

Model Reuse for the Training of Fault Scenarios in Aerospace

Pieter J. Mosterman* and Jason Ghidella†

The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA, 01760, USA

The use of models has become common in aircraft development. Many different models of aircraft subsystems are designed to analyze, optimize, and synthesize the subsystem behavior. An important behavior in training simulators is the aircraft behavior when failures occur and it would be beneficial to re-use the subsystem models that were designed in the development process. One of these models captures the behavior of the redundancy management system that operates on aircraft elevators. This paper shows how this model can be connected to another model that captures the dynamics of the actuator hydraulics and elevator mechanics to study transient effects of loss of control. The same model of the redundancy management system is then included by reference in an aircraft model for training purposes to investigate transient effects of fault scenarios. Finally, the paper discusses how the real-time code that is used in training simulators can be automatically generated from the same model, using customization features to ensure that the generated code is compatible with the software with which it needs to be integrated.

I. Introduction

THE use of Model-Based Design¹ has become prevalent in aircraft development. Models are now being used to design, analyze, optimize, and synthesize hardware and software systems.² For example, in terms of hardware, a model of a hydraulic actuator may be used to verify that a particular design, its dimensions, connections, and configuration, satisfies the requirements such as delivered power. In terms of software, a model of a feedback control law may be used to verify that a particular choice of sample rate and fixed-point resolution satisfies the requirements, such as stability and robustness.

Though widespread, the use of models is still very fragmented. Different teams of engineers are involved in the different stages of aircraft design, and each of these teams relies on models they designed themselves. In many cases, this is because no tools are available yet that allow transformation, or even correlation, of models that capture different system perspectives. For example, at present, there is modest support to relate a SolidWorks^{®3} Computer Aided Design (CAD) model to a Simulink^{®4} model. Though a SolidWorks model can be imported in SimMechanics,⁵ it cannot be exported back for further editing.

On other occasions, however, the same formalism or even tool may be used by different design teams, with great potential benefit for model reuse. Partially, this may require enculturation, as it is tempting to design a model for one's own purposes without trying to produce a more general representation of the system it aims to capture. It is not uncommon to design a model of a physical system based on a particular scenario in which it is used. For example, a number of fixed sequences of valve openings and closings may be built into the model of a hydraulic actuator, and so an extraordinary situation that was not pre-conceived, and that does not occur during nominal operation, is not included. As such, the model can be used in the design of, say, the reactive supervisory control, but it would be of no use for reconfiguration control in the face of failures.

In addition to removing as many built-in assumptions as is economical, the tool used to design the models and the formalism used for its representation determine the re-usability of the model. Though many industrial design efforts have standardized on MATLAB[®] and Simulink as their modeling and simulation environment, efforts are under way to allow model exchange between tools⁶ and the transformation of models

*Senior Research Scientist.

†Technical Marketing Manager.

into different formalisms.⁷ A more straightforward yet versatile approach to combine and integrate models is by exploiting the code generation facilities that are available in many modeling tools.⁸ By transforming the different models to be combined into C/C++ code, they can be compiled into one executable, which requires nothing more than some ‘glue code’. This approach is especially viable in cases where real-time execution is a critical requirement.

This paper discusses the reuse of a model created for the design of a reactive controller to manage the switching of redundant actuators. In particular, the actuators are used to position the control surfaces of the HL-20 crew rescue vehicle shown in Fig. 1, which was developed at NASA Langley.⁹

Once a model of the reactive control has been designed, a failure mode and effect analysis (FMEA) is performed. This analysis verifies that in each mode an operational actuator with required performance characteristics is available. Once completed, the switching *between* modes is analyzed, studying the transient effects. This analysis requires detailed continuous models of the actuator hydraulics.

The same controller model is then directly integrated into a model of the HL-20. This model connects to a FlightGear¹⁰ simulation and provides a visualization of flight behavior. Similarly, real-time code can be generated from the controller model for different targets and integrated with real-time flight simulators.

In Section II, the actuator redundancy control as designed in Stateflow^{®11} is presented. Section III then shows how Simulink can be used to connect the discrete event control to a continuous-time model of the actuator hydraulics, allowing the study of switching transients. In Section IV, it is then shown how the controller model can be directly included into a model of the HL-20. Section V discusses the need to integrate generated and existing code. Section VI presents conclusions.



Figure 1. The HL-20 crew rescue vehicle.

II. Redundancy Management

FAIL-CRITICAL systems rely on redundancy to achieve a desired level of safety. Such systems include the attitude control systems of aircraft, the ‘by-wire’ systems of automobiles, and cooling systems in nuclear plants.

A. Redundancy

Redundancy allows functionality to be assumed by components that were not designed to be the ones primarily responsible. For example, in aircraft, the primary attitude control relies on elevators to achieve a desired aircraft pitch. Because of the importance of this control functionality, the elevator control systems consists of several forms of redundancy.¹²

Figure 2 shows a simplified configuration of the redundancy in the aircraft elevator system. It consists of two elevators, one on the left and one on the right. Each elevator can be positioned by two actuators, only one of which should be active at any given time. Four actuators are connected to three separate hydraulic circuits, as shown in Fig. 2. The two primary flight control units (PFCU) are used to control either the inner or outer actuators. To this end, there are two control laws available. In nominal operation, a sophisticated *input-output* (IO) control law is applied to the left (LIO) and the right (RIO) outer actuators. In case of a failure, a *direct link* (DL) control law with reduced functionality is available for the left (LDL) and right (RDL) inner actuators. A more realistic and extensive version of this system has been presented in other work.^{12,13}

The example in Fig. 2 relies on *physical redundancy*: When one component fails, another can be activated. This is a potentially expensive proposition; it carries significant cost in terms of duplication, and adds weight, and, therefore, operational expense, to the system.

The alternative is *functional redundancy*, which uses models to derive additional information about the system. For example, an actuator model might prescribe that the positioning-rate cannot exceed a given value. In case two consecutive measurements are obtained that would require a faster rate, the measurement can be deemed erroneous.

Though functional redundancy is less expensive to design and operate, hardware redundancy is still

indispensable in aircraft systems.¹⁴

B. Managing Redundancy

Physical redundancy requires supervisory logic to determine which of the redundant components should be active, and in what mode the remaining components should operate. In general, this logic contains many interdependencies, since it is imperative that always one and only one component performs a given task. If more than one were active, slight disturbances in their control would lead to conflicting forces, which could damage the system.

For example, in the elevator control system in Fig. 2, if both the left inner and left outer actuator are active at a given time, they may aim to drive the left elevator to different positions. Even if there is only a slight difference between the two, this will cause stresses in the material that may result in damaged hardware.

In general, there is a variety of requirements for the redundancy management.¹⁵ They typically are formulated in natural language and may, indeed, contain inconsistencies. These high-level requirements need first to be translated into formal specifications. A controller model must then be designed that embodies the desired behavior. This tends to be a complex process. In this paper, a small set of requirements is used to guide the design of the redundancy management. These requirements are listed in Table 1.

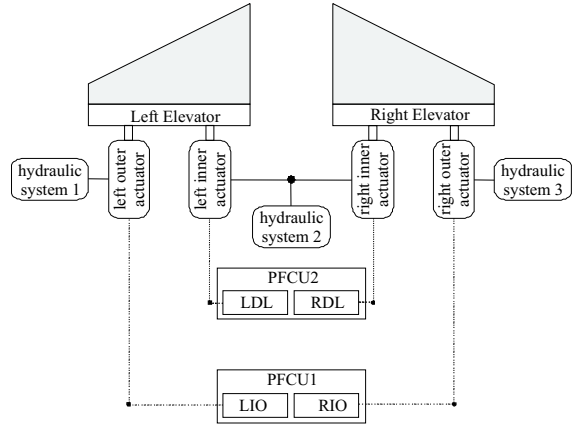


Figure 2. The elevator redundancy configuration.

Table 1. Elevator system requirements.

ID	Description
1	If possible the same control law should be <i>active</i> for both the left and right elevators.
2	If available, the IO control law should be <i>active</i> instead of the DL control law.
3	The actuator that is not active should be in <i>standby</i> .
4	If the pressure of the hydraulic circuit is low and the position measurement fails, the corresponding actuator should be switched to <i>off</i> .
5	If the pressure of the hydraulic circuit is nominal and the position measurement fails, the corresponding actuator should be switched to <i>isolated</i> .
6	Controller state changes should be made only in response to failure events.

Natural language requirements such as those in Table 1 can be incomplete, inconsistent, and difficult to interpret. The combination of Requirement 1 and Requirement 2 is an example of the possible ambiguity: If one actuator can be operated only in DL, should the other still be operated in IO? To formalize the desired behavior, a Stateflow model can be designed. This consists of a number of concurrent state transition diagrams that contain a hierarchical state transition structure. One such state transition diagram is used per actuator control module.

Figure 3 shows the switching logic for the left actuator control module that contains the IO control law. The structure of the other three state transition diagrams is identical; however, the transition conditions and the variables they set are different. The execution order of the four state transition diagrams have been deterministically set so that at each evaluation of the statechart, the LIO module is executed first, then the RIO module, then the LDL module, and finally the RDL module. This order of execution is critical in determining the logic for the recovery aspects of the system.

When the PFCUs are powered up, the state transition diagrams take the entry transitions (commonly referred to as the default transition, and represented in the diagram as arrows with a solid circle on one end) that move them into *passive*. This can be seen in Fig. 3, which also shows that the LIO_mode variable is set to 2. Analogously, the three other statecharts set the variables RIO_mode, LDL_mode, and RDL_mode to 2 as well. From this passive state, the four modules infer their active states. The LIO module in Fig. 3 moves to *active*

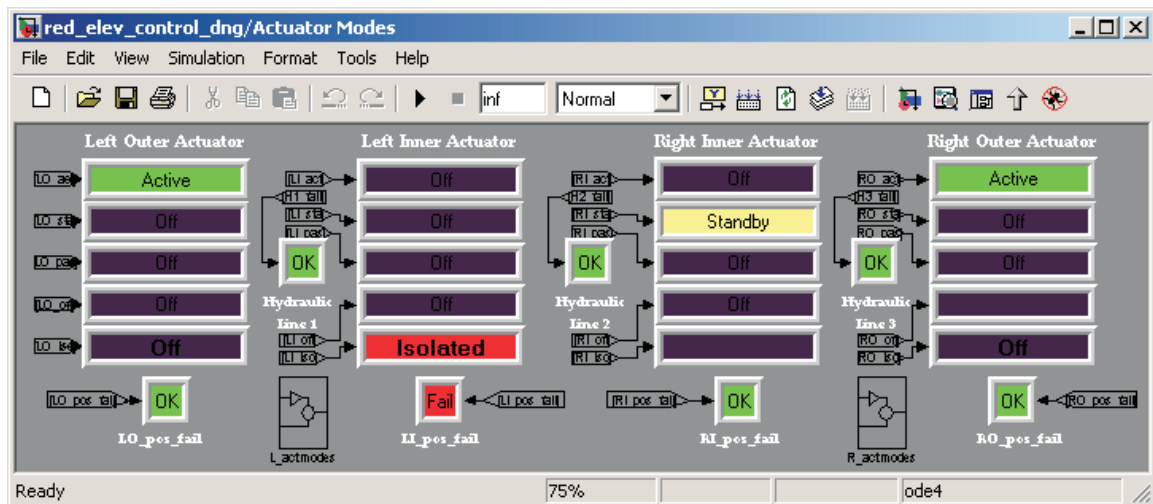


Figure 4. The FMEA graphical user interface.

outer actuator and the right outer actuator are *active* and control the left and right elevators, respectively. The left inner actuator is *isolated* and cannot be switched to *active* because of a faulty position measurement that mandates ground servicing. The right inner actuator is *standby*, and, therefore, shadows the control commands. Switching transients are thus minimized when the actuator is switched to *active*.

The failures that can be independently injected are listed in Table 2. For each of these individual failures, it is ensured that there is always one actuator active for each side. Furthermore, it is ensured that combinations of failures, either concurrently or sequentially, that have a certain probability leave the system with one active actuator for each side (as is the case in Fig. 4).

Table 2. Possible elevator system failures.

ID	Description
1	Loss of pressure in hydraulic circuit 1.
2	Loss of pressure in hydraulic circuit 2.
3	Loss of pressure in hydraulic circuit 3.
4	Faulty position measurement in the left outer actuator.
5	Faulty position measurement in the left inner actuator.
6	Faulty position measurement in the right inner actuator.
7	Faulty position measurement in the right outer actuator.

III. Switching Transients

ONCE the redundancy management has been formalized in the controller model and its correctness with respect to the requirements has been verified, the mode switching effects on transient behavior must be studied. To this end, the discrete event controller must be connected to a plant model that captures the behavior of the physical world, in this case, the hydraulics and mechanics of the elevator system. Details of the functioning of the hydraulic actuators can be found in previous work.^{12,16}

Different teams of engineers are usually responsible for the plant design and the controller design. In the case of the elevator system, the discrete event reactive control that captures the redundancy management requires a tool that facilitates modeling state transition behavior. It is preferable to represent the hydraulic actuators, on the other hand, in continuous-time models based on differential equations. Stateflow and Simulink were chosen as the respective tools because they facilitate the connection of the different models.

The top level of the plant model contains an actuators subsystem that consists of models of the four

hydraulic actuators: left outer actuator, left inner actuator, right inner actuator, and right outer actuator. The four actuators are connected to the two (left and right) elevators by the force exerted by the piston of the actuator cylinders. The model of each of the elevators is shown in Fig. 5(a). The elevators are modeled as a mass that is submitted to (i) the piston force generated by the actuator, (ii) a wind load that is proportional to the elevator deflection, and (iii) viscous friction that works against elevator movement.

The piston force is generated by the hydraulic actuators. The model of these actuators is shown in Fig. 5(b). The actuators consist of a servo valve, a spool valve, and a cylinder, as illustrated in Fig. 6. The servo valve controls the flow of oil into the cylinder as determined by the feedback control algorithm. When the actuator operates in *standby*, the piston in the servo valve shadows movement of the piston in the servo valve of the *active* actuator. To prevent the control pressure in the *standby* actuator from operating on the cylinder, its spool valve is switched to disallow a flow of oil between the servo valve and cylinder. In this state the actuator is not *active*, and the spool valve acts as a load to the cylinder, preventing interference with the control exerted by the redundant actuator.

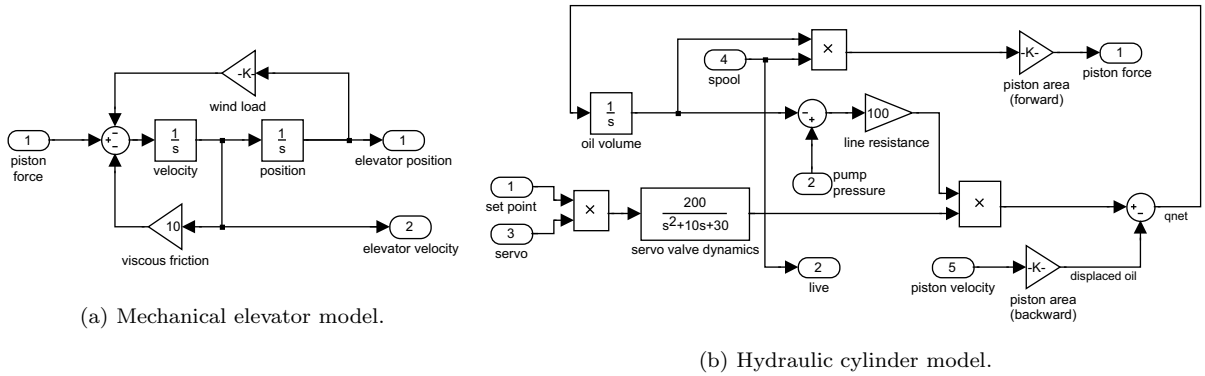


Figure 5. Mechanical and hydraulic domain models.

Similar to the spool valve, the servo valve can be turned off. In case the actuator is *off* or *isolated*, it does not shadow the control commands, i.e., the piston in the servo valve is kept at a fixed position. The movement of the piston is modeled by a second-order transfer function, accounting for the piston mass moving against the oil viscosity and elasticity. The position of the servo valve piston determines the flow of oil from the supply to the positioning cylinder, and, therefore, the effective piston force. The piston force is computed by the integrator ($\frac{1}{s}$) that represents the cylinder oil volume.

The feedback control law then takes as input the actuator deflection and produces the setpoint of the servo valve piston. The reactive supervisory control that includes the redundancy management described in Section II takes the failure status of the elevator position and velocity measurements and produces as output the state of the servo valve and spool valve, i.e., whether or not they are active.

Given the models of the hydraulic and mechanical parts of the elevator system, the performance of the different control laws and the switching transients can now be studied. Figure 7(a) shows a simulation of the elevator deflection in response to step changes in pilot input. At $t = 15$ [s] (marked by the broken line) a failure of the hydraulic circuit 1 (see Fig. 2) is introduced. The failure causes the actuator control to switch from IO to DL for both the left elevator and the right elevator. Simulation shows that the step response of the DL control law at $t = 20$ [s] is slower than the step response of the IO control law at $t = 10$ [s], although it does have less overshoot.

Figure 7(b) shows a switch from the DL control law back to the IO control law shortly after $t = 20$ [s] (indicated by the broken line). The switch in control law occurs during the step response of the DL control law as shown in Fig. 7(a). Even during such a setpoint change maneuver, a change in actuators has little effect on the elevator deflection because the actuator that is switched to become *active* was shadowing the

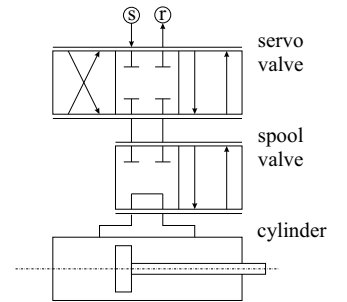


Figure 6. Schematic of a hydraulic actuator.

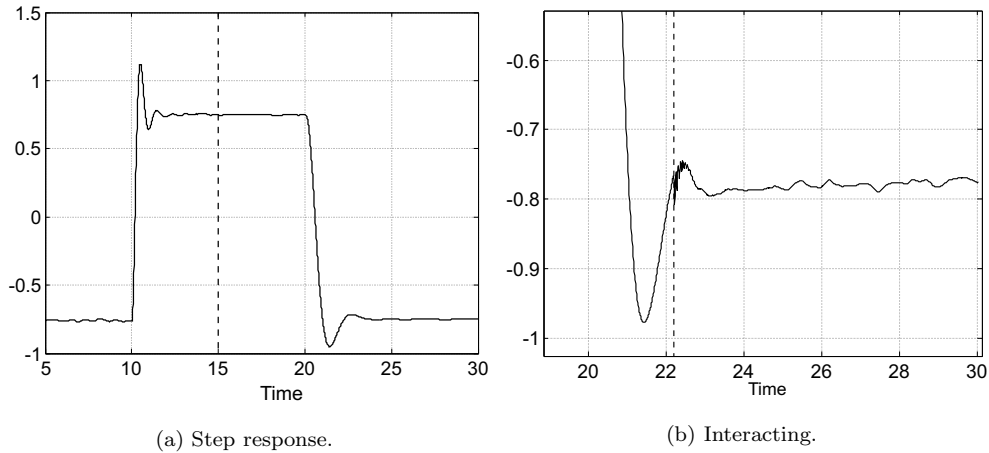


Figure 7. Step response of the different control laws.

actuator that was deactivated. Therefore, the piston in the servo valve had no initialization time, and it immediately produced the correct output force when the switch occurred. Note that some transients do occur, because of the change from a DL to IO control law.

Studying such transient effects on the subsystem level, as described above, and at the integrated system level (as presented in other work¹²) are key to Model-Based Design. Model-Based Design allows verification of dynamic behavior in different scenarios using models, obviating the need for a more expensive implementation, such as a physical prototype.

IV. Integrating the Redundancy Management Into the HL-20 Model

ONCE the elevator control system has been studied in detail, the effect of failures on overall behavior and the interaction with other control laws must be studied. This requires the integration of the control law into a more complete aircraft model.

In other work,⁹ an aircraft model of the HL-20 crew rescue vehicle shown in Fig. 1 was designed without failure-handling capabilities. The use of Simulink both for modeling the HL-20 as well as the redundancy management control, allows for a straightforward integration of the two.

In particular, the redundancy management is included in the aircraft model by reference. This integration is illustrated in Fig. 8, where the model block `FDIR_Application` is a reference to the entire fault detection, isolation, and reconfiguration functionality of the HL-20 primary attitude control systems. Part of this functionality is the redundancy management discussed in Section II. Other major systems that are referenced are the `FlightControlSystem_Application` and `GuidanceSystem_Application`.

The referencing of another model provides a powerful means for model exchange: The referenced models can be independently developed as stand-alone applications and tested to verify that they satisfy their requirements. They can also be easily integrated into configuration management systems, keeping them under version and configuration control. Modeling scalability is also enhanced, as models can reference multiple models and can themselves be referenced multiple times. Design components can be created with well defined interfaces, guaranteeing that the reference to the model will behave in the same way as the stand-alone model. This is not always true for subsystems that have been copied into a larger model, as context-dependent information can change the way the subsystem functions.

A test harness can be developed for the referenced model and, again, the model can be included as a reference. So, the same model can be exploited in different configurations and for different purposes. This streamlines the development process and prevents delays and problems caused by the use of outdated model versions.

Figure 9 shows four frames of a landing maneuver of the HL-20 that is animated using the simulation model described. The position and orientation of the HL-20 as computed by Simulink are visualized us-

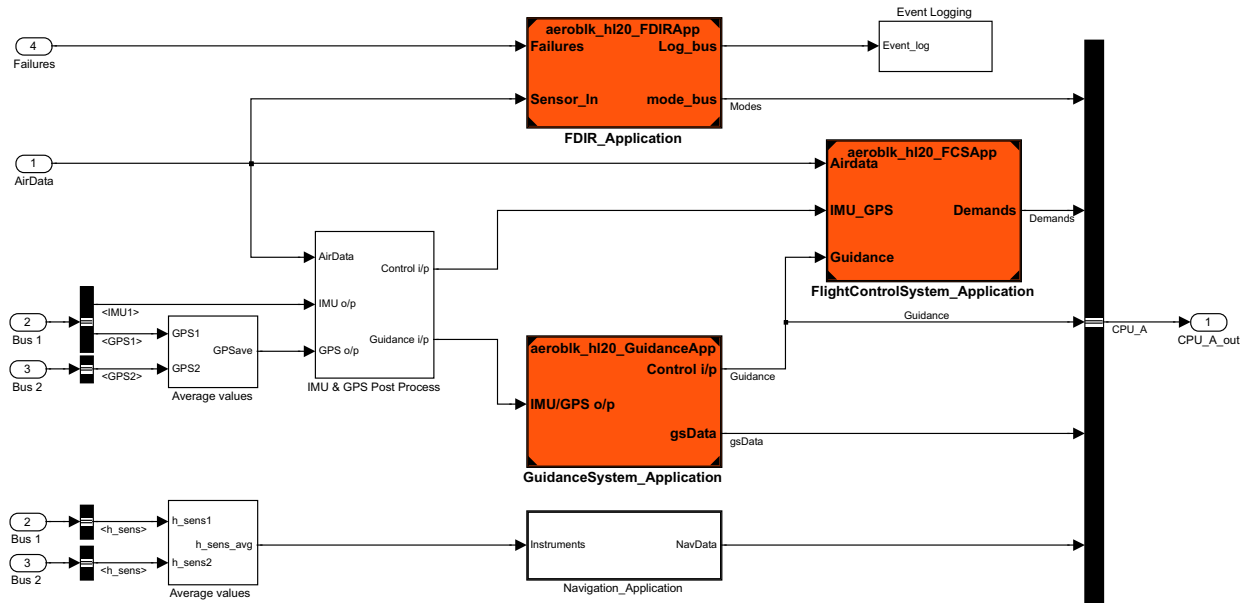


Figure 8. Integrating the elevator redundancy management into the HL-20 model.

ing FlightGear. Simulink and FlightGear communicate through their respective application programming interfaces (API). Note that the frames in Fig. 9 are not equally spaced in time.

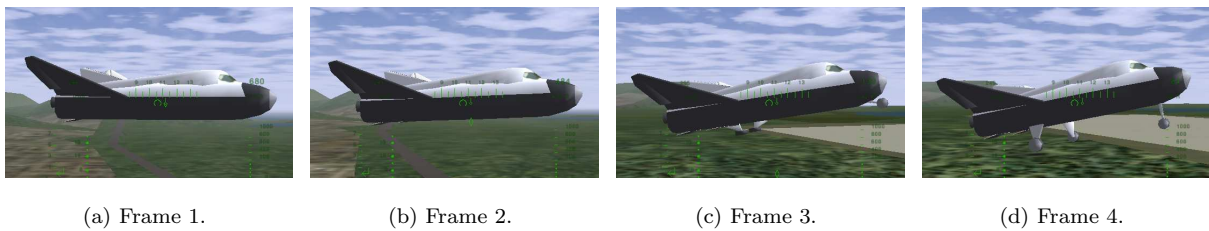


Figure 9. Animation of the attitude change during a landing maneuver.

V. Code Generation

THE redundancy management presented in Section II could be integrated with the HL-20 aircraft model at the model level because Simulink was used for both systems. Models of different systems may, however, use different platforms. Since many modeling environments have automatic code generators, C/C++ code has become a common denominator. Models can then be coupled and integrated at the code level, and only ‘glue’ code needs to be generated. Sophisticated code generators allow the user to precisely define the interface of the automatically generated code.

A. Model Transformation

Throughout the development of an embedded control system, code is generated for several different applications, such as simulation acceleration, software-in-the-loop, processor-in-the-loop, hardware-in-the-loop, and rapid prototyping.¹⁷ Typically, these applications rely on different hardware platforms, ranging from a desktop personal computer (PC) to fixed-point embedded processors, and they require different optimizations (e.g., static memory allocation instead of dynamic allocation to reduce response times).

To support model reuse, it is advantageous if the code for these different applications can be generated from the same model. Code generation then becomes a *model transformation* process, and by specifying different transformations, the different applications can be supported.

This approach enables systems engineers and software engineers to work with the same model. The systems engineers can design the integration of the parts and account for characteristics such as control law stability and response time while the software engineers, using the same model, take care of the scheduling, software partitioning, and data typing attributes.

B. Real-time Code

There are many definitions of ‘real-time’. In this paper, real-time means that the simulated behavior is aligned with the passing of actual, chronological time.

To generate real-time code, the response time of the code must not exceed a given upper limit. In particular, given the sample rate with which the code is executed, the computation of the output from the input must complete in less time than the time between samples. This implies that the computations cannot be iterative. As such, the model for which code is generated should apply a fixed-step numerical solver and cannot include *algebraic loops*. Algebraic loops are cyclic dependencies in computations in which the computation of a variable requires its value to be known. Simulink can minimize the occurrence of such cases.¹⁸

If these preconditions are satisfied, real-time amenable code can be automatically generated from a Simulink model by Real-Time Workshop[®] Embedded Coder¹⁹ (e.g., automatically generating code to appropriately handle data integrity in multi-tasking environments). In the redundancy management example in Section II, the code for the four statecharts that model the behavior does not require iterative solutions, and, therefore, a real-time version of that code can be generated.

C. Code Customization

Having real-time code automatically generated from a model is valuable; however, the benefit is greatly diminished if the automatically generated code cannot be integrated with existing software or packaged in a way that is compatible with the required coding standards.

This means that a code generator must include code packaging features that enable the generated code to comply with software styles and standards. The module packaging features of Real-Time Workshop Embedded Coder allow the use of templates (either pre-packaged or custom-designed) to organize the automatically generated code in a desired format.

There are three types of templates:

- A *code* template to organize C code function files. These are typically the .c files.
- A *data* template to organize the C code data files. These are typically the .h files.
- A *custom* template for advanced users to: (i) generate almost any type of .c and .h file, (ii) organize generated code into sections, (iii) generate calls to model functions (e.g., `model_step`), (iv) generate code to connect to model input and output, (v) generate a `main` function, and (vi) extract information about the model and the structure of the generated files.

The sophisticated model packaging features ensure that real-time code can be automatically generated in the correct form and with the desired organization, and that global data can be defined and referenced to interface with existing real-time flight simulator software.

VI. Conclusions

MODELS have consistently proven their value in all facets of system design. To fully realize their potential, however, they must be used for all key development tasks, including requirements capture, documentation, code generation, test, and verification. Model-Based Design supports this approach by enabling different design teams to build on and use the same model throughout the development process.

This paper described the use of Model-Based Design in the training of fault scenarios in aerospace. It showed how the discrete-event redundancy management system can be designed and validated in isolation

and then coupled with continuous-time models that capture the hydraulics and mechanics of the actuator. The resultant hybrid dynamic system can be used to study the transient effects of mode switching.

Model referencing features were then applied to the redundancy management functionality in a more extensive model of the HL-20 crew rescue vehicle. Including the redundancy management functionality in an aircraft model facilitated the use of simulation for training by enabling safe experimentation with failure scenarios.

This paper discussed how C/C++ code can be automatically generated in different forms and executed in real time, enabling the redundancy management system to be incorporated with existing flight simulation software.

References

- ¹Barnard, P., “Graphical Techniques for Aircraft Dynamic Model Development,” *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Providence, Rhode Island, Aug. 2004, CD-ROM.
- ²Aberg, R. and Gage, S., “Strategy for Successful Enterprise-Wide Modeling and Simulation with COTS Software,” *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Providence, Rhode Island, Aug. 2004, CD-ROM.
- ³SolidWorks, *Introducing SolidWorks*, SolidWorks, Concord, MA, 2002.
- ⁴Simulink, *Using Simulink*, The MathWorks, Natick, MA, June 2004.
- ⁵SimMechanics, *SimMechanics User’s Guide*, The MathWorks, Natick, MA, June 2004.
- ⁶Flatscher, R. G., “Metamodeling in EIA/CDIF Meta-Metamodel and Metamodels,” *ACM Transactions on Modeling and Computer Simulation*, Vol. 12, No. 4, 2002.
- ⁷de Lara, J. and Vangheluwe, H., “Defining Visual Notations and Their Manipulation through Meta-Modelling and Graph Transformation,” *Journal of Visual Languages and Computing*, Vol. 15, No. 3, June 2004.
- ⁸Müller-Glaser, K. D., Frick, G., Sax, E., and Kühn, M., “Multi-Paradigm Modeling in Embedded Systems Design,” *IEEE Transactions on Control System Technology*, Vol. 12, No. 2, March 2004.
- ⁹Gage, S., “NASA HL-20 Lifting Body Airframe Modeled with Simulink and the Aerospace Blockset,” *MATLAB Digest*, Vol. 10, No. 4, July 2002.
- ¹⁰FlightGear, *FlightGear is an open-source, multi-platform flight simulator*, <http://www.flightgear.org>, 2004.
- ¹¹Stateflow, *Stateflow User’s Guide*, The MathWorks, Natick, MA, June 2004.
- ¹²Mosterman, P. J., Remelhe, M. A. P., Engell, S., and Otter, M., “Simulation for Analysis of Aircraft Elevator Feedback and Redundancy Control,” *Modelling, Analysis, and Design of Hybrid Systems*, edited by S. Engell, G. Frehse, and E. Schnieder, Springer-Verlag, Berlin, 2002, pp. 369–390.
- ¹³Mai, G. and Schröder, M., “Simulation of a Flight Control Systems’ Redundancy Management System using StateMate MAGNUM,” 7. User group meeting STATEMATE, April 1999.
- ¹⁴Osder, S., “Practical View of Redundancy Management Application and Theory,” *Journal of Guidance, Control, and Dynamics*, Vol. 22, No. 1, January-February 1999, pp. 12–21.
- ¹⁵Seebeck, J., *Modellierung der Redundanzverwaltung von Flugzeugen am Beispiel des ATD durch Petrinetze und Umsetzung der Schaltlogik in C-Code zur Simulationssteuerung*, Diplomarbeit, Arbeitsbereich Flugzeugsystemtechnik, Technische Universität Hamburg-Harburg, 1998.
- ¹⁶Mosterman, P. J., “HYBRISIM – A Modeling and Simulation Environment for Hybrid Bond Graphs,” *Journal of Systems and Control Engineering*, Vol. 216, 2002, pp. 35–46, special issue paper.
- ¹⁷Mosterman, P. J., Prabhu, S., and Erkkinen, T., “An Industrial Embedded Control System Design Process,” *Proceedings of The Inaugural CDEN Design Conference*, edited by J. Angeles and P. Stuart, Montreal, Canada, July 2004, CD-ROM, ID: 02B6.
- ¹⁸Denckla, B. and Mosterman, P. J., “An Intermediate Representation and Its Application to the Analysis of Block Diagram Execution,” *Proceedings of the 2004 Summer Computer Simulation Conference*, edited by A. G. Bruzzone and E. Williams, San Jose, CA, July 2004, CD-ROM.
- ¹⁹Real-Time Workshop Embedded Coder, *Real-Time Workshop Embedded Coder User’s Guide*, The MathWorks, Natick, MA, June 2004.