. Supervisory controllers are often monolithic programs with little separation of concerns applied, resulting in tight coupling between elements within the controller, as well as a lack of explicit states and transitions between states. A tight coupling between components of any system can lead to difficulty performing modifications as well as extensive re-testing of the whole system when one small aspect of the program is changed. In general, the development of supervisory controllers may not consider future development: what starts as small procedural programs intended to fulfill a single purpose, may grow into complex and unmanageable systems where legacy code is intertwined with new features, such that any change, e.g., adding an extra state, might have far-reaching effects.

This research introduces an innovative approach to developing supervisory controllers that prioritize modifiability and transparency by using Discrete Event System Specifications (DEVS) [2] and the Cadmium library [3]. The use of the DEVS formalism to develop and design supervisory controllers for aircraft is novel within the aerospace field. DEVS offers the ability to effectively apply separation of concerns in the development of controllers, such that behavior is compartmentalized. Subsets of the supervisory controller behavior can then be expanded upon in future (e.g., adding detect-and-avoid, path planning, etc.) without impacting the entire system and can be unit tested separately before being integration tested. We demonstrate the practical application of this method by presenting a comprehensive development cycle for a supervisory controller tailored to a concrete and realistic use case—the NRC’s Bell-412. This method can be extended to other aviation systems like drones and other types of aircraft. The development process encompasses the entire autonomous helicopter mission, spanning from takeoff to landing.

To meet the demands of aerospace applications, our DEVS-based supervisory controller is designed to address several domain-specific constraints. First, autonomous missions require precise timing for many critical transitions such as landing and other maneuvers; DEVS inherently supports time-aware execution, allowing each state to define its duration and ensuring proper time management of the transitions. Second, the complexity of autonomous helicopter missions (including takeoff, en-route navigation, and landing) can benefit from a modular design, which is not generally available in these kinds of supervisory applications. The use of DEVS enables the formalization of the actual supervisor into a set of formal mathematical entities that permit better analysis, design, and execution. DEVS allows the decomposition of complex models like the supervisory controller into behavioral models (called atomic) and structural models (called coupled). These models can be formally defined and analyzed prior to implementation. DEVS execution engines are based on the concept of closure under coupling, and it is guaranteed that the execution of such models is correct provided that they follow the formal system specifications. This provides a major benefit, allowing each mission phase to be modeled independently and integrated hierarchically, which improves maintainability and scalability. Third, aerospace systems often rely on diverse communication protocols. To ensure seamless integration, the supervisor uses interface atomic models to abstract communication mechanisms (e.g., UDP and shared memory), thereby supporting robust and flexible interaction with other autonomy system components.

Key innovations of this paper can be summarized as follows:

1. Application of the DEVS formalism to the helicopter supervisory control: while DEVS has been used in other domains, its application to supervisory control in aerospace—particularly for autonomous helicopter missions—is unprecedented. This work demonstrates how DEVS can be used to model, simulate, and deploy a modular, extensible supervisory controller that meets real-time flight constraints.
2. End-to-end lifecycle integration: the proposed methodology includes the entire development lifecycle—from graphical modeling using DEVS-graphs to real-time deployment on flight hardware—without requiring model transformation or reimplementa-tion. This continuity between simulation and deployment is a significant advancement over traditional approaches.
3. Modular and extensible architecture: the supervisory controller is designed to be aviation platform-agnostic and behaviorally modular. It supports integration of new autonomy components (e.g., detect-and-avoid, path planner) and adaptation to different aircraft types, which is a major step toward scalable autonomous solutions in aerospace systems.

4. Validation on a real helicopter platform: the methodology was not only simulated but also deployed and tested on the NRC's Bell 412 helicopter, demonstrating its practical viability. This real-world validation distinguishes the work from purely theoretical or simulation-based studies.

The paper is organized as follows: Section 2 presents the background on supervisory controllers for autonomous systems. Section 3 explains the Supervisor development methodology, including the DEVS formalism. Section 4 focuses on the Supervisor model development and Section 5 focuses on the implementation of those models. Section 6 describes the interface development to integrate with other autonomy system components. Section 7 explains how the Supervisor was tested in simulation and deployed in the helicopter. Section 8 presents the conclusions and future work of this research.

2. Background

A discrete-event system (DES) is a discrete-state, event-driven system of which the state evolution depends entirely on the occurrence of asynchronous discrete events over time [4]. Discrete-event supervisory control introduced by [5] provides a discrete-event control mechanism that is executed by forcing or delaying specific events, to ensure the desired system's state flow. In short, the control mechanism intends to prevent the system from entering an "unacceptable" state by speeding up or slowing down the state transitions. An example of an "unacceptable" state in the context of autonomous flight could be landing on an obstacle if a suitable landing location is not found or running out of fuel while conducting an autonomous mission.

DES has been known for supervisory control for a long time [6]; however, it only became popular in autonomous applications recently, due to the increasing complexity of these systems [7]. When the system is relatively simple (i.e., because it performs only one task), a separate supervisory controller is redundant since the inner-loop controller can execute any control mechanism. On the other hand, when the system is complex and consists of several components that interact with each other, we need a high-level controller to monitor the inner states of each component, derive its own state, and make decisions accordingly. Such a controller can be essentially represented using a state machine that depicts the desired system's state-flow to ensure each component executes in a timely manner. The desired states can be pre-programmed (desired state transitions are established "offline") or derived in real-time (desired state transitions are established "online").

Several examples of autonomous systems where a discrete-event controller ensures the desired state flow, are given below. In [8], the authors developed an autonomous system for driving in urban environments; they proposed the system structure and method of self-driving cars consisting of three main parts: perception, planning, and control. Each part was designed to recognize the real-time driving environment, make an action plan based on a finite state machine, and activate kinematic model-based control. In [9] the authors presented a systematic and mathematical design procedure for a hybrid state system-based controller for the intelligent mission management of an Unmanned Aircraft System. The proposed controller utilizes a discrete-event system flight executive based on a finite-state machine for high-level decision-making and a continuous-state controller for the lower-level autopilot. The flight executive and autopilot are integrated together to form the hybrid state controller. An example of supervisory control focusing specifically on the landing phase of a flight is described in [10]. The authors presented a visual servoing method for autonomous multi-rotor landing, where the behavior of the multirotor during the whole landing procedure was handled by a finite state machine.

An additional instance of a supervisory controller applied during the landing phase can be found in Borshchova's work [11]. Here, they developed a discrete-event supervisor

that functioned alongside the inner-loop controller. In their research, the landing phase was represented as a Time Transition Model (TTM) and was tested in real-time to help the pilot and crew in handling exceptions during the landing phase. However, a significant challenge arose due to state explosion issues associated with the use of TTMs, making real-time implementation problematic.

Although there have been many advances in the field of DES, control engineers working on autonomous applications might lack experience with modeling and specification frameworks and software expertise in engineering design, resulting in monolithic programs with little separation of concern and tightly coupled components. Typically, the research documentation on supervisory controllers (including described above) does not give sufficient details of software implementations, making the reader speculate that their finite-state machines do not include states implicitly, but rather use “if-else” conditions to implement the desired logic. In general, the development of supervisory controllers does not seem to consider future development: what starts as small procedural programs intended to fulfill a single purpose, may grow into complex and unmanageable systems where legacy code is intertwined with new features, such that any change, e.g., adding an extra state, might have far-reaching effects. This makes it challenging to transfer the supervisors onto different platforms, use various sub-components interchangeably, add other behaviors/models, and re-test the logic. From the CVLAD perspective, if the supervisory controller is not designed to support modularity from the beginning of its development cycle, it might be very challenging to modify the software to be able to apply it on other aircraft platforms, since the state machine will be significantly different (e.g., helicopters can hover over the desired landing point, while fixed-wing aircraft must join the glide slope).

To fulfill these gaps and address the challenges discussed above, we use DEVS [2] and the Cadmium DEVS environment [3] to provide an alternate approach to building supervisors, focusing on modifiability and transparency. DEVS was chosen to model the CVLAD supervisory controller because it addresses the current challenges with supervisory controllers. DEVS is a hierarchical and modular modeling formalism that separates models and their execution engines. A target system can be built as a composite of atomic and coupled model components that can be linked to each other’s inputs and outputs allowing complex models to be built from simple building blocks. The communication between models is performed through instantaneous occurrences where values are transmitted to or received by a DEVS model. Modularity allows the behavior of large systems to be divided and modeled independently, increasing cohesion within components. The hierarchical nature of DEVS means that small components can be assembled to model much larger systems (such as CVLAD’s autonomy system). Modeling systems hierarchically and in a modular fashion allow sections of the model to change without affecting the whole model as well as partitioning behavior into logical blocks which can more easily be understood. This allows the models used in the simulation engine to be deployed on the helicopter without modification. Another advantage of DEVS is that the notion of “time” is built into the framework. Compared to less sophisticated and attractive at first glance methods like the QT library [12], DEVS allows for an easy approximation of complex dynamic discrete-event systems.

A graphical specification (DEVS-graphs) can be used as described in specifying DEVS models [13]. The main benefit of the DEVS-graphs notation is that it allows for greater interaction and clearer communication with stakeholders who may have limited knowledge of formalisms, mathematical notation, and programming. Additionally, it assists the modeler with visualizing the system description. These benefits result in a developed system that accurately represents the stakeholders’ needs. Notations exist for modeling both atomic and coupled DEVS models. For atomic models (Figure 2), this notation is similar to

a finite state machine: there are nodes (representing the states) connected by directed edges (representing the transitions), though several key extensions have been made.

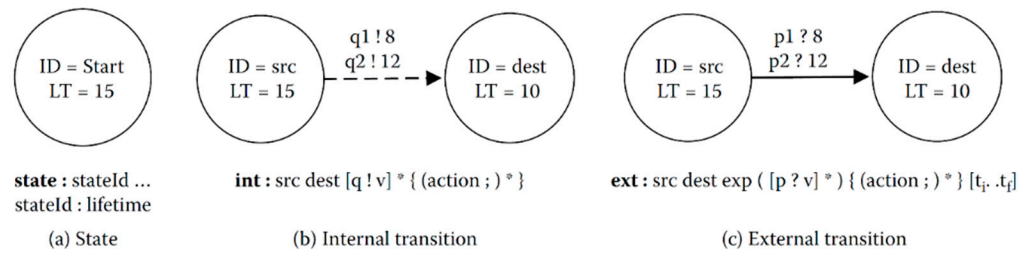


Figure 2. DEVS atomic models using DEVS-Graphs [14].

As shown in Figure 2, the edges connecting nodes can be of two types: a dashed line to represent an internal transition or a solid line to represent an external transition. Some additional annotations are required to fully define the model: (1) each state must be labeled with an ID and a lifetime (LT) associated with the state; (2) each internal transition must be labeled with any outputs to ports and the associate output values that are generated when the internal transition fires; and (3) each external transition must be labeled with the input port and value pair that must be received in order for the transition to occur.

Coupled models can then be built up by connecting the inputs and outputs from atomic models to each other. It is useful in this circumstance to hide the structure of the atomic model inside a black box (Figure 3) and focus on the interfaces of the model when constructing coupled models. Atomic and coupled models are then connected together using arrows, the ends of which specify the sending or receiving port.

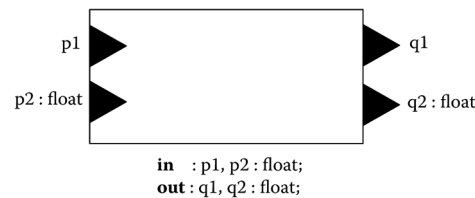


Figure 3. DEVS coupled models using DEVS-Graphs [14].

Although there are many DEVS simulators [15], we used Cadmium, a DEVS modeling and simulation engine that allows us to complete the whole development cycle (including embedded real-time execution) without modifying the original models. In Cadmium each atomic and coupled model are represented by a class, each of which can be instantiated into an object and then incorporated into larger coupled models [16].

Atomic models in Cadmium must contain: (1) a structure containing the input and output definitions for the ports, (2) a tuple with the input port types, (3) a tuple with the output port types, (4) a default constructor, (5) a function to handle internal transitions, (6) a function to handle external transitions, (7) a function to handle an internal and external transition occurring at the same time (confluence function), (8) a function to send outputs and (9) a function to manage the timing of each state. Coupled models in Cadmium must contain: (1) a structure containing the input and output definitions for the ports, (2) a Ports object with the input port types, (3) a Ports object with the output port types, (4) a Models object with the atomic/coupled models in the current coupled model, (5) an EICs object to define the external to internal couplings, (6) an EOCs object to define the internal to external couplings, and (7) a ICs object to define the internal couplings. Note that classes for all the objects are defined within the Cadmium simulator.

RT-Cadmium [17] allows executing Cadmium models in real time. The models are defined in the same way as in Cadmium, and the models execute based on the real-time clock instead of using virtual time. To connect with external devices, the I/O ports used for the DEVS models use an interface and drivers. The user models, the drivers and RT-Cadmium libraries are compiled to produce an executable that runs on different hardware platforms. A modeling subsystem is connected to runtime and messaging subsystems. The main runtime subsystem manages the overall aspects of the real-time execution and provides timing functions with microsecond precision. It controls atomic components, the top coupled component ports that are connected to the external environment and uses the models to build a hierarchy. Finally, it starts the main real-time task. The runtime subsystem includes simulators (that execute the atomic component functions in real-time), a root coordinator (that handles real-time event scheduling, and spawns drivers), and coordinators, in charge of message passing and scheduling of the subcomponents. The messaging subsystem is in charge of transmitting messages between the different components in the runtime subsystem, which makes the model execution advance (in virtual or real-time).

Other DEVS environments, both for simulation and real-time execution include [18], as well as [8], who defined real-time models using the DEVS framework. In [19], the authors showed how to reuse models developed in different simulation engines by interfacing E-CD++ [20] and PowerDEVS [21]. PowerDEVS provides a method to model hybrid systems and execute RT models. Action-Level Real-Time DEVS [22] is used to model Network-on-Chip systems. One of the advantages of RT-Cadmium is that the models can run on bare hardware, without the need for an operating system or other middleware, making modular classes simple to be reused.

3. Methodology

We propose to use a spiral project lifecycle with evolutionary prototypes [23] to develop supervisory controllers. Alongside the proposed spiral lifecycle, decomposing the supervisory controllers into smaller modular components allows for easier management of individual work items. Each modular sub-component of the supervisory controller can be iterated upon until each is fully verified and ready for integration. Figure 4 shows a diagram of the lifecycle and demonstrates how validation is the key driver in iterating upon a given work item, ensuring the final deliverables' acceptance.

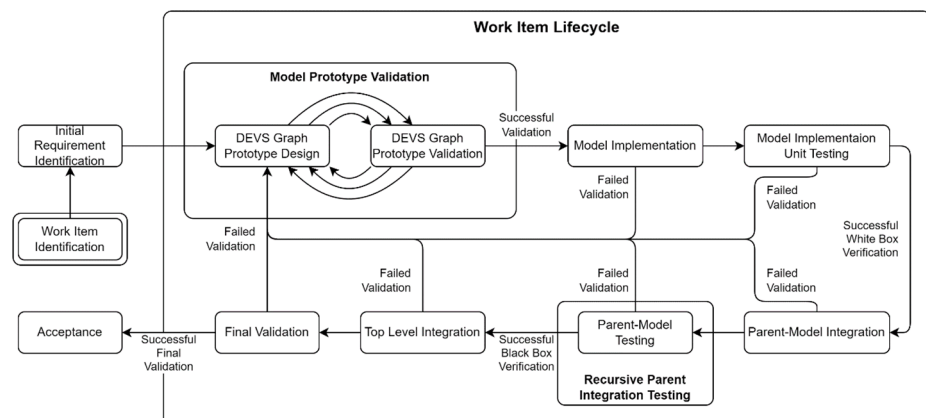


Figure 4. Project Lifecycle for Each Identified Work Item.

First, the work items that will combine to form the supervisory controller need to be identified. These work items can be identified through different methods: initially from elicited requirements [24] and existing programs, then later through decomposition of su-

pervisory controller components into smaller encapsulated and decoupled subcomponents. The use of DEVS greatly helps in the identification of work items as models that grow too complex or lose encapsulation of one specific behavior can be decomposed hierarchically into interacting sub-models.

Secondly, requirements are gathered for the identified work item. Requirements can be identified through elicitation from stakeholders or domain experts, in the case of new programs, or can come from the partitioning of a larger model's requirements into sub-models [24]. Requirements for a work item are not necessarily static and can increase in fidelity with each iteration.

With an initial set of requirements, a DEVS Graph prototype should be created that satisfies the understood requirements [23]. DEVS Graphs is an invaluable tool in prototyping DEVS models. Due to the illustrative nature and ample tool support for creating diagrams, DEVS graphs can be drafted rapidly compared to other formal specification methods. By increasing the speed at which prototypes can be produced, the time between prototype iterations is reduced.

Once a DEVS Graph prototype is defined, the prototype can be presented to stakeholders for validation [25]. By examining the prototype with the stakeholders, the behavior can be easily explained and validated against the stakeholders' requirements [24]. Small adjustments to the DEVS Graph can be made in real-time as the prototype is validated for rapid iteration that is informed by stakeholder expertise.

If larger modifications to the prototype are required, the prototype can be revised and then presented to stakeholders later. Following a cyclical process of requirements elicitation, modelling, and validation at this stage provides multiple benefits. While the prototype model is being refined by the implementor using the latest requirements, domain experts have time to process the current design potentially revealing unforeseen requirements that can be added to the design early on in the development process saving time and resources. This process of requirements elicitation, modelling, and validation should be repeated until all requirements are met and the domain experts are confident that the generated requirements are sufficient. Only once the domain experts are satisfied with the set of requirements and how they are being met should this process be concluded. It is important to remark that all this process is done without the need to implement any part of the software.

Once the stakeholders approve the prototype, the DEVS Graph can be implemented directly into a C++ class using the Cadmium library [3]. As the DEVS Graph prototype is already a direct representation of the DEVS model and Cadmium provides a one-to-one implementation of DEVS models in C++, the implementation of prototypes into classes is trivial as it is a direct translation of the DEVS Graph into C++ code.

After we implement the prototype using Cadmium, the implementation should be unit tested [25]. Test drivers can be constructed using Cadmium to harness onto the implementation and iteratively run through a test suite. Test cases for the implementation can be derived from the stated requirements and from the DEVS Graph prototype. After running the test suite, trajectories and events must be analyzed to verify that the model exhibited behavior that met all the stated requirements. Additional validation with stakeholders can occur at this point to ensure that the implementation does not miss any requirements.

Upon successful unit testing, the implementation can be integrated into any larger Cadmium models. If the model was incorporated into a multi-level coupled model, integration testing should be conducted recursively on all higher-level models. For each higher-level model, any existing test cases should be reviewed and modified to consider the new model's behavior. All test drivers should be re-run, and subsequent faults are to be identified and addressed in all affected models.

After the new model is fully verified using integration and unit testing, the model can be integrated into the supervisory controller, at which point a validation demonstration can be performed. By validating the supervisory controller alongside stakeholders, erroneous or missing behaviors in the system as a whole can be identified and remedied in further iterations or work items.

The above process for the development of supervisory controllers presents several benefits. Through regular validation of evolutionary prototypes against stakeholder requirements throughout the lifecycle, concerns regarding missing or previously unknown requirements can be quickly addressed before development proceeds too far [23]. By testing new models at each level, the unexpected consequences of integration can be reduced and faults in model design and implementation can be addressed quickly [25]. By iterating on work items until a final satisfactory validation is performed, it can be guaranteed that the behavior of the supervisory controller was incrementally improved by the development.

It is important to remark that the supervisory controller we developed using this process was the same implementation that was deployed into the specific application. As the Cadmium DEVS models could be run in simulated or real-time as well as with simulated or real stimuli, the models developed using this process could be deployed directly onto the helicopter. To develop the application-specific interfaces for the Bell-412 autonomy system, a similar process was used with minor variations as requirements stemmed less from stakeholder requirements and more from previously defined interfaces that the Supervisor had to adhere to. It is important to remark that the interfaces were independent from the Supervisor: shall the interface of a component change, the supervisor does not need to be modified, just the interface.

This approach is readily extensible to other aircraft platforms, as the Supervisor remains agnostic to specific interface implementations. By adapting only the interface modules, the same supervisory logic can be reused across different aircraft systems. Additionally, improvements to the Supervisor—such as inclusion of Detect and Avoid and Path Planner sub-modules—can be integrated without altering the interface structure, supporting scalable and maintainable autonomy development.

4. Supervisor DEVS Model Development

In this section and the rest of the paper, we show a case study of how the methodology is applied in the test case of NRC Bell 412 autonomous helicopter. We developed a supervisory controller for the entire autonomous mission, from takeoff to landing.

As stated in the introduction, a helicopter equipped for autonomous flights has several hardware and software components. Before a flight mission starts, the mission planner designates the intended path of the autonomy system (waypoints), as well as speeds, altitudes etc. The mission planner also designates the circular area in which it is intended that the aircraft will land—this is called the Planned Landing Point (PLP). Once the aircraft approaches the PLP, the LIDAR-based landing zone evaluation system will identify Landing Points (LPs) within the PLP radius. LPs are regions large enough for the helicopter to land in and are clear of obstacles. Multiple LPs might be found by the LIDAR, so for how long LPs are sought after and which LPs are “accepted” will be the responsibility of the Supervisor. The Supervisor will also receive inputs from the FCC, mission manager, pilot, and aircraft, and determine whether the FCC should be ordered to land the helicopter at the received LP location, or to hand control over to the pilot, if no suitable landing point is found.

The following sections describe the models for the CVLAD Supervisory Controller and its sub-models.

4.1. Supervisor Coupled Model

The purpose of the Supervisor coupled model (Figure 5) is to manage the autonomous flight by receiving signals from external systems (such as the FCC, mission planner, and mission manager), processing those signals to advance the state of the system and notifying those systems back with the actions to be taken. A preliminary supervisory controller just for the landing phase of the flight was already described in [26].

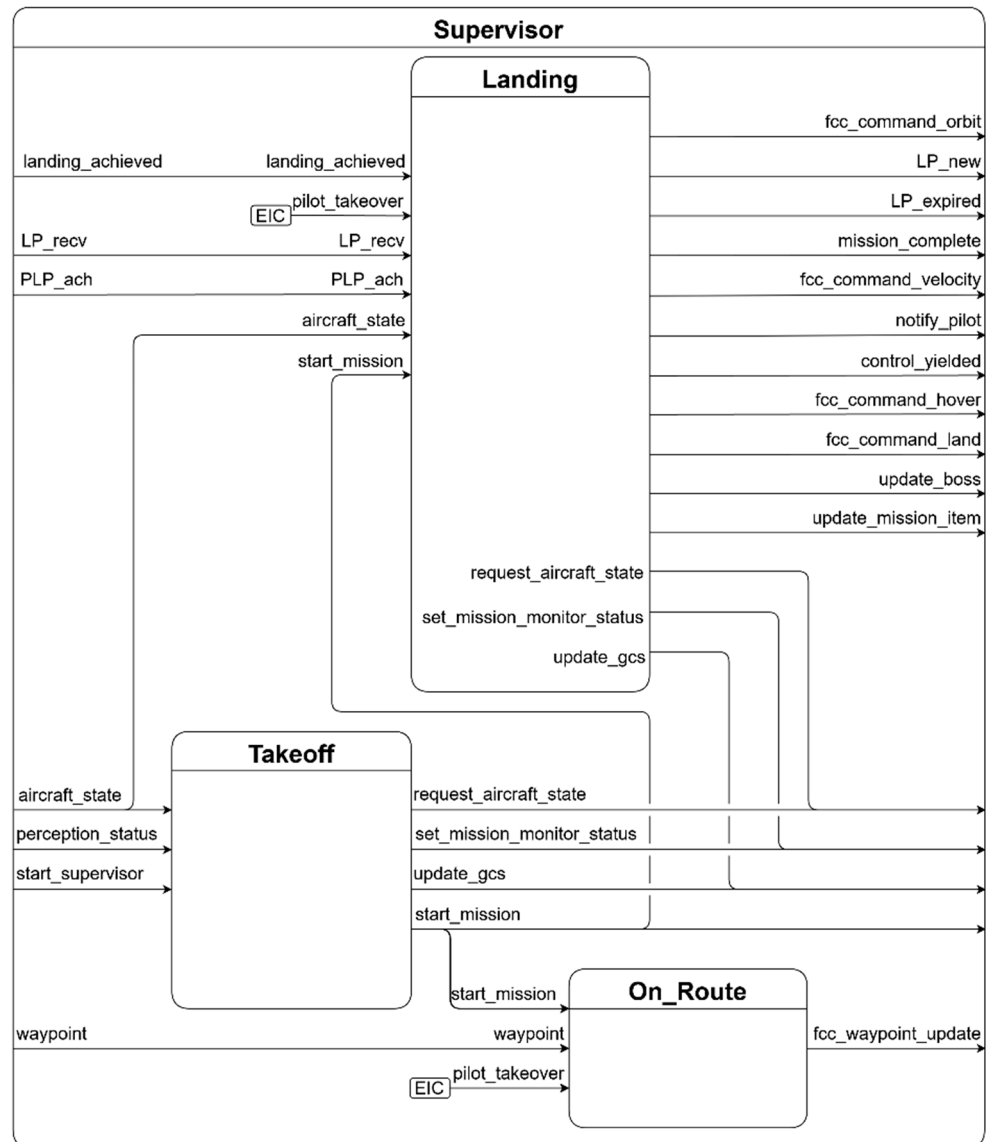


Figure 5. Supervisor coupled model.

To fully encapsulate all the behavior of the CVLAD Supervisory Controller, the Supervisor coupled model was decomposed into 3 sub-components Takeoff, On Route, and Landing. Each subcomponent encapsulates a phase of a mission:

1. The Takeoff model is used to initialize the mission, verify the status of the autonomy system prior to takeoff, and to alert the other Supervisor sub-components that the mission has started.
2. The On Route model facilitates the forwarding of mission items (e.g., waypoints) to the flight control computer as they are reached throughout the flight.

3. The Landing model defines the behavior of the Supervisor from the receipt of the last mission item until the helicopter has landed or handed control to the pilot, through several trajectories based on the availability of safe landing points.

The Supervisor coupled model (Figure 5) uses 8 input ports and 16 output ports. The input ports along with a description are presented in Table 1 and the output ports in Table 2.

Table 1. Inputs to the Supervisor coupled model.

Input Port	Description of the Inputs Received
aircraft_state	Receives the current state of the aircraft
landing_achieved	Signal indicating that the aircraft has successfully landed
lp_recv	Receives the landing point from the perception system
perception_status	Operational status of the perception system
pilot_takeover	Signal indicating that the pilot has taken control
plp_ach	Signal indicating that the planned landing point has been achieved
start_supervisor	Signal indicating the mission has started and takeoff should commence
waypoint	Receives the next waypoint in the mission to handle

Table 2. Outputs from the Supervisor coupled model.

Output Port	Description of the Outputs Sent
control_yielded	Sends an acknowledgement that the supervisor has relinquished control of the aircraft
fcc_command_hover	Sends hover command to the FCC
fcc_command_land	Sends land command to the FCC
fcc_command_orbit	Sends orbit command to the FCC
fcc_command_velocity	Sends velocity command to the FCC
fcc_waypoint_update	Sends waypoint command to the FCC
lp_expired	Sends notification that the LP accept timer has expired
lp_new	Sends new valid landing point
mission_complete	Declares the mission as being complete after landing
notify_pilot	Notifies the pilot that they should take control of the aircraft
request_aircraft_state	Requests the current aircraft state
set_mission_monitor_status	Tells the Mission Manager to stop monitoring mission progress
start_mission	Sends a notification that the mission has started
update_boss	Sends updates to autonomy guidance
update_gcs	Sends updates to the mission planning software
update_mission_item	Updates Mission Manager that the last mission item has been reached

In the following section, we discuss the definition of the Landing Point Manager atomic model, which serves as an illustrative example model. Note that all the atomic models in the system are defined in a similar way.

4.2. Landing Point Manager Atomic Model

The Landing Point Manager atomic model depicted in Figure 6 serves the purpose of receiving, evaluating, and deciding upon Landing Points (LPs). It is responsible for managing LP acceptance or rejection and making informed decisions in situations where a suitable LP cannot be located.

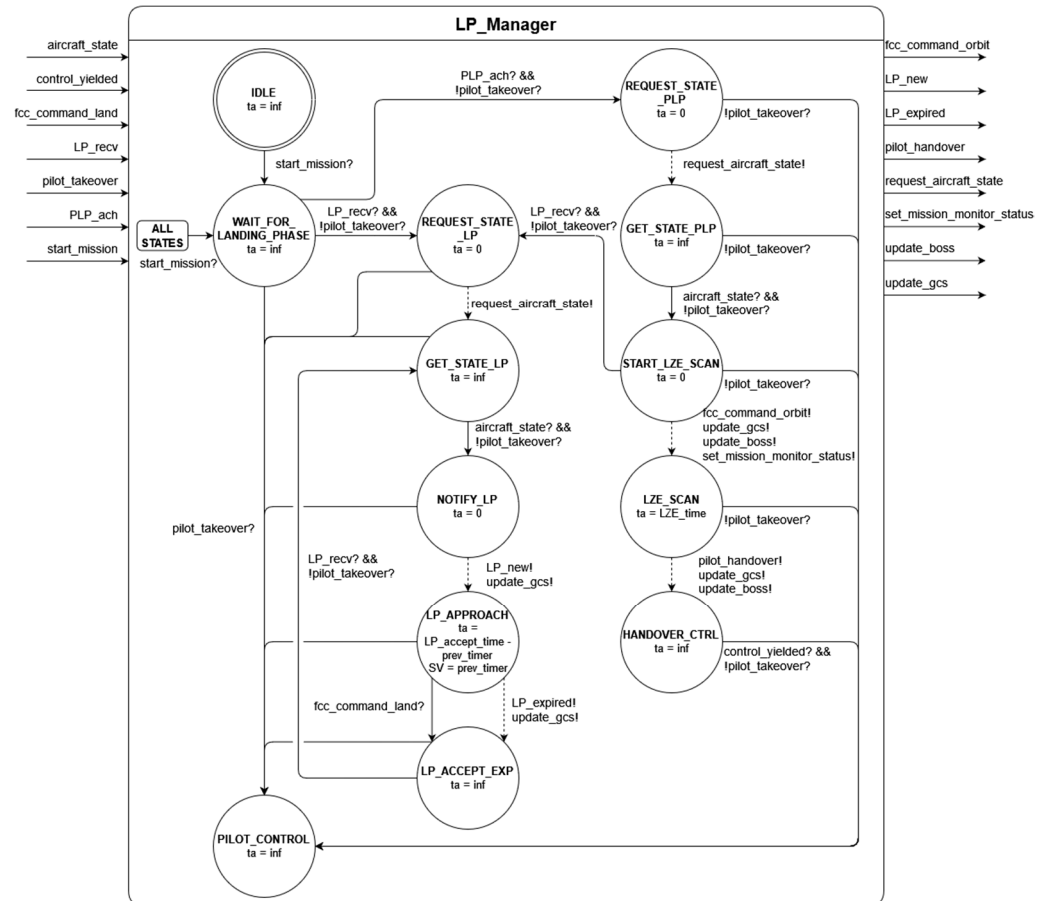


Figure 6. Landing Point Manager atomic model.

The atomic model uses thirteen states to represent this behavior: (1) IDLE, (2) WAIT_FOR_LANDING_PHASE, (3) REQUEST_STATE_PLP, (4) GET_STATE_PLP, (5) START_LZE_SCAN, (6) LZE_SCAN, (7) HANDOVER_CONTROL, (8) PILOT_CONTROL, (9) REQUEST_STATE_LP, (10) GET_STATE_LP, (11) NOTIFY_LP, (12) LP_APPROACH, (13) LP_ACCEPT_EXP. The model is initialized in the IDLE state. It remains in the IDLE state until the start_mission is received, at which point it transitions to the WAIT_FOR_LANDING_PHASE state.

The atomic model has two main trajectories. The first trajectory occurs when the PLP is achieved (a signal is received on the PLP_ach input port) before an LP is received. After the PLP is completed the Landing Point Manager will request the aircraft state (position, attitude, velocities etc.). Knowing the aircraft's state, the model requests for the helicopter to be stabilized, so the landing zone can be evaluated. If an LP is not received during the scan of the landing zone, the model will hand over control of the aircraft to the pilot (transition to the state HANDOVER_CTRL). If an LP is received while scanning the landing zone (state START_LZE_SCAN), the model will join the second trajectory.

The second trajectory occurs when an LP is received (a signal is received on the LP_rcv input port) before the PLP is achieved or if the landing zone evaluation successfully identifies an LP (state START_LZE_SCAN). After an LP is received, the model then requests the aircraft state and starts the LP_APPROACH timer. This timer allows new valid LPs to be considered for a defined duration. If the received LP is valid (located far enough from the previous LP), the model sends a new LP output using the LP_new port. Once the LP_APPROACH timer expires, the system will transition to an end state (LP_ACCEPT_EXP) and will not allow any further updates to the LP, while indefinitely waiting for the pilot to

take control (a signal sent on an output port `pilot_handover`). Further to that, the pilot can take control of the helicopter at any state (a signal received on an input port `pilot_takeover`).

5. Implementing the Supervisor

After developing the DEVS models, a simulation of the Supervisor DEVS models was implemented using the Cadmium library as well as the real-time version of Cadmium. Using the DEVS models developed earlier, test drivers were created to simulate the models for verification and validation of the behavior by the NRC experts. An evolutionary prototype life cycle was used to iterate upon the implementation, adding more functionality and implementing stubbed components with each cycle. First, a simulation of the Supervisor was created using Cadmium for verification and validation of the DEVS atomic and coupled models. The same models were then used, without modification, using a real-time version of Cadmium removing the need of performing verification and validation again. Each C++ model implementation was simulated using a test driver, then simulation results were inspected for verification and validation.

5.1. Landing Point Manager Atomic Model

The Landing Point Manager atomic model was implemented by translating the specification in DEVS Graphs into a C++ class that could be used by the Cadmium simulation library. The input and output ports of the model are defined by the `input_ports` and `output_ports` namespaces. The model structure includes a set of states (an enumeration) used to define the state variable `state_type`. The initial state of the model is `IDLE`. The `internal_transition` method uses a case statement based on the current state. The external transition method checks the inputs of the model and then chooses the transition based on the input received and the current state. The outputs from the model were defined based on the current state: message bags were constructed and sent to the output ports with information to be sent to other models within the Supervisor as well as the pilot. Finally, the time advance function for the model was defined by returning a `TIME` given the current state of the model using a switch-case statement. The remaining atomic models as well as the interface atomic models were defined in a similar manner based on the DEVS Graphs specification.

5.2. Supervisor Coupled Model

The Supervisor coupled model was implemented using the sub-models previously defined in C++ as well as the structure conveyed by the graphical specification. The sub-models of the Supervisor were built using the `make_dynamic_atomic_model` and `make_shared` functions provided by Cadmium; then, references to each model were given in the `submodels` structure. Once each sub-model was initialized, the couplings were defined: the connections between inputs to the Supervisor and the inputs of sub-models were defined in the `eics` structure, the connections between outputs of sub-models and Supervisor outputs were defined in the `eoics` structure, and the connections between outputs of sub-models and the inputs of other sub-models were defined in the `ics` structure. Other coupled models inside the Supervisor were defined in a similar manner.

6. Interface Development

To integrate the Supervisor with the rest of the autonomy system, interfaces needed to be created. Most of the aircraft's autonomy system communicates using the User Datagram Protocol (UDP), however the Supervisor was also required to interface through other means as well. To retrieve aircraft state information (e.g., latitude, longitude, altitude, air speed, pitch, etc.) an interface to a shared memory segment populated by the aircraft avionics

was required. In addition to communicating with autonomy system components using UDP, another method with more reliability in packet delivery was necessary. To avoid the overhead associated with using a TCP connection, a simple library was developed to deliver UDP packets reliably using the stop-and-wait automatic repeat request (ARQ) protocol [27]. This implementation was a practical solution tailored to the deployment environment and is not presented as a novel contribution in this paper. Using the library, a reliable interface for communication with the Supervisor was created.

6.1. UDP Output Interface

To facilitate communication between events generated by the Supervisor and the autonomy system components, a DEVS atomic model was created: the UDP Output model (Figure 7).

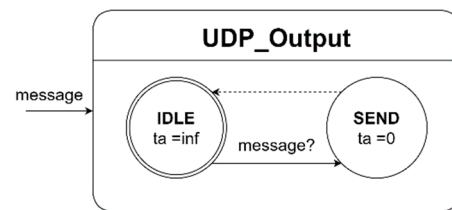


Figure 7. UDP Output atomic model.

The UDP output atomic model was used to translate an event generated by the Supervisor and send it as a UDP packet to a specified address and port. The content was sent to a predetermined network address and port instead of an external output port. The UDP Output model was designed to send a DEVS event from the Supervisor as a UDP packet to a singular autonomy system component. Multiple UDP Output models could then be combined to notify all the necessary components when a DEVS event occurs, for example, the pilot display and Mission Manager when the mission complete output is generated.

6.2. Shared Memory Interface

Onboard the aircraft, a shared memory segment is populated with the current state of the aircraft. To access the current state of the aircraft a model was required to access the shared memory segment. To avoid the overhead of polling the segment and supplying a constant stream of aircraft states, it was determined that the model would accept a request for the current aircraft state from the Supervisor, after which it would access the shared memory segment, and provide an event containing the current aircraft state, as is shown in Figure 8.

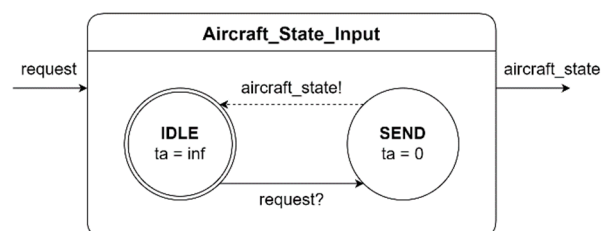


Figure 8. Aircraft State Input atomic model.

6.3. Reliable UDP Interface

Parallel to the development of the Supervisor, the requirement for reliable delivery of certain events across the whole autonomy system was identified, so a dependable UDP

library was developed for use with the Supervisor. As most network communication within the autonomy system uses UDP and the overhead of TCP was to be avoided, a library was created to reliably deliver packets over UDP. The library was implemented using the Boost ASIO C++ library [28] and utilized the stop-and-wait ARQ protocol [28] to ensure that packets were delivered reliably and without replication. As the library would no longer be compatible with all UDP interfaces (through the introduction of a new upper layer packet header), new interface models were required so the Supervisor could receive events over Reliable UDP (RUDP) as well as send events as well.

The Supervisor UDP Input model (Figure 9) was used to receive RUDP packets, demultiplex the packet to an input port of the Supervisor based on a header field, then convert the packet into an event that can be understood by the Supervisor. The input model was designed to connect to each input of the real-time Supervisor model so the Supervisor could reliably receive UDP packets from the mission manager component.

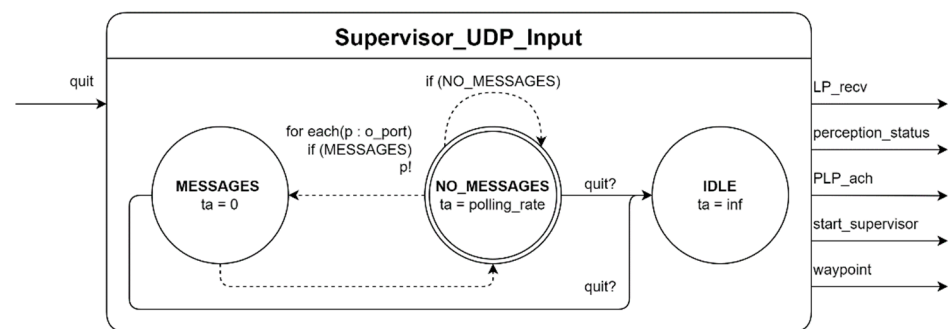


Figure 9. Supervisor UDP Input atomic model.

The RUDP Output atomic model (Figure 10) was used to translate an event generated by the Supervisor and send it as an RUDP packet to a specified address and port using the RUDP library. The model functions in much the same way as the UDP Output model by translating a DEVS event into a packet exchanged over the network.

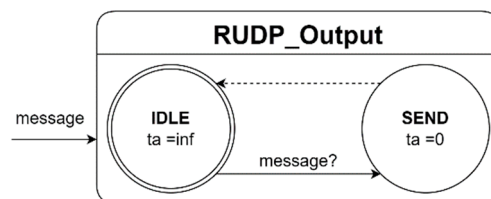


Figure 10. Supervisor RUDP Output atomic model.

7. Supervisor Model Verification and Deployment

Verification of the Supervisor was conducted throughout the project: the graphical specification of the DEVS models were visually tested with the help of the CVLAD team, then the simulated models were verified using simulated-time, interactive real-time test drivers, Bell 412 Digital Twin, and then through actual flight tests.

Simulation was chosen as the most appropriate form of verification for the project for a number of reasons. The overhead of creating simulation test cases aligned well with the development required to implement the supervisory controller, so test drivers could be written in parallel. This meant that an acceptable level of verification could be performed within the scope of the project time and budget. In addition, creating the unit test harnesses for the supervisory controller made it easier to integrate into the Digital Twin simulator later in integration testing as temporarily unavailable input interfaces could use existing

unit test harnesses. Hardware performance characteristics (CPU load, RAM usage, etc.) were not monitored during any of the simulated time testing as only the logged results of the executed trajectories were of interest. Performance was considered during real-time testing, though on a heuristic basis.

Atomic models were tested to confirm that correct transitions were made given certain inputs and time advance functions, and that outputs were generated at the correct transition. We used white box testing to evaluate the control flow of atomic models. The tests were written to evaluate the whole behavior of the system and cover all possible evolution paths. Black box testing was used to verify the coupled models. Additionally, coupled models were built by linking together atomic models already verified through white box testing.

The graphical specifications and description of DEVS atomic and coupled models (including its purpose and behavior) were provided to the stakeholders for feedback to meet the requirements. Interactive test drivers were also used for validation of the Supervisor by stakeholders. A command line Supervisor test driver allowed researchers to test if all required behavior was present as well as validate that the outputs were generated at the right time. Finally, testing on Bell 412 Digital Twin helped de-risking before taking the software into the flight test.

Due to confidentiality constraints around the NRC Bell 412 Digital Twin, this paper only presents testing in simulated time as well as actual deployment on Bell 412.

7.1. Testing in Simulated Time

Test drivers were created to test the behavior of each simulated atomic and coupled DEVS model. Each test driver consisted of an input reader (provided by the Cadmium library) coupled to each input of the model under test to read input events. The state changes and events were then recorded using loggers provided by Cadmium. The logs of each simulation run could then be analyzed to determine whether the model under test was exhibiting the required behavior.

Figure 11 shows an example test driver (for the Landing Point Manager atomic model shown in Section 4.2 and implemented in Section 5.1). The model consists of seven input readers, named as the port the reader is coupled to with “ir_” (input reader) prefixed.

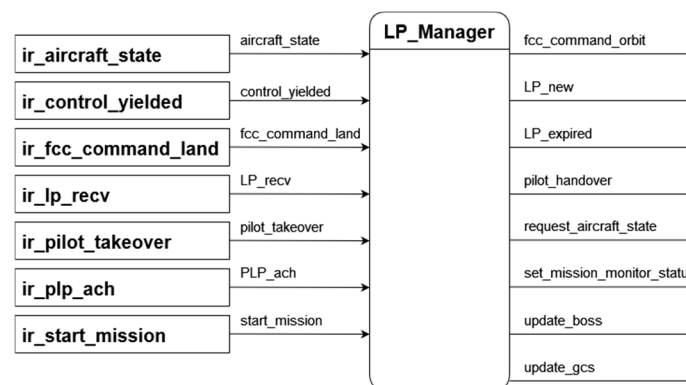


Figure 11. Example of a Test Driver for the Landing Point Manager atomic model.

The test cases for the Landing Point Manager test driver were derived from a transition tree presented in Figure 11. The transition tree was created starting at the initializing node in Figure 11 (i.e., ID). From the ID state, we defined one node for WAIT_FOR_LANDING_PHASE (WLP in Figure 11). From WAIT_FOR_LANDING_PHASE four new nodes were created (one for each transition) for REQUEST_STATE_LP (RSL in Figure 11), WAIT_FOR_LANDING_PHASE (WLP in Figure 11), PILOT_CONTROL (PC in Figure 11), and REQUEST_STATE_PLP (RSP in Figure 11). If a node represented a final

state in the atomic model or it was already in the tree, the node was marked as terminating with an “X”. This process was repeated from the newly created nodes until there were no transitions left in the atomic model.

Using the transition tree criteria, each path from the initialization node was tested to a terminating node denoted (defined by the “X” under the node). It is important to remark that to test transitions occurring from an input in a state that has a zero time advance it was required to start the model in that state to test it. For example, in Figure 12, to test the path from WLP to PC the test driver would initialize the model in the WLP state and have an event when time equals zero to transition to PC. This method only tested the defined transitions. To accomplish complete coverage of the models the sneak paths also needed to be tested. This means that each state in an atomic model needed to be tested for all the inputs that would not cause it to transition. This was done to confirm that no undefined behavior could occur.

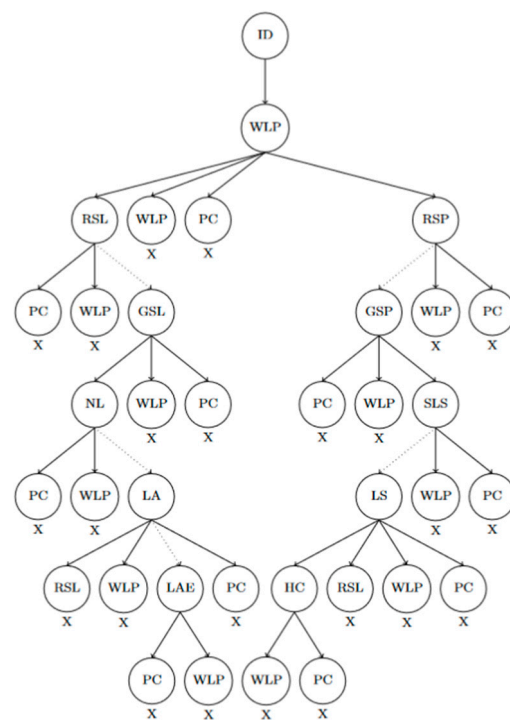


Figure 12. Landing Point Manager transition tree.

Once each test case was defined, the test driver was used to simulate the model. Table 3 shows test cases for the Landing Point Manager atomic model. The scenario shows the model receiving an input at time 00:00:02:000 on the input port start_mission, indicating that the system must become active. At 00:00:10:000, a new input was received on plp_ach port indicating that the planned landing point was achieved. At 00:00:12:000, an input on the aircraft_state port was received notifying the system of the aircraft’s current state. After the simulation was completed, log files of events and state transitions became available. A sample of the log for the Landing Point Manager test driver is shown in Table 4, it includes the time of an event, the state of each atomic model at the time, the port on which the output generated (if any), and the value of the output. In Test 0, the model started in the IDLE state. Once the model received an input on the start_mission port at 00:00:02:000 (Table 3), it transitioned to the WAIT_FOR_LANDING_PHASE state. At 00:00:10:000 (Table 3) the model received the plp_ach signal and transitions to the REQUEST_STATE_PLP state. As time advanced, it immediately changed state to GET_STATE_PLP and generated an output

in the o_request_aircraft_state output port with the value 1, the system then waited for the aircraft state. The rest of the log can be interpreted in a similar way.

Table 3. Simulation test inputs to the Landing Point Manager atomic model.

LP_Manager—Test: 0		
Time	Input Port	Value
00:00:02:000	start_mission	1
00:00:10:000	plp_ach	0 10 45 -75 100 45
00:00:12:000	aircraft_state	99 0 0 0 0 0
...

Table 4. Simulation results for the tests of the Landing Point Manager atomic model.

LP_Manager		Test: 0	
Time	State	Output Port	Value
00:00:00:000	IDLE		
00:00:02:000	WAIT_FOR_LAND- ING_PHASE		1
00:00:10:000	REQUEST_STATE_PLP		0 10 45 -75 100 45
00:00:10:100	GET_STATE_PLP	o_request_air- craft_state	1
...
Test: 1			
Time	State	Output Port	Value
00:00:00:000	START_LZE_SCAN		
00:00:00:001	PILOT_CONTROL		

7.2. Deployment on the Bell 412 Helicopter

Finally, the Supervisor underwent extensive testing on the NRC’s Digital Twin, replicating the Bell 412 autonomy system. Simulation testing was conducted as a necessary precaution since any error during autonomous flight could lead to catastrophic consequences. Digital Twin served to de-risk any undetected faults in the design or implementation of the supervisory controller by deploying it in a simulation environment that was as close to the Bell-412 as possible. This included the same network environment and hardware that the supervisory controller would be deployed onto the aircraft. The simulations conducted in the Digital Twin all used flight plans developed by NRC flight test engineers that had been used previously on test flights using the Bell-412 so that the results could be compared to real-life flight test data. A variety of scenarios were considered during this testing phase by simulating sensor failures to test a number of trajectories through the supervisory controller’s behavior. As real-world test flight plans were used during Digital Twin simulations, the results could be validated by the NRC team of system engineers, flight test engineers, and test pilots who were present on the original flight tests. By including a diverse team of domain experts in the validation of simulation test results, bias from any one individual was reduced and could be further reduced by comparing the results to the real-world results of the flight tests.

Following successful validation on the Digital Twin, the Supervisor was then subjected to real flight testing on the NRC’s Bell 412 autonomy system, under the supervision of a pilot. This autonomy system comprises several software components distributed across

different hardware and interconnected through a LAN, as illustrated in Figure 13. These components include:

1. Flight Control Computer (FCC): Responsible for inner-loop autonomy state control, the FCC sends digital commands to the helicopter's flight controls and acts like an advanced autopilot.
2. Mission Planner: Utilizing a custom NRC version of QGroundControl [29], this software configures autonomous helicopter missions, specifying takeoff locations, creating flight plans, and defining speeds and altitudes between waypoints.
3. Mission Manager: NRC-developed software that oversees the helicopter's flight path, manages mission execution (desired speed, heading, altitude, etc.).
4. Landing Zone Evaluation Module: Comprising the Peregrine LIDAR system [30], this module generates coordinates for the most suitable landing location within the designated landing area.
5. Autonomy Guidance: This pilot display software communicates the autonomy's intent and reference points to the pilot.
6. Aircraft Inertial Navigation System (INS): Located within the aircraft, this module provides information about the aircraft's state, including roll, pitch, yaw, position, speed, and more, to the other autonomy components.
7. Supervisory Controller: Serving as a high-level supervisory controller, it monitors the state of the autonomy system and its sub-systems.

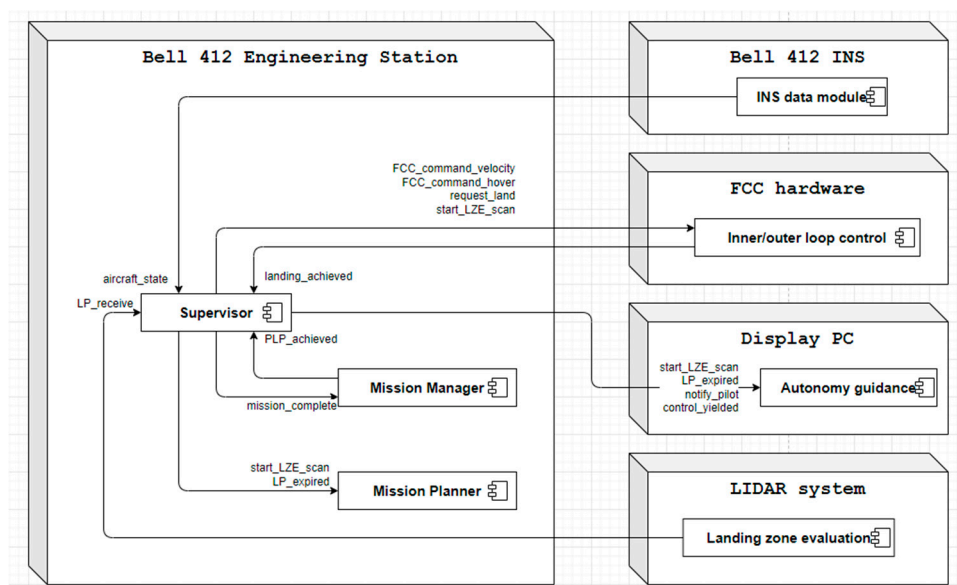


Figure 13. Hardware and software architecture in the Bell 412 autonomy system.

As depicted in Figure 13, the Supervisor, Mission Manager, and Mission Planner share the same hardware (Bell 412 Engineering Station), while the FCC, Autonomy Guidance, INS, and Landing Zone Evaluation module are housed on separate hardware units distributed across the LAN. Due to confidentiality constraints, this paper cannot disclose further details about the Bell 412 autonomy components. However, a video highlighting an autonomous flight demonstration using the Supervisor described herein can be accessed via the following link: <https://www.youtube.com/watch?v=dQqOMaNegEc> (accessed on 22 September 2025).

Notably, this configuration mirrors the setup in the Digital Twin used for testing before transitioning to actual flight. The primary distinction is that all Digital Twin components reside on the same computer and use aircraft models instead of actual aircraft.

8. Conclusions and Future Work

In this work, we presented a method to develop supervisory controllers for aviation systems. This method has been prototyped and tested on a concrete use case—development of the supervisory controller for the NRC’s Bell 412 autonomy system but could be extended to other systems in aviation. The model (and therefore, the final software) was developed using a formal method, in this case, the DEVS formalism and DEVS-Graph notation. The graphical models were then used to implement the atomic and coupled models in C++, and Cadmium was used to develop the Supervisor. The models were validated using the graphical specifications and verified using Cadmium in both simulation and real-time testing suites. Since the project life cycle followed the spiral model, the system was rapidly prototyped and tested after every update.

While the DEVS framework offers strong modularity and extensibility, enabling new behaviors and mission profiles to be added with minimal reconfiguration, one practical limitation is the potential for increased model complexity. Compared to automaton-based approaches, which often rely on fewer transitions and simpler state representations, DEVS models can require a larger number of explicitly defined states and transitions to capture the same behavior. This can lead to more detailed specifications and longer development times. However, this trade-off is justified by the long-term benefits of maintainability and scalability that DEVS provides.

When analyzing the integration of the Supervisor with the rest of the autonomy system, the issue of cause-effect problems was encountered. In the cases when transport time for messages was low, for example over a bus in a single hardware, there was little chance that messages were received out of order. In other cases, when the transport time of messages was high due to source and destination components being located on different hardware, there were times when the messages arrived in a different order to which they were sent, or the packet was lost, resulting in the autonomy system exhibiting incorrect and unexpected behavior. Proving a solution to the cause-effect problem was later addressed by the research team by developing a reliable communication protocol, called “CVLAD link, or CLINK”, and a health monitoring system, which ensured no important messages were dropped or out of order.

It is crucial to emphasize that in tackling the aforementioned challenge, there was no necessity for modifications to the supervisory controller, and our focus was only on enhancing the reliability of communication interfaces. However, due to confidentiality concerns, further details about these interfaces cannot be disclosed in this paper.

Author Contributions: J.H.—software, writing, original draft preparation; T.T.—software, writing, original draft preparation; C.R.M.—methodology, conceptualization, writing, review and editing; I.B.—methodology, conceptualization, writing, review and editing; G.W.—methodology, supervision, review and editing. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

NRC	National Research Council
DEVS	Discrete Event System Specification
CVLAD	Canadian Vertical Lift Autonomy Demonstration
ASRA	Advanced Systems Research Aircraft
FCC	Flight Control Computer
DES	Discrete-Event System
TTM	Transition Time Model
PLP	Planned Landing Point
LP	Landing Point
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
ARQ	Automatic Repeat Request
RUDP	Reliable User Datagram Protocol

References

- Colucci, F. *Supervised Autonomy, Step-by-Step*; Vertical Flight Society: Fort Worth, TX, USA, 2022.
- Zeigler, B.P.; Kim, T.G.; Praehofer, H. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*; Academic Press: New York, NY, USA, 2000.
- Belloli, L.; Vicino, D.; Ruiz-Martin, C.; Wainer, G. Building Devs Models with the Cadmium Tool. In Proceedings of the 2019 Winter Simulation Conference, National Harbor, MD, USA, 8–11 December 2019; Mustafee, N., Bae, K.H.G., Lazarova-Molnar, S., Rabe, M., Szabo, C., Haas, P., Son, Y.-J., Eds.; Institute of Electrical and Electronics Engineers, Inc.: Piscataway, NJ, USA, 2019; pp. 45–59.
- Wonham, W.M.; Cai, K.; Rudie, K. Supervisory Control of Discrete-Event Systems: A Brief History—1980-2015. *IFAC-PapersOnLine* **2017**, *50*, 1791–1797. [[CrossRef](#)]
- Ramadge, P.J.; Wonham, W.M. Supervisory Control of a Class of Discrete Event Processes. *SIAM J. Control Optim.* **1987**, *25*, 206–230. [[CrossRef](#)]
- Fokkink, W.; Goorden, M. Offline supervisory control synthesis: Taxonomy and recent developments. *Discret. Event Dyn. Syst.* **2024**, *34*, 605–657. [[CrossRef](#)]
- Bordón-Ruiz, J.; Besada-Portas, E.; López-Orozco, J.A. Cloud DEVS-Based Computation of UAVs Trajectories for Search and Rescue Missions. *J. Simul.* **2022**, *16*, 572–588. [[CrossRef](#)]
- Song, H.; Kim, T. Application of Real-Time DEVS to Analysis of Safety-Critical Embedded Control Systems: Railroad Crossing Control Example. *Simulation* **2005**, *81*, 119–136. [[CrossRef](#)]
- Hejase, M.; Oguz, A.E.; Kurt, A.; Ozguner, U.; Redmill, K. A Hierarchical Hybrid State System Based Controller Design Approach for an Autonomous UAS Mission. In Proceedings of the 16th AIAA Aviation Technology, Integration, and Operations Conference, Washington, DC, USA, 13–17 June 2016. [[CrossRef](#)]
- Delbene, A.; Baglietto, M.; Simetti, E. Visual Servoed Autonomous Landing of an UAV on a Catamaran in a Marine Environment. *Sensors* **2022**, *22*, 3544. [[CrossRef](#)] [[PubMed](#)] [[PubMed Central](#)]
- Borshchova, I. Vision-Based Automatic Landing of a Rotary UAV. Ph.D. Thesis, Faculty of Engineering and Applied Science, Memorial University of Newfoundland, St. John's, NL, Canada, 2017.
- QStateMachine Class. Qt Documentation. Available online: <https://doc.qt.io/qt-6.2/qstatemachine.html> (accessed on 13 June 2025).
- Praehofer, H.; Pree, D. Visual Modeling of DEVS-based Multiformalism Systems Based on Higraphs. In Proceedings of the 1993 Winter Simulation Conference, Los Angeles, CA, USA, 12–15 December 1993; Evans, G.W., Mollaghasemi, M., Russell, E.C., Biles, W.E., Eds.; Institute of Electrical and Electronics Engineers, Inc.: Piscataway, NJ, USA, 1993; pp. 595–603.
- Wainer, G.A. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*; CRC Press: Boca Raton, FL, USA, 2009.
- Van Tendeloo, Y.; Vangheluwe, H. An Evaluation of DEVS Simulation Tools. *Simulation* **2017**, *93*, 103–121. [[CrossRef](#)]
- Zeigler, B.P.; Chow, A.C.; Kim, D.H. Abstract Simulator for the Parallel DEVS Formalism. In Proceedings of the Fifth Annual Conference on AI, Simulation, and Planning in High Autonomy Systems, Gainesville, FL, USA, 7–9 December 1994; pp. 157–163.
- Niyonkuru, D.; Wainer, G. A DEVS Based Engine for Building Digital Quadruplets. *Simulation* **2021**, *97*, 485–506. [[CrossRef](#)] [[PubMed](#)]

18. Hu, X.; Zeigler, B. Model Continuity in the Design of Dynamic Distributed Real-Time Systems. *IEEE Trans. Syst. Man Cybern.-Part A Syst. Hum.* **2005**, *35*, 867–878. [[CrossRef](#)]
19. Moallemi, M.; Jafer, S.; Ahmed, A.S.; Wainer, G. Interfacing DEVS and visualization models for emergency management. In Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium (TMS-DEVS '11), San Diego, CA, USA, 3–7 April 2011; pp. 111–116.
20. Wainer, G.A.; Glinsky, E. Model-Based Development of Embedded Systems with RT-CD++. In Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Toronto, ON, Canada, 25–28 May 2004.
21. Bergero, F.; Kofman, E. PowerDEVS: A Tool for Hybrid System Modeling and Real-Time Simulation. *Simulation* **2011**, *87*, 113–132. [[CrossRef](#)]
22. Gholami, S.; Sarjoughian, H. Action-Level Real-Time Network-On-Chip Modeling. *Simul. Model. Pract. Theory* **2017**, *77*, 272–291. [[CrossRef](#)]
23. Hughes, B.; Cotterell, M. *Selection of an Appropriate Project Approach, in Software Project Management*; McGraw-Hill: London, UK, 1999; pp. 64–70.
24. van Lamsweerde, A. Domain Understanding and Requirement Elicitation. In *Requirements Engineering: From System Goals to UML Models to Software Specifications*; John Wiley: Chichester, UK, 2009; pp. 61–87.
25. Ammann, P.; Offutt, J. *Introduction to Software Testing*; Cambridge University Press: Cambridge, UK, 2017.
26. Horner, J.; Trautrim, T.; Ruiz-Martin, C.; Borshchova, I.; Wainer, G. Discrete-Event Supervisory Control for the Landing Phase of a Helicopter Flight, Winter Simulation Conference 2022. In Proceedings of the Winter Simulation Conference 2022, Singapore, 11–14 December 2022; pp. 441–452.
27. Tanenbaum, S.; Feamster, N.I.; Wetherall, D. *Computer Networks*; Pearson: London, UK, 2021.
28. Kohlhoff, C. Boost.Asio Documentation, Boost C++ Libraries, 8 December 2022. Available online: https://www.boost.org/doc/libs/1_81_0/doc/html/boost_asio.html (accessed on 1 June 2025).
29. DroneControl, 2019, QGroundControl. Available online: <http://qgroundcontrol.com> (accessed on 1 June 2025).
30. Aaron Aupperlee, 2017, The Pittsburgh Tribune-Review. Available online: <https://www.govtech.com/fs/airbus-partners-with-near-earth-autonomy-to-ensure-self-flying-cars-can-land-safely.html#:~:text=A%20laser%20scanner%20developed%20by%20Near%20Earth%20Autonomy,cars%2C%20and%20for%20slopes%20that%20could%20complicate%20landing> (accessed on 1 June 2025).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.