Proceedings of the 2022 Winter Simulation Conference
B. Feng, G. Pedrielli, Y. Peng, S. Shashaani, E. Song, C.G. Corlu, L.H. Lee, E.P. Chew, T. Roeder, and P. Lendermann, eds

DISCRETE- EVENT SUPERVISORY CONTROL FOR THE LANDING PHASE OF A HELICOPTER FLIGHT

James Horner
Tanner Trautrim
Cristina Ruiz Martin
Gabriel Wainer

Department of Systems and Computer Engineering
Carleton University
1125 Colonel By Drive
Ottawa, ON K1S 5B6, CANADA

Iryna Borshchova

Flight Research Lab
National Research Council Canada
1920 Research Road, Building U61
Ottawa, ON K1V 2B1, CANADA

ABSTRACT

We introduce a new method for supervisory control used in the landing phase of an autonomy system for the Bell-412 Advanced Systems Research Aircraft, which includes fly-by-wire capabilities. The complexity of the autonomy system makes it necessary to include a high-level supervisory controller that monitors mission's progress and allocates resources on board accordingly. Supervisory controllers are commonly embedded within a monolithic program and lack explicit state flows, requiring significant effort to modify and test the system's behavior. This research uses the DEVS formalism and the Cadmium simulation engine to model, implement, verify, validate, test, and deploy a state-based and event-driven supervisory controller for helicopters. We use the NRC's Bell-412 helicopter autonomy system as a case study to present the whole development cycle. The methodology is illustrated with simulated models that were tested using graphical specifications and domain experts and verified using Cadmium in both simulated and real-time testing suites.

1 INTRODUCTION

The Canadian Vertical Lift Autonomy Demonstration (CVLAD) is a project to build an autonomy system for Canada's National Research Council's (NRC) Bell-412 Advanced Systems Research Aircraft (ASRA). ASRA includes fly-by-wire capabilities, an advanced Flight Control Computer (FCC), LiDAR-based perception, a ground control station, a mission manager, a path planner, a detect-and-avoid module, and pilot displays (Colucci 2022). One of the major tests for the autonomy system will be to conduct an arctic resupply mission where the helicopter will take off with supplies, fly while avoiding any potential obstacles, then land at the destination all with minimal pilot input. The mission can be broken down into three distinct phases: takeoff, on-route, and landing.

The onboard autonomy system is complex and it includes many subsystems; therefore, we need a high-level supervisory controller to monitor the state of the components. The autonomy system components react to external stimuli, and send information to the Supervisor to keep an internal state that reflects the state of the autonomy system. This state information can be used to manage resources on the helicopter, as multiple subsystems might compete over shared resources (for example, the FCC and pilot might both try to control the aircraft when only one can do so at a time; or the path-planner and detect-and-avoid components might both try to alter a planned trajectory). The Supervisor must monitor the mission's progress and allocate resources on board accordingly.

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. Tight coupling between components of any system can lead to difficulty in performing modifications as well as extensive re-testing of the whole system when one small aspect of the program is changed. While supervisory controllers often aim to have state-based behavior, the lack of explicit states and transitions between states obscure the implemented behavior of the controller. The lack of explicit state flows impacts the modifiability of the behavior, as implicit states are hard to identify and change. 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 work aims to show an alternate approach where modifiability and transparency are built into the code using Discrete Event System Specifications (DEVS) (Zeigler et al. 2000) along with the Cadmium simulator (Belloli et al. 2019). A model of the supervisory controller was developed for the landing phase of the mission whose implementation can be deployed onto the Bell-412 as part of the autonomy system. The models and simulation are verified and validated against the CVLAD team's requirements. Once validated, the model developed becomes the actual software integrated into the helicopter. This same approach can be used to develop or redesign other components of the controller system such as adding on-route detect-and-avoid and path planning modules.

The paper is organized as follows: Section 2 presents the background on supervisory controllers, the DEVS formalism, and the Cadmium simulator. Section 3 explains the supervisor development process using DEVS. Section 4 describes the implementation of the supervisor along with the verification and validation process. Section 5 presents the conclusions and future work of this research.

2 BACKGROUND

Advancement in sensing, actuation, and control system computing technologies has increased the complexity of intelligent systems such as smartphones, autonomous vehicles, smart grids, automated buildings or automated flight systems (Chaterji et al. 2019). As the number of functionalities of these systems increases, the complexity of their software also increases. Therefore, supervisory controllers that manage and coordinate the interactions of the different components are needed.

Since 1980, the development of supervisory control of discrete-event systems (SCDES) has evolved from a centralized perspective to more structured architectures (Wonham et al. 2017). Over the last 20 years, SCDES has been established on the basis of the five fundamental concepts: feedback, stability, controllability, observability, and quantitative optimality (Athans and Falb 1966).

Although there have been many advances in the field of SCDES, control engineers lack of experience with modeling and specification frameworks and software expertise in engineering design (Wonham et al. 2017), resulting in monolithic programs with little separation of concerns and tightly coupled components. Additionally, industrial applications are case-specific. James et al. (2019) presented a formal design and implementation of a supervisory controller for a didactic manufacturing cell. They use a modular approach where they design a supervisor for each plant. Although they use automata to model the system, there is no formal specification. They rely on PLC Ladder Logic Programming to model and simulate the system and there is no explanation on how to translate this model into the actual supervisory controller. Van Beek et al. (2014) developed the Compositional Interchange Format for modeling, synthesis, simulation-based validation, verification, and visualization, real-time testing of model-based supervisory controllers. This approach could become unnecessarily complex, given that the supervisor is based on a hybrid observer.

Bhatti (2021) discussed the design and validation of a hybrid supervisory controller for a retrofitted P4 parallel Chevrolet Blazer. The controller, component models, and input/output interaction layers were developed MathWorks Simulink, which makes it challenging for the actual deployment. Borshchova (2017) developed a discrete-event supervisor that works in parallel with the inner-loop controller and is designed to assist the automatic landing of a multi-rotor unmanned aircraft on a moving target. It used discrete-event specifications and was tested in real-time to assist the pilot and crew when exceptions occur during the landing phase of the flight. The challenge with this approach is the explosion of states, which makes it difficult to implement and monitor in real life.

To fulfill these gaps and address the challenges discussed, we use the DEVS formalism as an alternate approach to building supervisors, focusing on modifiability and transparency.

2.1 The DEVS Formalism

Discrete Event System Specification (DEVS) (Zeigler et al. 2000) was chosen as the methodology to model the CVLAD Supervisory Controller because it addresses the current challenges with supervisory controllers described earlier. DEVS is a hierarchical and modular modeling formalism that separates the target system into atomic and coupled model components. Atomic and coupled models can be linked to each other's inputs and outputs allowing complex models to be built from simpler building blocks. The communication between models is performed through instantaneous occurrences where values are transmitted to and 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 allows 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. A full description of DEVS can be found in (Zeigler et al. 2000).

2.1.1 Graphical Specification of DEVS Models

A graphical specification (DEVS-graphs) can be used as described in specifying DEVS models (Praehofer and Pree 1993). 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 the formalisms, mathematical notation, and programming skills. Additionally, it assists the modeler with visualizing the system. 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 1), 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.

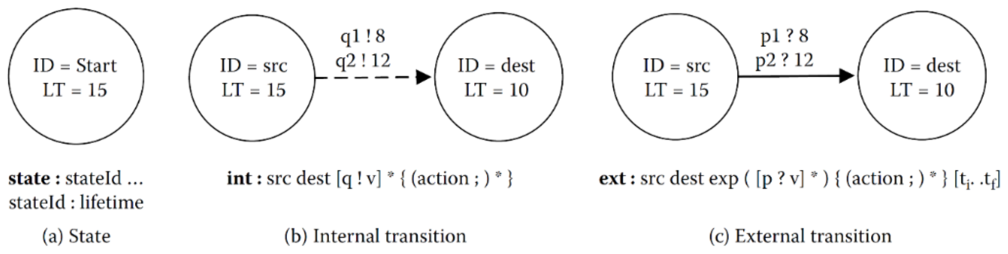

Figure 1: DEVS atomic models using DEVS-Graphs (Wainer 2009).

As shown in Figure 1, 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 time advance value (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 2), 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.

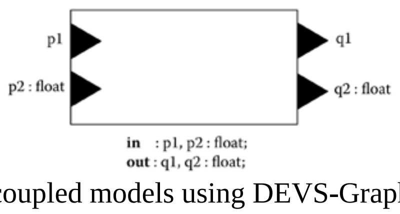

Figure 2: DEVS coupled models using DEVS-Graphs (Wainer 2009).

2.1.2 Cadmium and Real-Time Cadmium

Although there are many DEVS simulators (Tendeloo and Vangheluwe 2016), 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. The implementation is in C++, and it can use external libraries execute on a hardware target platform. Cadmium implements the abstract hierarchical simulation algorithm (Zeigler et al. 1994) for DEVS models.

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 the classes for all `objects` are defined within the Cadmium simulator.

RT-Cadmium (Niyonkuru and Wainer 2021) 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 Modelling subsystem is connected to a RunTime and Messaging subsystems. The Main Runtime Subystem 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 spawns 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 Hu and Zeigler (2005), as well as Song and Kim (2005), who defined RT models using the DEVS framework. Moallemi et al. (2011) showed how to reuse models developed in different simulation engines by interfacing E-CD++ (Wainer and Glinsky 2004) and PowerDEVS (Bergero and Kofman 2011). PowerDEVS provides a method to model hybrid systems and execute RT models. Action-Level Real-Time DEVS (Gholami and Sarjoughian 2017) 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 of an operating system or other middleware, making modular classes simple to be reused.

3 SUPERVISOR DEVS MODEL DEVELOPMENT

As discussed earlier, we developed a supervisor for the landing phase of missions. Before a mission starts, the mission planner designates the general 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 perception system will identify Landing Points (LPs) within the PLP. LPs are regions large enough for the helicopter to land in and are clear of obstacles. Multiple LPs might be found by the perception system, so for how long LPs are sought after and which LPs are accepted will be the responsibility of the Supervisor. On approach to the PLP, the autonomy system will start to receive inputs from the perception system regarding safe places to land. The perception system uses LiDAR to identify safe LPs where there is enough clearance for the helicopter to safely reach the ground. It is the job of the Supervisor to receive LPs from the perception system alongside other inputs from the FCC, mission manager, pilot, and aircraft, to determine whether the FCC should be ordered to land the helicopter or to hover then hand control over to the pilot, if no suitable landing point is found.

The behavior of the supervisor in the landing phase was separated from the rest of the system to take advantage of the hierarchical reinnature of DEVS: the overall behavior, as explained above, was partitioned into atomic models, then those atomic models were assembled to form the Supervisor. The following sections describe the models for the CVLAD Supervisory Controller and its sub-models.

3.1 Supervisor Coupled Model

The purpose of the Supervisor coupled model (Figure 3) is to manage the landing phase of the mission by receiving signals from external systems (such as the FCC, ground station, mission manager, path planner, and pilot display), processing those signals to advance the state of the system and notifying those systems back with the actions to be taken.

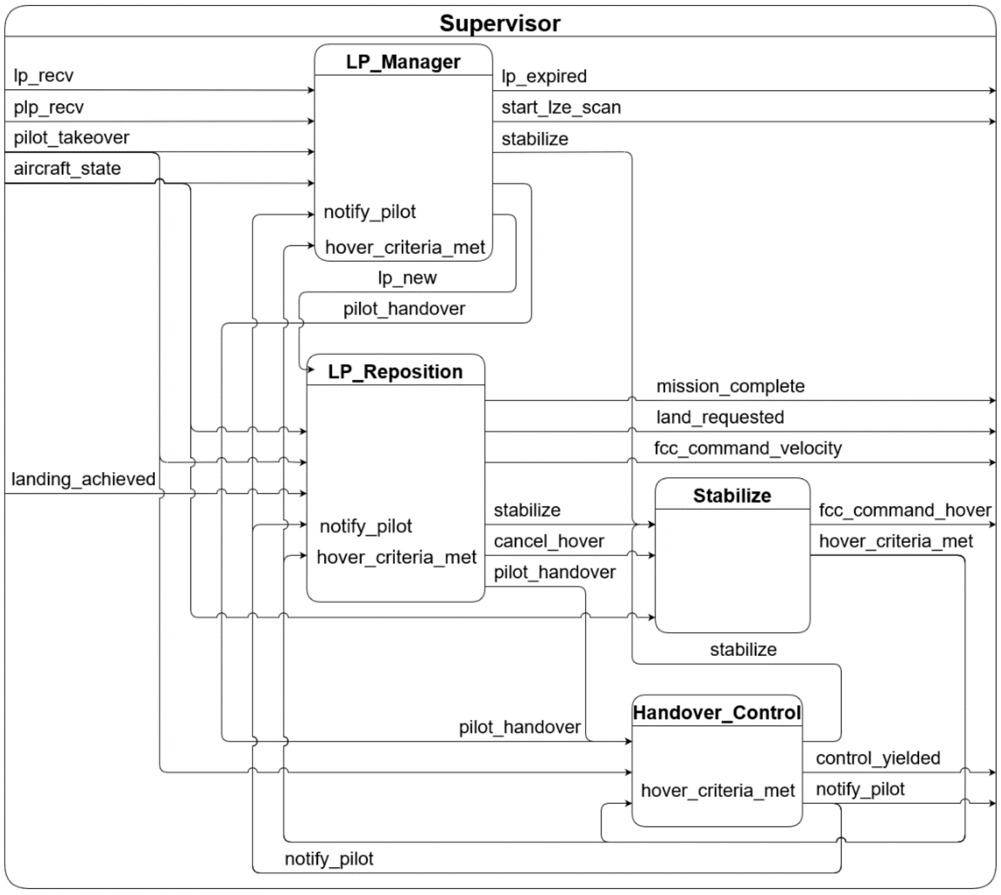

Figure 3: Supervisor coupled model.

To fully encapsulate all the behavior of the CVLAD Supervisory Controller, the Supervisor coupled model was decomposed into 4 sub-components - LP Manager, LP Reposition, Stabilize, and Handover Control. Each subcomponent encapsulates a specific behavior of the system.

The LP Manager model facilitates the receipt, validation, and acceptance of landing points as well as the scanning behavior when the planned landing point has been reached.

The LP Reposition model coordinates the final stage of the landing phase, from receipt of a valid landing point to declaring the mission complete after landing.

The Stabilize model defines the procedure of transitioning the helicopter to a given hover criteria (i.e., a stable position).

The Handover Control model is used to transfer control of the aircraft away from the autonomy system to the evaluation pilot.

The Supervisor coupled model (Figure 3) uses 5 input ports and 8 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	Current position, heading, and velocity of the helicopter
landing_achieved	Acknowledgment: the helicopter has landed
lp_recv	Landing points
pilot_takeover	Acknowledgment: the pilot has forcibly taken control of the helicopter
plp_ach	Acknowledgment: the helicopter has achieved the planned landing point

Table 2: Outputs from the Supervisor coupled model.

Output port	Description of the outputs sent
control_yielded	Notification: the Supervisor has relinquished control of the helicopter
fcc_command_hover	Command to hover at a given location
fcc_command_velocity	Command to change the velocity of the helicopter
land_requested	Request to start landing the helicopter at a given location
lp_expired	The LP accept timer has expired, so no more LPs will be accepted
mission_complete	Declaration that the mission is complete after landing
notify_pilot	Request to ask the pilot to take control of the helicopter
start_lze_scan	Command to start the scan of the landing zone

In the following section, we discuss the definition of the Handover Control atomic model. All the atomic models in the system are defined in a similar way. The Landing Point Reposition coupled model is defined in a similar way to the Supervisor coupled model. All models and the implementation presented in Section 4 can be found in our GitHub repository (Horner and Trautrim 2022).

3.2 Handover Control Atomic Model

The Handover Control atomic model in Figure 4 hands over control of the aircraft to the pilot.

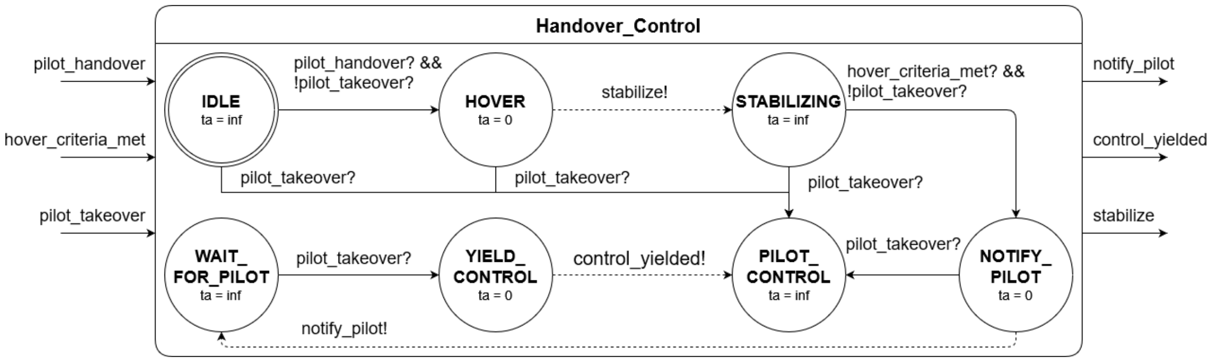

Figure 4: Handover Control atomic model.

The atomic model uses seven states to represent this activity: (1) IDLE, (2) HOVER, (3) STABILIZING, (4) WAIT_FOR_PILOT, (5) YIELD_CONTROL, (6) PILOT_CONTROL, (7) NOTIFY_PILOT. The model is initialized in the IDLE state. It remains IDLE state until an input is received. The inputs will arrive at the model when the aircraft reaches an impermissible state such as when an LP cannot be found. If the model receives an input from the *pilot_handover* input port, it transitions into the HOVER state and it immediately transitions to the STABILIZING state after sending a hover request through the corresponding output port (i.e., *stabilize*). The model stays in the STABILIZING state until it receives an input through *hover_criteria_met* (which means that the aircraft is now hovering). Then, the model transitions to the NOTIFY_PILOT state, then immediately transition to the WAIT_FOR_PILOT state after sending a *notify_pilot* output (which notifies the pilot that their intervention is required). Once the model receives the *pilot_takeover* signal (i.e., the pilot notifies the system that they have taken control of the aircraft) the model transitions to the YIELD_CONTROL state. The model transitions to the PILOT_CONTROL state after sending a *control_yielded* output (which signals to the aircraft that is has relinquished control). The pilot can manually take control of the aircraft using a switch in the cockpit. Once activated, the model will receive a *pilot_takeover* input and transition from any state (except YIELD_CONTROL and WAIT_FOR_PILOT) to the PILOT_CONTROL state. This disengages the model leaving full control of the aircraft to the pilot.

4 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 perform verification and validation again. Each C++ model implementation was simulated using a test driver, then simulation results were inspected for verification and validation, as is shown in Sections 4.3.1 and 4.3.2.

4.1 Handover Control Atomic Model

The Handover Control atomic model was implemented by translating the specification in DEVS Graphs into a C++ class that could be used by the Cadmium simulation library as shown in Figure 5.

```
template<typename TIME> class Hdov_Ctl {
    public:
    DEFINE_ENUM_WITH_STRING_CONVERSIONS(States:   (IDLE),(HOVER),(STABILIZING),(NOTIFY_PILOT),
                                (WAIT_FOR_PILOT),(YIELD_CONTROL),(PILOT_CONTROL) );

    using input_ports = tuple<typename Hdov_Ctl_defs::i_hover_criteria_met,
          typename Hdov_Ctl_defs::i_pilot_handover, typename Hdov_Ctl_defs::i_pilot_takeover>;
    using output_ports = tuple<typename Hdov_Ctl_defs::o_notify_pilot,
          typename Hdov_Ctl_defs::o_control_yielded, typename Hdov_Ctl_defs::o_stabilize>;

    struct state_type { States current_state; }; state_type state;

    Hdov_Ctl() { state.current_state = States::IDLE; }

    void internal_transition() {
        switch(state.current_state) {
            case States::HOVER:
                state.current_state = States::STABILIZING; break;
        …  }
    }
    void external_transition(TIME e, typename make_message_bags<input_ports>::type mbs) {
        switch (state.current_state) {
            case States::HOVER:
                received_pilot_takeover = get_messages<i_pilot_takeover>(mbs).size()>=1;
                if (received_pilot_takeover) {
                        state.current_state = States::PILOT_CONTROL; }
            break;
            … }
    }
    typename make_message_bags<output_ports>::type output() const {
        switch(state.current_state) {
            case States::NOTIFY_PILOT:
                bag_port_out.push_back(true);
                get_messages<typename Hdov_Ctl_defs::o_notify_pilot>(bags) = bag_port_out;
                break;
        …  }
    }
    TIME time_advance() const {
        switch(state.current_state)
           case States::IDLE: return numeric_limits<TIME>::infinity(); break;
            ...}
    }
};
```

Figure 5: Code snippet of the implementation of the Handover Control atomic model.

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, as seen in Figure 4. The external transition method checks the inputs of the model and then deciding the transition based on the input received and the current state (also defined in Figure 4). 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.

4.2 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 as shown in Figure 6.

```
class Supervisor {
    public:
    shared_ptr<model> lp_manager = translate::make_dynamic_atomic_model< … >("lp_manager", … );
    shared_ptr<model> stabilize = translate::make_dynamic_atomic_model< … >("stabilize");
    shared_ptr<model> Hdov_Ctl  = translate::make_dynamic_atomic_model< … >("Hdov_Ctl");

    shared_ptr<coupled<TIME>> lp_reposition = make_shared<coupled<TIME>>("lp_reposition", … );

    Ports iports = {
        typeid(Supervisor_defs::i_landing_achieved),typeid(Supervisor_defs::i_aircraft_state),
        …  };

    Ports oports = {
        typeid(Supervisor_defs::o_LP_expired), typeid(Supervisor_defs::o_start_LZE_scan),…
    };

    Models submodels = { lp_manager, stabilize, Hdov_Ctl, lp_reposition };
    EICs eics = { ... };
    EOCs eocs = { ... };
    ICs ics = { ... };
};
```

Figure 6: Code snippet of the implementation of the Supervisor coupled model.

The sub-models of the Supervisor are built using the `make_dynamic_atomic_model` and `make_shared` functions provided by Cadmium; then, references to each model are given in the `submodels` structure. Once each sub-model is initialized, the couplings are 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 `eocs` 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 are defined in a similar manner.

4.3 Model Testing

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 simulation and interactive real-time test drivers.

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 are 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 (Section 5.2) allowed researchers at the FRL to test if all required behavior was present as well as validate that the outputs were generated at the right time.

4.4 Simulation Testing

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 7 shows an example test driver (for the Handover Control atomic model shown in Section 3.2 and implemented in Section 4.1). The model consists of three input readers, named as the port the reader is coupled to.

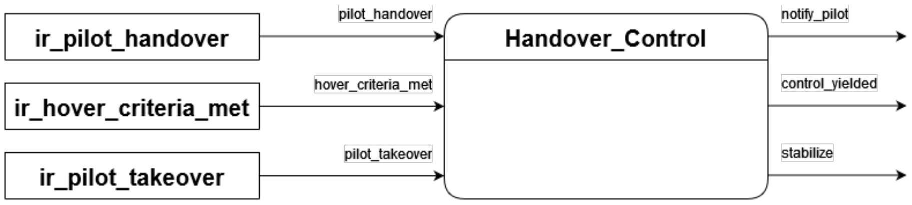

Figure 7: Example of a Test Driver for a DEVS model.

The test cases for Figure 7 are derived from a transition tree presented in Figure 8. The transition tree is created starting at the initializing node in Figure 4 (i.e., IDLE). From the IDLE state, we define two leaves (one for each transition) for PILOT_CONTROL (PC in the figure) and HOVER (HOV in the figure). If a node is a final state or it is already in the tree mark, the leave is marked as terminating with an "X". This is repeated from the newly created leaves until there are no transitions left in the atomic model.

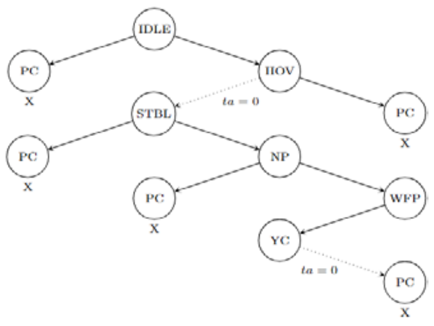

Figure 8: Handover Control transition tree.

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 6, to test the path from HOV to PC the test driver would initialize model in the HOV state and have an event at time equals zero to transition to PC. This method of testing 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.

Once each test case was defined, the test driver is used to simulate the model. Table 3 shows test cases for the Handover Control atomic model. The first scenario shows the model receiving an input at a time 00:00:05:000 in the input port *pilot_handover*, indicating that the system must get ready to release control to the pilot. At time 00:00:10:000, a new input is received in *hover_criteria_met* indicating that the hover criterion has been met. At time 00:00:15:000, an input in *pilot_takeover* is received indicating that the pilot took control. After the simulation run, log files of events and state transitions are available. A snippet of the log tables for test cases 0 and 1 of the Handover Control are shown in Table 4, which 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 starts in the IDLE state. Once the model receives an input in *pilot_handover* at time 00:00:05:000 (Table 3), it transitions to the HOVER state. As the time advance of the state is zero, it immediately changes state to STABILIZING and generates an output in the *o_stabilize* with the value 0 0 0 15 16.4 5 3 15 3 0 0 0, stating a hover criterion for the aircraft. The rest of the log is interpreted in a similar way.

Table 3: Simulation test inputs to the Handover Control atomic model.

Hdov_Ctl - Test: 0		
Time	**Input Port**	**Value**
00:00:05:000	pilot_handover	0 0 0 0
00:00:10:000	hover_criteria_met	1
00:00:15:000	pilot_takeover	1
…		

Table 4: Simulation results for the tests of the Handover Control atomic model.

Hdov_Ctl	**Test: 0**		
Time	**State**	**Output Port**	**Value**
00:00:00:000	IDLE		
00:00:05:000	HOVER		
00:00:05:000	STABILIZING	o_stabilize	0 0 0 15 16.4 5 3 15 3 0 0 0
…	…	…	…
Test: 1			
Time	**State**	**Output Port**	**Value**
00:00:00:000	NOTIFY_PILOT		
00:00:00:001	PILOT_CONTROL		

4.5 Real-Time Testing

A real-time test driver was developed to allow different trajectories through the Supervisor to be explored in an interactive manner by researchers at the FRL and by other stakeholders. A command line Supervisor test driver (Figure 9) was built as a Supervisor Command Line Input atomic model coupled to each of the Supervisor input ports. The simulation would run until all simulated DEVS models were passivated, then the log files would be parsed into more presentable tables for review by the user. In this way, the Supervisor could be tested in real-time and dynamically, giving free rein to the tester in exploring all aspects of the Supervisor's behavior. The Supervisor Command Line Input atomic model was designed and implemented to query the user for input and direct the entered message to the appropriate port of the Supervisor. The user input is parsed to determine the destination port and to populate a message structure to send.

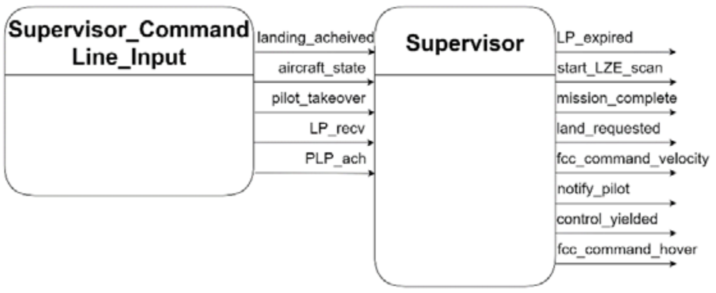

Figure 9: Command Line Supervisor Test Driver DEVS model.

Before the Supervisor could be deployed onto the NRC's Bell-412 ASRA, thorough real-time integration testing must be conducted in a controlled environment. To prepare for deployment on the helicopter, the Supervisor and integration models would need to be tested on the NRC's simulator: a version of the autonomy system leveraging high fidelity physics simulations to supply Bell-412 helicopter dynamics including hardware-in-the-loop (HIL).

Incremental testing is vital to ensure the safety on the aircraft as well as save valuable flight-testing hours. By testing in a simulated environment, it can be ensured that all components can communicate effectively, and no obvious faults exist in the production system. Testing with HIL allows for rapid iteration if faults are uncovered, so turn-around-time for bug fixes are smaller and the system can be tested again sooner. To integrate the Supervisor with the rest of the autonomy system, the simulated version, and then on board the aircraft, interfaces needed to be created to connect the Supervisor to the Local Area Network (LAN) of the aircraft. The aircraft's autonomy system sends messages over the LAN using the User Datagram Protocol (UDP). To facilitate the communication between events generated by the Supervisor and the autonomy system components over the network, two DEVS atomic models were created: the UDP Input and UDP Output models (Figure 10).

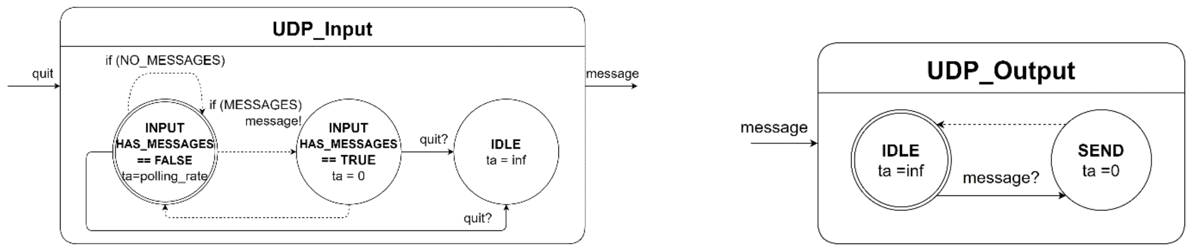

Figure 10: Input and output atomic models. (a) UDP Input (b) UDP Output.

The UDP Input model (Figure 10a) is used to receive UDP packets of a generic type and convert the packet into an event that can be understood by the Supervisor. The UDP output atomic model (Figure 10b) is used to translate an event generated by the Supervisor and send it as a UDP packet to a specified address and port. The content is sent to a predetermined network address and port instead of an external output port. The UDP Input model was designed to connect to each input of the real-time Supervisor model so the Supervisor could receive UDP packets from any other autonomy system components the network, for example, the perception system or the FCC. 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 (Note: the architecture of the aircraft and its simulator cannot be detailed due to a non-disclosure agreement).

5 CONCLUSIONS AND FUTURE WORK

In this work, the behavior of the supervisory controller in the landing phase of a mission for the NRC's Bell-412 ASRA was modeled using DEVS formalism and DEVS-Graph notation. The graphical models were then used to implement the atomic and coupled models in C++ so Cadmium could be used to simulate the Supervisor. The simulated 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.

The use of DEVS as a method to develop and design supervisory controllers is novel within the aerospace field. Supervisory controllers developed using common industry techniques often suffer from several issues leading to difficulty in maintenance, future development, and testing. 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 unit tested separately before being integration tested.

When analyzing the integration of the Supervisor with the rest of the autonomy system, the issue of cause-effect problems was encountered. On one hand, when transport time for messages is low, for example over a bus in a single computer, there is very little chance that messages might be received out of order. Also, if a system is purely reactionary, i.e., stateless and have the same behavior when supplied with an input, the order in which inputs are received does not matter, as the system cannot enter an incorrect state. On another hand, when one piece of software can change the state of another by sending messages, there is a risk that if the messages being passed between the processes arrive in a different order to which they were sent, the system will exhibit incorrect and unexpected behavior. Proving a solution to the cause-effect problem on the actual helicopter will be the future steps of this research.

REFERENCES

Athans, M and P. Falb. 1966. *Optimal Control: An Introduction to the Theory and its Applications*. New York: McGraw-Hill.

Belloli, L., D. Vicino, C. Ruiz-Martin, and G. Wainer. 2019. "Building Devs Models with the Cadmium Tool". In *Proceedings of the 2019 Winter Simulation Conference*, edited by N. Mustafee, K.H.G. Bae, S. Lazarova-Molnar, M. Rabe, C. Szabo, P. Haas, and Y.-J. Son, 45-59, Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Bergero, F., and E. Kofman. 2011. "PowerDEVS: a Tool for Hybrid System Modeling and Real-Time Simulation". *Simulation* 87(1–2): 113–132.

Bhatti, A., and R. Fraser. 2021. *Model-Based Design Framework Development of a Hybrid Supervisory Controller for a P4 Parallel Hybrid Vehicle.* Master Thesis. Waterloo: University of Waterloo.

Borshchova, I. 2017. *Vision-based Automatic Landing of a Rotary UAV.* PhD Thesis. St. John's: Faculty of Engineering and Applied Science. Memorial University of Newfoundland.

Chaterji S., P. Naghizadeh, M.A. Alam, S. Bagchi, M. Chiang, D. Corman, B. Henz, S. Jana, N. Li, S. Mou, M. Oishi, C. Peng, T. Rompf, A. Sabharwal, S. Sundaram, J. Weimer, and J. Weller. 2019. "Resilient Cyberphysical Systems and their Application Drivers: A Technology Roadmap," *arXiv*, https://arxiv.org/pdf/2001.00090.pdf.

Colucci, F. 2022. "Supervised Autonomy, Step-by-Step". *Vertiflite,* Fort Worth, USA: Vertical Flight Society.

Gholami S, and H. Sarjoughian. 2017. "Action-Level Real-Time Network-On-Chip Modeling". *Simulation Modelling Practice and Theory* 77: 272–291

Horner, J., and T. Trautrim. 2022. *Flight Supervisor*. https://github.com/SimulationEverywhere-Models/FlightSupervisor, accessed 8th August 2022.

Hu, X., and B. Zeigler. 2005. "Model Continuity in the Design of Dynamic Distributed Real-Time Systems". *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 35(6): 867–878

Niyonkuru, D., and G. Wainer. 2021. "A DEVS Based Engine for Building Digital Quadruplets". *Simulation* 97:7 485-506.

Praehofer, H., and D. Pree. 1993. "Visual Modeling of DEVS-based Multiformalism Systems Based on Higraphs". In *Proceedings of the 1993 Winter Simulation Conference*, edited by G.W. Evans, M. Mollaghasemi, E.C. Russell, and W.E. Biles, 595–603, Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Song H., and T. Kim. 2005. "Application of Real-Time DEVS to Analysis of Safety-Critical Embedded Control Systems: Railroad Crossing Control Example". *Simulation* 81(2): 119–136.

Van Beek, D.A., W.J. Fokkink, D. Hendriks, A. Hofkamp, J. Markovski, J.M. van de Mortel-Fronczak, and M.A. Reniers. 2014. "CIF 3: Model-Based Engineering of Supervisory Controllers". *International Conference on Tools and Algorithms for the Construction and Analysis of Systems,* edited by E. Ábrahám, and K. Havelund, 575-580. Berlin, Heidelberg: Springer.

Van Tendeloo Y., and H. Vangheluwe. 2017. "An Evaluation of DEVS Simulation Tools". *SIMULATION* 93(2):103-121.

Wainer, G. A. 2009. *Discrete-Event Modeling and Simulation: A Practitioner's Approach*. Boca Raton: CRC Press.

Wainer, G.A., and E. Glinsky. 2004. "Model-Based Development of Embedded Systems with RT-CD++." 10th *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 25th-28th May 2004, Toronto, ON, Canada.

Wonham, W.M., K. Cai, and K. Rudie. 2017. "Supervisory Control of Discrete-Event Systems: A Brief History–1980-2015". *IFAC-PapersOnLine* 50(1): 1791-1797.

Zeigler, B.P., A.C. Chow, and D.H. Kim. 1994. "Abstract Simulator for the Parallel DEVS Formalism". *Proceedings of the Fifth Annual Conference on AI, Simulation, and Planning in High Autonomy Systems.* 7th – 9th December 1994, Gainesville, Florida, USA, 157-163.

Zeigler, B. P., T. G. Kim, and H. Praehofer. 2000. *Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems.* New York: Academic Press.

AUTHOR BIOGRAPHIES

JAMES HORNER is a Software Engineering student at the Department of Systems and Computer Engineering at Carleton University. His email address is jameshorner@cmail.carleton.ca

TANNER TRAUTRIM is a Software Engineering student at the Department of Systems and Computer Engineering at Carleton University. His email address is tannertrautrim@cmail.carleton.ca

CRISTINA RUIZ MARTIN PhD, is an Instructor at the Department of Systems and Computer Engineering at Carleton University. Her email address is cristinaruizmartin@sce.carleton.ca

IRYNA BORSHCHOVA PhD, is a Research Officer at the National Research Council of Canada, Flight Research Laboratory. For over a decade she has been involved in research and development of detect-and-avoid systems. She is a member of RTCA, ASTM and EUROCAE. Her email address is iryna.borshchova@nrc-cnrc.gc.ca

GABRIEL WAINER PhD, is a Full Professor at the Department of Systems and Computer Engineering at Carleton University. He is a Fellow of SCS. His email address is gwainer@sce.carleton.ca