Working in Teams: Modeling and Control Design within a Single Software Environment

Seth Popinchalk[*], John Glass[†], Rohit Shenoy[‡] and Rob Aberg[§]
The MathWorks Inc, Natick, MA, 01760

In this paper we will demonstrate the multi-user collaborative enhancement of a Simulink model of the NASA HL-20 with the goal of designing a new multi-loop, multi-component approach and landing control system. The model is maintained using a CVS-based configuration management system by multiple engineers. Using this model the approach and landing guidance control systems are designed with commercial off-the-shelf software. The control system is developed in a single design environment using a single model which gives insight into design trade-offs and loop interactions from a system-wide perspective. The workflow and tools that were used will be presented to coordinate with the modeling and control design work that was progressing simultaneously during the course of this project.

I. Introduction

WORKING on a large project often entails collaboration between multiple groups working on parallel or dependant tasks. Setting up a structured mechanism to collaborate efficiently while leveraging the benefits of Model-Based Design can help ensure that projects stay on schedule and are not derailed by challenges such as versioning of components and configuration management (CM).

One example where teams work together is in the design of control system. Traditionally, control systems are designed independently by multiple teams and assembled along with a plant model at a later date. Unfortunately the impact of the control system designs will not become apparent until the full system is available for analysis, causing unneeded iterations. To reduce the severity of the unexpected behavior, a more a systematic approach is required. An alternative approach is to design and optimize each of the control systems simultaneously using a single model for design and/or validation at regular time intervals, to meet the design requirements. The control systems should be designed in the modeling environment using both the nonlinear simulation and the automatic linearization of the model. A linearized model can be used for linear time and frequency domain analysis which is the basis for the most widely used control design techniques. Utilizing this design process allows multiple teams to work using the same models and share control designs. This process works well on the small projects but scales more effectively through the use of configuration management where multiple teams can work on a single model.

In this paper we will present a workflow and CM tools that were used in the design of an aircraft control system. These tools were used to coordinate the modeling and control design work that was progressing simultaneously during the course of this project. A key element in coordinating the efforts of multiple teams is the use of a logical architecture to divide the model and allow for component level development. By clearly defining interfaces between components before any design is implemented, and by maintaining those interfaces, we enable a collaborative approach to Model-Based Design. A flawed architecture can cause the interruption of work when merging models because of the added complexity of resolving many changes in common systems.

For this project, the work was broken into a modeling problem and the control design problem. The modeling team was tasked with enhancing an existing model, which only provided for roll, pitch, and yaw control, by adding a new aero-brake channel. The addition of the aero-brake enables the control design team to assert a larger margin of

[*] Senior Application Engineer, 3 Apple Hill Dr., Member AIAA.
[†] Simulink Control Tools Team Lead, 3 Apple Hill Dr., Member AIAA.
[‡] Control Design Market Manager, 3 Apple Hill Dr., Member AIAA.
[§] Simulink Development Manager, 3 Apple Hill Dr., Member AIAA.

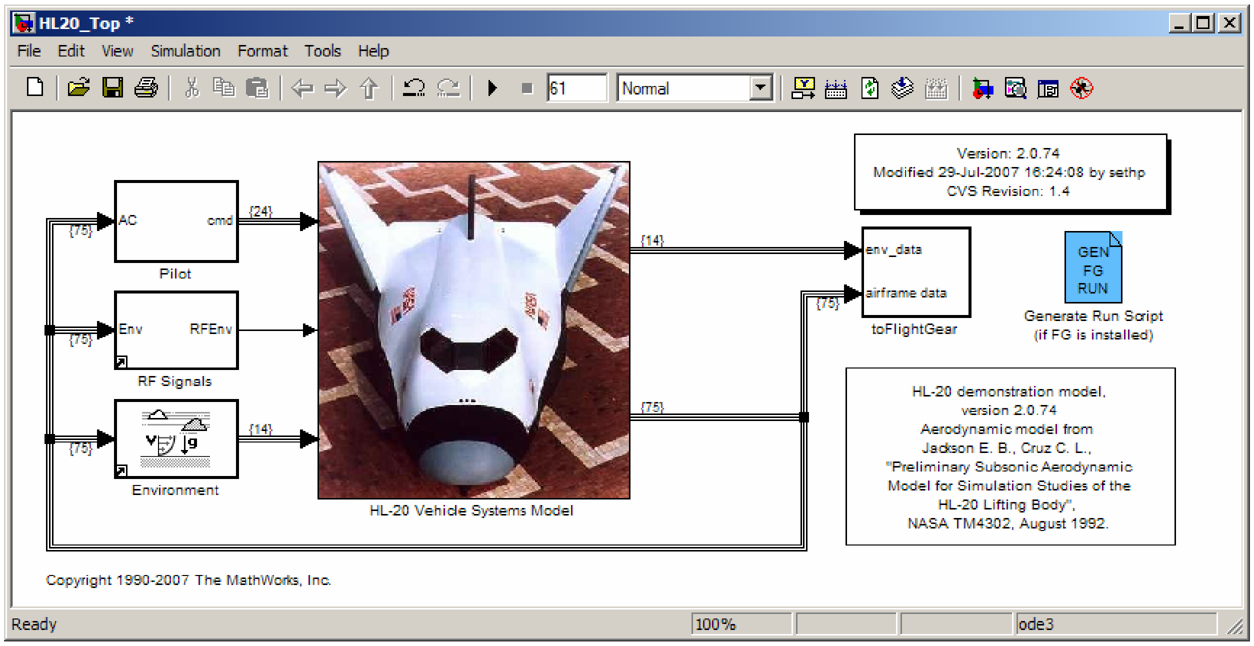
safety for the approach. It also enables the aircraft to come in faster with the option to expend the additional energy through maneuvers in adverse conditions, or to shed kinetic energy as the landing approaches and the margins narrow.

In aircraft there are multiple control systems with many variables, operating modes, and design requirements. In the case of the HL-20, there are multiple control systems that need to be designed and tuned. The challenge with an aircraft like the HL-20 is that multiple control systems, such as guidance approach and landing, need to be designed to work collaboratively in the appropriate operating mode to perform a task such as autonomously landing the unpowered aircraft.

II. Background

A project to upgrade the HL-20 model with enhanced controls and an aero-brake was initiated, and while the scope of the project was small enough to be undertaken by individuals without a rigorous configuration management system, it was decided that this project should leverage recent work done to provide a reference implementation of a Simulink® [9] based CM system [1].

The HL-20 model is a demonstration model provided with Aerospace Blockset [13], a software add-on to Simulink provided by The MathWorks. The HL-20 consists of an integrated six-degrees-of-freedom (6-DOF) vehicle model, avionics and sensor models as well as an environment model, and an interface to the third-party, open source software FlightGear[14] for visualization of simulation results. The vehicle model contains the vehicle airframe dynamics, including landing gear and control surface components. The avionics model provides a guidance control system distributed on three redundant processors. All of the components within the model are built up from more than 11,000 blocks. The model, shown in Figure 1, is configured to provide a simulation of the final 60 seconds of approach and landing of the HL-20



Version: 2.0.74
Modified 29-Jul-2007 16:24:08 by sethp
CVS Revision: 1.4

HL-20 demonstration model,
version 2.0.74
Aerodynamic model from
Jackson E. B., Cruz C. L.,
"Preliminary Subsonic Aerodynamic
Model for Simulation Studies of the
HL-20 Lifting Body",
NASA TM4302, August 1992.

Copyright 1990-2007 The MathWorks, Inc.

Figure 1. The HL-20 model provided with Aerospace Blockset.

The HL-20 model is a collection of Simulink models (MDL-files), and a MATLAB® [8] data file (MAT-file). There is one top-level model which contains components maintained as reference models and linked library blocks. Simulink provides the model reference feature to allow one model to reference another model as a component. These components are independent of the top model and can be simulated by themselves, or referenced by multiple models, multiple times. The model reference component has a defined interface, and any model that includes that component must conform to that interface. The HL-20 model also has a MAT-file that holds all the data used within the model. Simulink bus signals are used to aggregate system data into a structured output and pass information among components. Bus objects are used to define interfaces between components, and provide a specification for the interface.

The model has the full 6-DOF dynamics of the plant as well as the guidance controls implemented within it. We use this collection of files as an executable specification, working with it to understand the behavior of the system. This, in turn, helps us analyze, design and implement the controls system with Model-Based Design. If we were interested in also testing the system in a real-time environment, we could leverage Real-Time Workshop® [10] to generate embeddable C code for inclusion in a rapid prototyping environment. In a similar way, the plant model can be converted to C code for the purpose of hardware-in-the-loop testing.

The design of the guidance control systems for the HL-20 model has gone through a number of iterations over time. The first iteration incorporated an inner glideslope feedback controller [3] to track a prescribed glideslope trajectory. A heading control system was designed [4] to reject lateral wind gust disturbances as the aircraft followed the glideslope while landing. The next logical step is to add an energy management system that handles the control of the potential energy of the aircraft (the altitude) and the kinetic energy (the velocity). The goal of the redesign is to expand on the initial conditions that the aircraft can have when approaching the landing, and to essentially expand the overall robustness of the control strategy. The previous iteration started at a specific initial height (2275.5 m) and velocity (180 m/s). In Fig. 2, the ✓indicates a successful landing; and clearly with the current design, the aircraft cannot tolerate large deviations in the initial altitude.

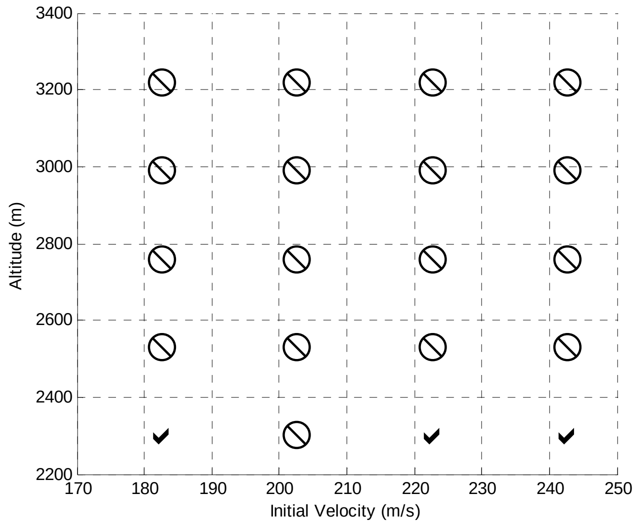


Figure 2 - (Initial strategy) Landing success of different initial conditions.

III. Overview of Version Control and Configuration Management

As any project commences, there is a need to track the progress and maintain proper versions of the files involved. If all members on the team create their own copy of the project files and work independently of each other, it will be impossible to determine which version of the files are the latest ones. Revision control can provide a central repository for project files and ensure that no two people make modifications to the repository at the same time. Revision control systems like CVS [15] provide a mechanism to record the version with the file and a method to retrieve an earlier version of a file if needed.

When a team member has been tasked with updating a specific component within the model, they will check out the model into their local working directory, which we refer to as a sandbox. A sandbox is typically used to try new designs and concepts, and acts as a working area when performing an upgrade task. This checked-out file becomes writable and they can proceed to edit and update the file. The remainder of the sandbox is left as read-only which provides the team member with confidence that they are only modifying this one file. As changes are made and files updated, the files will be checked back into CVS and given a new version. The older versions are still centrally available if they are needed.

A notable drawback to revision control by itself is that it is founded on the assumption that the latest versions of all files are the best ones to work with and that they indeed work together. Consider for example a top level model, like the HL-20, that references four other models. Simulink model reference components can define their interfaces using bus objects. If an interface was to change, that one component would be updated to meet the new interface. In order to achieve a working model, all consumers of that interface would also need to be changed. If this does not happen, the current version of every file in CVS will not work together. Performing an *update diagram* command causes Simulink to compile the model and ensure consistency between components (among many other actions). If an inconsistent interface is found, Simulink will report the error and identify it. When this is the case, the latest revisions of all the files in CVS represent a broken model which is waiting to be fixed. Configuration management is the solution to this type of revision control problem.

Configuration management is the process of tracking a collection of files and marking them as a configuration. As the example in Figure 3 shows, a configuration represents a collection of files we know something about. In some cases it is known that a configuration is a working set of files. In others it may be known that these have been fully tested and are ready for release. The process of using configuration management is built on top of the revision control process. In addition to checking out and checking in files, we have the added steps associated with labeling the specific version of all files that are part of that configuration.

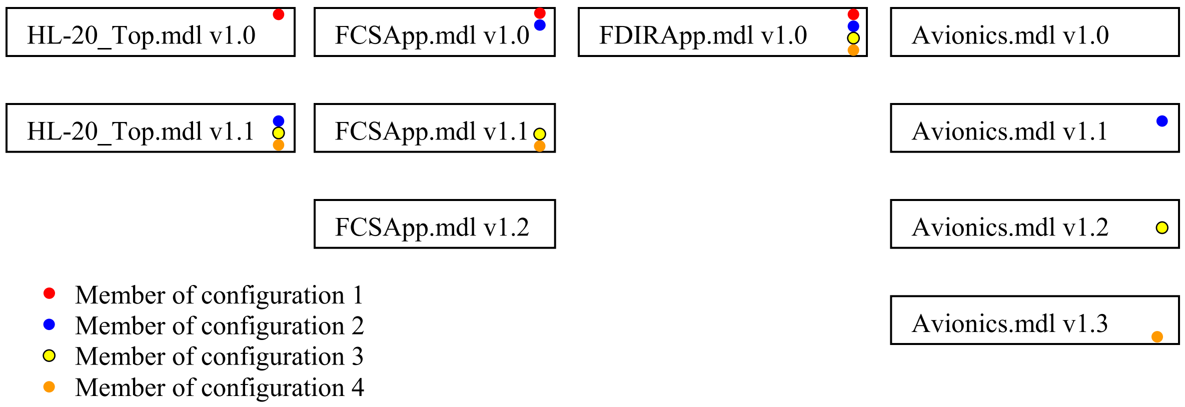
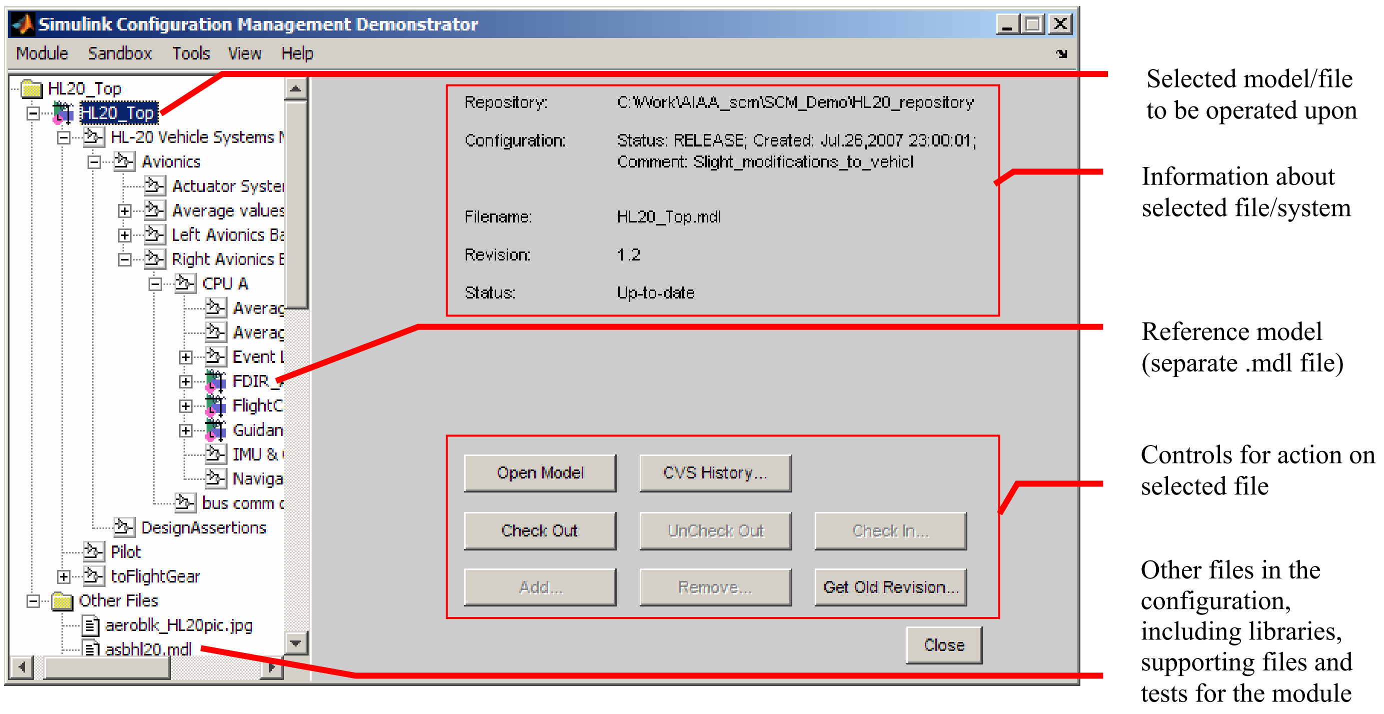


Figure 3. Individual files have versions which increment with each check in of a changed file; a collection of these files is defined as a configuration and is tagged as such within CVS.

Configuration management software is available which can be integrated into any design process, however the interfaces are not customized for Model-Based Design. Using Simulink for Model-Based Design we wanted to have an integrated tool which would do more than just manage the CVS interface. The Simulink Configuration Management Demonstrator (SCM) provided an interface which was built around the model view of the project Figure 4. The SCM demonstration is a reference implementation of a custom interface to a third-party configuration management tool, CVS. The aim of the demo is to illustrate successful configuration management workflows. The SCM is written entirely in m-code, and works with The MathWorks software, from Release 2006a onwards on Microsoft Windows platforms. It consists of a graphical user interface, which provides a user with easy access to CM.[**]

The display of the SCM provides a view of the model with all systems and reference models as nodes within a tree, a logical way to think of the hierarchy of the model. When working on the model reference components it is easy to use the tree to select the appropriate file that must be checked out for editing. The SCM also provided a simple test (via a Simulink update diagram) to verify that a given sandbox represented a valid configuration. The SCM was built to interface to TortoiseCVS [15], however it is conceptually possible to integrate the SCM with any third-party configuration management software by implementing the interface functions.

[**] The demo is available by contacting the authors.



Repository: C:\Work\AIAA_scm\SCM_Demo\HL20_repository
Configuration: Status: RELEASE; Created: Jul.26,2007 23:00:01; Comment: Slight_modifications_to_vehicl
Filename: HL20_Top.mdl
Revision: 1.2
Status: Up-to-date

Figure 4. The Simulink® configuration management interface.

At the initial planning meeting for the project we introduced the idea of using the configuration management environment to allow the two project teams to check out and lock files they were working on, and then check them back in when the work was complete. This is the multi-committer workflow, shown in Figure 5, where all contributors have the permissions to check out and check in files in the repository. For this to work the model would need to be sufficiently modular where only the interfaces would be shared by the components. This locking of files would remove the need to sit together to merge any files that were updated by both teams, and we believed it would simplify the work.

Each team member would begin by installing the Simulink Configuration Management demonstrator and configuring their installation to use the central repository on the network. This would allow them to use the SCM interface to manage their work using the following proposed workflow:

1. Select the module to "get". The SCM would then copy the latest version of the files within that module to the user's sandbox.
2. Check out files, make edits, check in files. Repeat until task is finished.
3. Test the files in the sandbox to ensure that they are a valid, working set of files.
4. Convene a meeting to qualify the latest changes in the repository.
5. Create a new configuration within the repository.

As changes accumulated within the repository the teams would occasionally convene a meeting to decide when it was appropriate to make a new release. All files would be checked into the repository and labeled with CVS "sticky tags" to identify them as a working configuration. This collection of files could then be retrieved at a later time and used as a starting point for work. The planned process would involve getting the latest versions of all files checked into the repository and then testing the collection to determine its worthiness as a good configuration. If the members of the teams reviewing the changes were satisfied, a new configuration would be made and tagged with a brief identifying description.

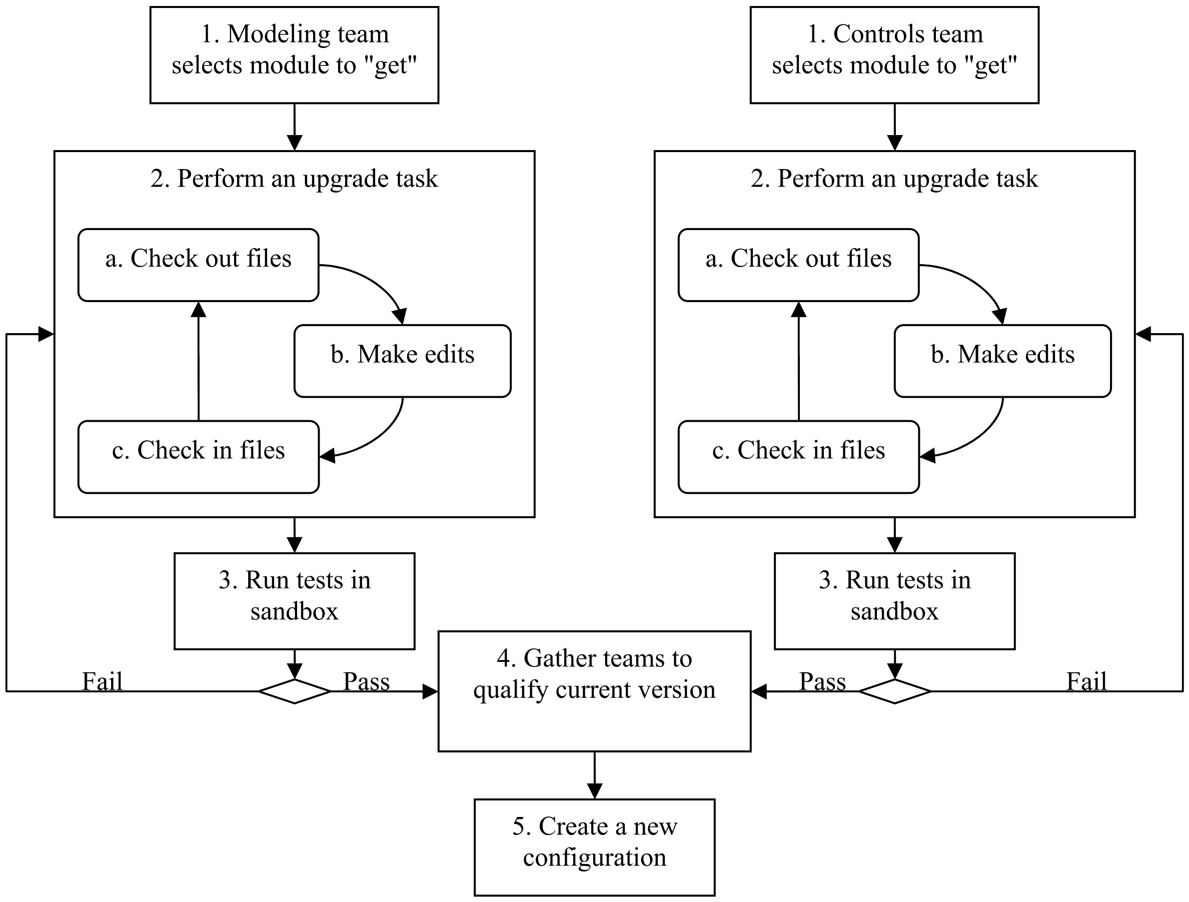

1. Modeling team selects module to "get"
1. Controls team selects module to "get"
2. Perform an upgrade task
a. Check out files
b. Make edits
c. Check in files
2. Perform an upgrade task
a. Check out files
b. Make edits
c. Check in files
3. Run tests in sandbox
3. Run tests in sandbox
Fail
Pass
4. Gather teams to qualify current version
Pass
Fail
5. Create a new configuration

Figure 5 - The multi-committer workflow.

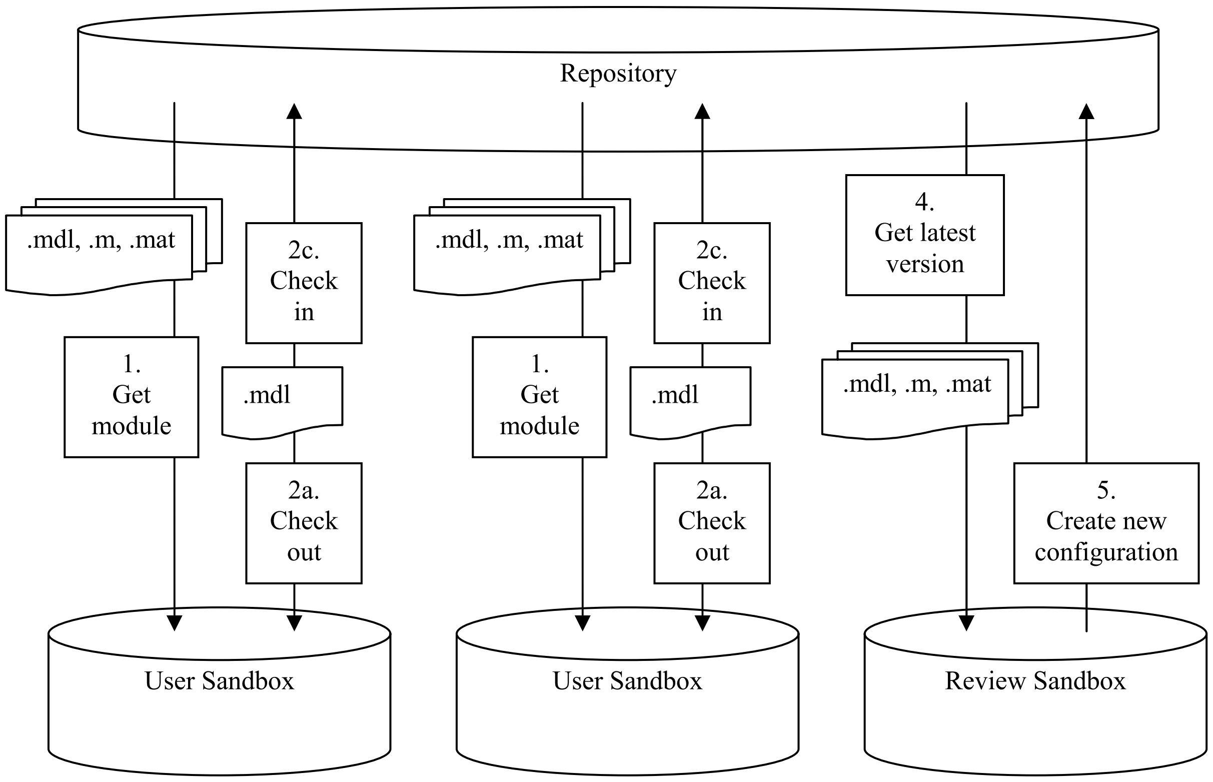


Figure 6. The multi-committer file flow.

In the multi-committer workflow team members maintain their own working sandbox that reflects a given module or the system. A user sandbox, or a special review committee sandbox must be created when considering the creation of a new configuration. Additional testing before submitting the configuration back to the repository is done in the review sandbox.

IV. Control Team Design of the Energy Management Control System

The current implementation of the HL-20 demo implements a guidance control system that only handles the landing of the aircraft on the inner glideslope trajectory. The final long term goal for the control team was to implement a landing control system to take the aircraft from a high altitude, high velocity condition on an outer glideslope, through a pre-flare maneuver, to a low altitude inner glideslope trajectory, as shown in Figure 7 [3].

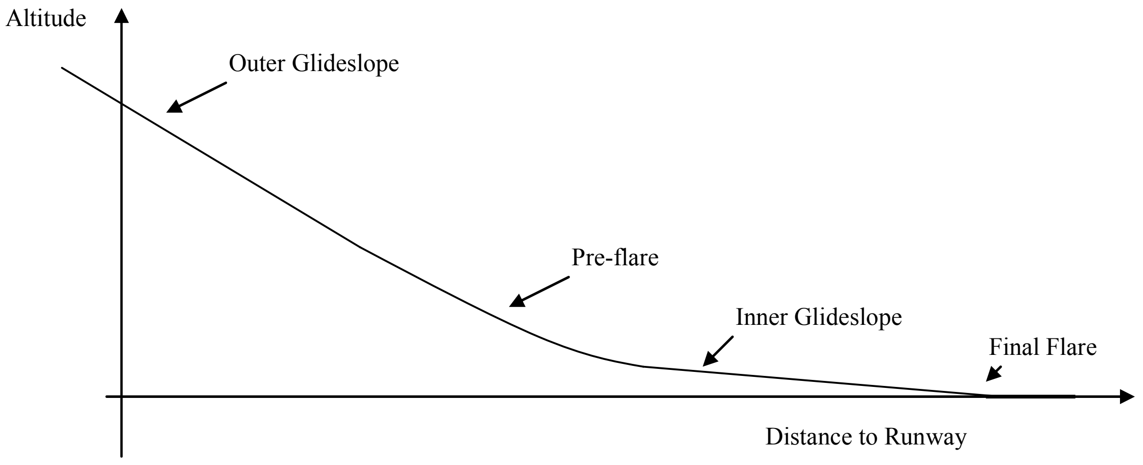


Figure 7. Aircraft landing maneuvers.

By adding an energy management system that handles the control of the aircraft's potential and kinetic energy (altitude and velocity), the redesign would expand the aircraft's possible initial conditions as it approaches a landing.

The first step in the design was to reorganize the Guidance Control Application shown in Figure 8. The two main components are subsystems labeled Phi Control and the new Energy Management Control. The Phi Control subsystem regulates the heading of the aircraft. This re-organization was the first revision checked into the CM system.

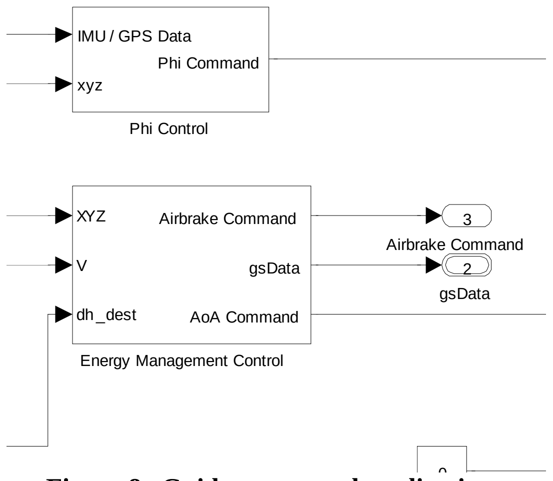


Figure 8. Guidance control application.

The role of the *energy management control system* is to process the guidance data which contains the coordinates of the aircraft relative to the runway (signal XYZ), the velocity of the aircraft (signal V), and the altitude of the aircraft relative to the runway (dh_dest). In the energy management control system there are two components, as shown in Fig. 9:

1. The *Airbrake Control* uses the estimate of the velocity of the aircraft along with the distance to the runway to compute the command for the airbrake (signal Airbrake Command). The airbrake command is fed to the physical aircraft component (designed by the modeling team) which controls the flight surfaces.
2. The *Angle of Attack (AoA) Control* uses the measurement of the altitude of the aircraft and the target distance to the runway to compute the command for the angle of attack for the aircraft.

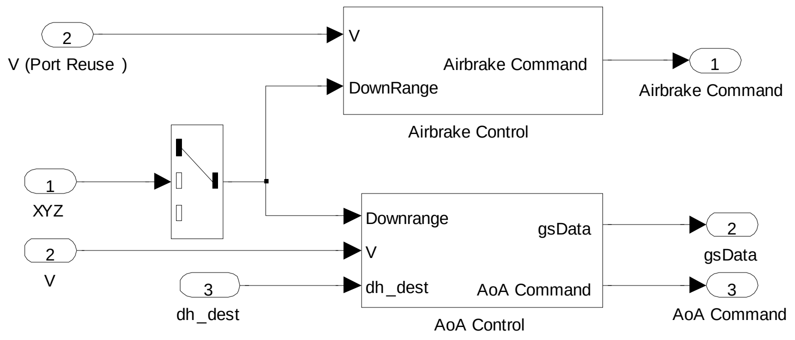


Figure 9. Energy management control system.

The *airbrake control system*, shown in Figure 10, was constructed to track a prescribed velocity trajectory given the distance to the runway. In this case the Discrete-Time Integrator is designed to open up the airbrakes when the measured velocity is more than the desired velocity. The integrator is designed to reset once the measured velocity meets the desired velocity.

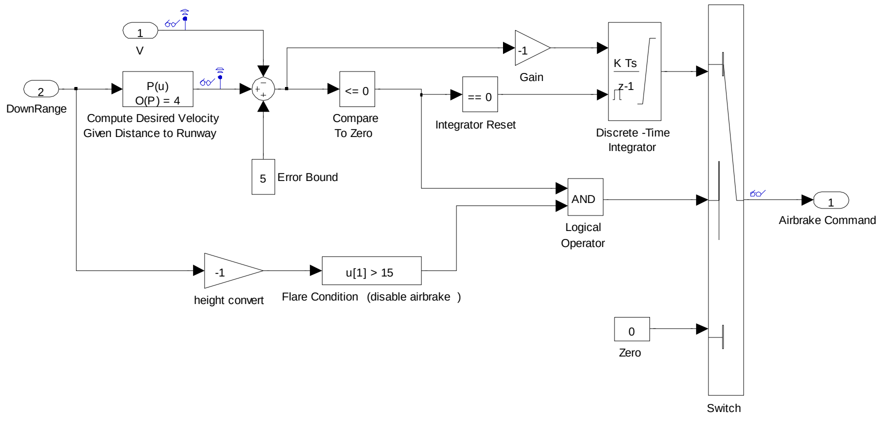


Figure 10. Airbrake control system.

The *glideslope control system* comprises two components:

- The *Glideslope Error* subsystem is used to calculate the error in the glideslope that is fed to the Glideslope Feedback Controller.
- The *Glideslope Feedback Controller* takes the error in the glideslope and calculates the desired angle of attack that is fed to the inner flight control system.

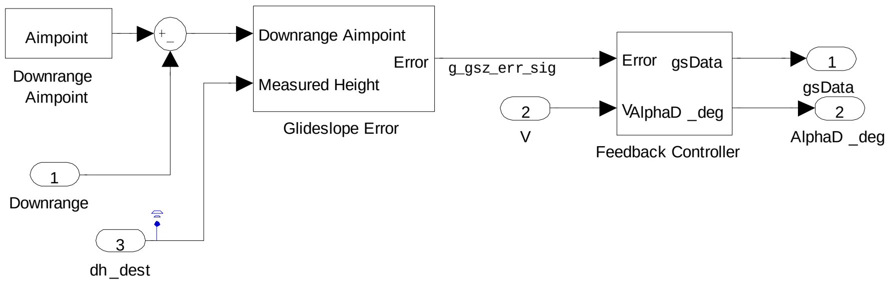


Figure 11. Glideslope control system.

The first step was to design the feedback controller. To do this, Simulink® Control Design [12] software was used to tune the control elements below. For a detailed explanation of the use of these tools see [5]. The Simulink Control Design software works with the Control System Toolbox [11] software to facilitate linear control design.

Using the Simulink Control Design Graphical User Interface, we set up the control problem by specifying three controller blocks labeled Gain, Phase Advance, and Phase Advance 1, as shown in Figure 12. The second step was to select the closed-loop signals of interest for the design. In this case the closed-loop input signal is the desired height of the aircraft and the closed-loop output signal is the actual height of the aircraft. Using this information, Simulink Control Design automatically computes linear approximations of the model, using specified operating points, and identifies feedback loops to be used in the design. Because we were working with a high fidelity model the linearized model includes higher order effects such as the actuator dynamics. The linearization algorithm used in Simulink Control Design employs an exact linearization about a specific operating point. The accuracy of the linearization for perturbations about the linearized operating point is highly dependent on the nonlinearities in a model. To better understand the impact of the nonlinearities upon the range of accuracy of the linearization, Simulink Control Design provides tools for computing operating points using trim or simulation snapshots. This allows for the variations in the linearization due to the nonlinearities at different operating points to be easily studied.

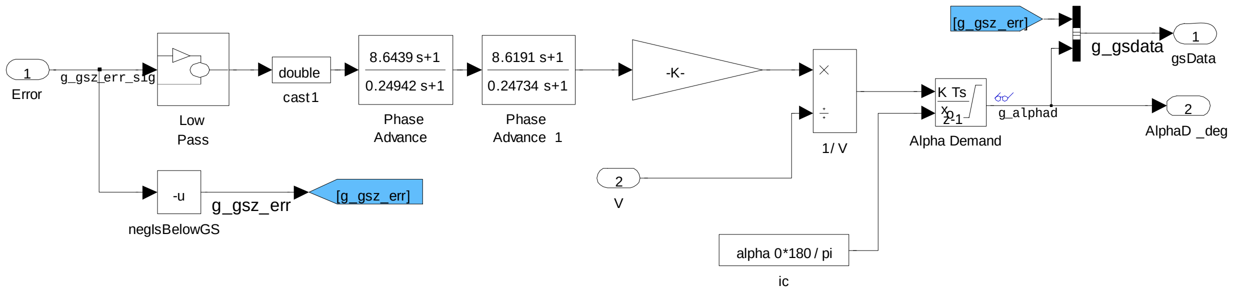


Figure 12. Glideslope feedback control system.

Simulink Control Design can be used to tune control system using a variety of classical control design methodologies, such as Bode and root locus plots, along with automated synthesis techniques, such as linear quadratic Gaussian (LQG) and H-infinity (H_∞) loop shaping, within a single user interface. The benefit of having these techniques in a single interface is that a control design problem can be examined using a number of different methodologies.

In this design, we wanted to produce a controller that had a limited bandwidth along with a fairly simple structure with a few states. The design was completed using a Nichols plot, shown in Figure 13, to tune each of the control elements. The Nichols plot design tool has an interactive interface that allows graphical shaping of the pole and zero locations of the controllers. This interactivity helped to visualize the tradeoff between the bandwidth, order, and stability margins of the control system.

The glideslope loop was designed using a Nichols chart to meet the design requirement of having a phase margin greater than 35 degrees. The final controller (shown in the continuous domain for convenience here, but discretized to a sample rate of 60 Hz in the implementation) contains three states and is shown below.

$$K(s)=\frac{1}{180}\frac{\left(\frac{s}{0.1156}+1\right)}{\left(\frac{s}{4}+1\right)}\frac{\left(\frac{s}{0.1156}+1\right)}{\left(\frac{s}{4.04}+1\right)}\frac{1}{s}$$

We have been able to limit the frequency of the fastest pole of the controller to be nearly 4 rad/s. Given the sample rate of 60 Hz we were able to easily implement the controller as discrete elements.

Figure 14 shows the closed-loop, unit step response about the operating point for which the Simulink model was linearized.

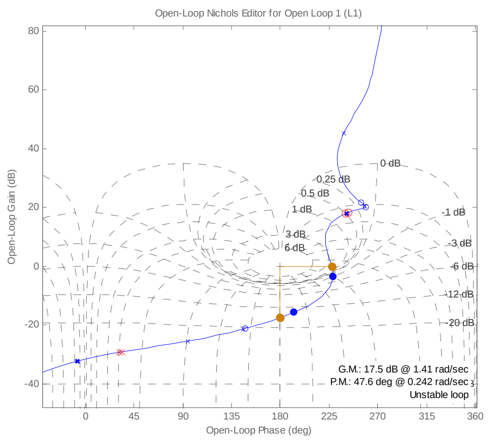


Figure 13. Open-loop Nichols plot of the glideslope feedback control system.

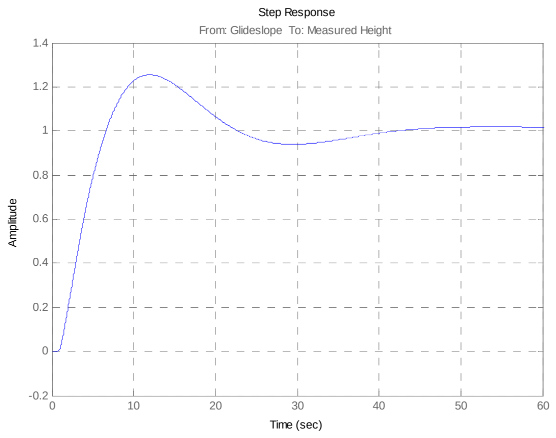


Figure 14. Linear closed-loop step response of the glideslope feedback control system.

Upon implementing this feedback loop for the first time, we found that when there were large deviations between the measured and desired altitude, there was significant integrator wind-up. In many cases this wind-up ultimately led to the glideslope control system becoming unstable. To limit this wind-up, the Glideslope Error subsystem was designed using two strategies. The first strategy checked into the CM system is shown in Figure 15. This subsystem filters the reference signal to limit the brunt of the changes from the reference input. The filter was designed to limit the rate of change of reference signal when there is a large deviation between the glideslope trajectory and the measured height.

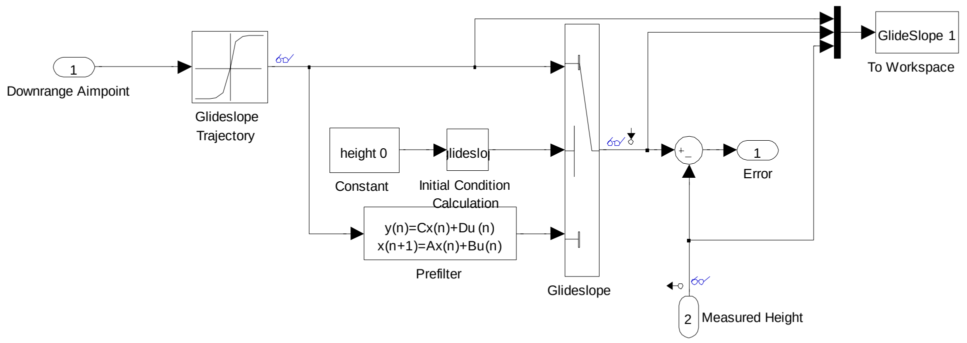


Figure 15. Strategy 1 – Prefilter-based Glideslope Error subsystem

A second strategy was developed and checked into the CM System used a weighting function on the error signal entering the glideslope feedback loop. The implementation of the Glideslope Error subsystem is show in Figure 16.

The strategy uses a linear function (the polynomial block named *Weighting on Error Signal*) which was designed to weight the error to be zero at the beginning of the glideslope and one when the aircraft is at the runway. The goal here is to gradually react to the full effect of the error causing integrator wind-up, which in turn would make the aircraft unstable.

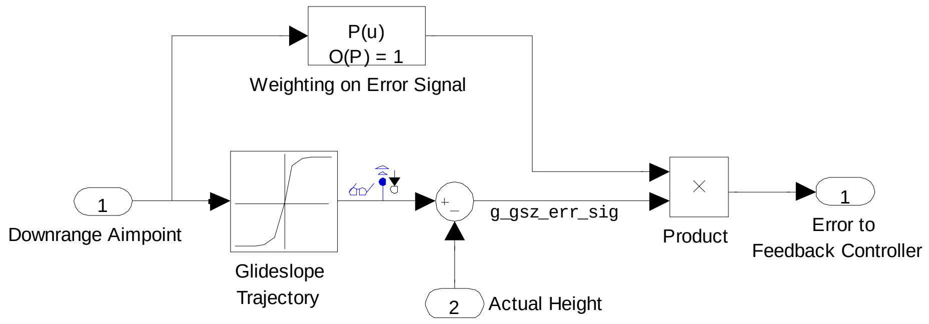


Figure 16. Strategy 2 – Scaled error-based Glideslope Error subsystem

V. Understanding the Multi-Committer Model of CM

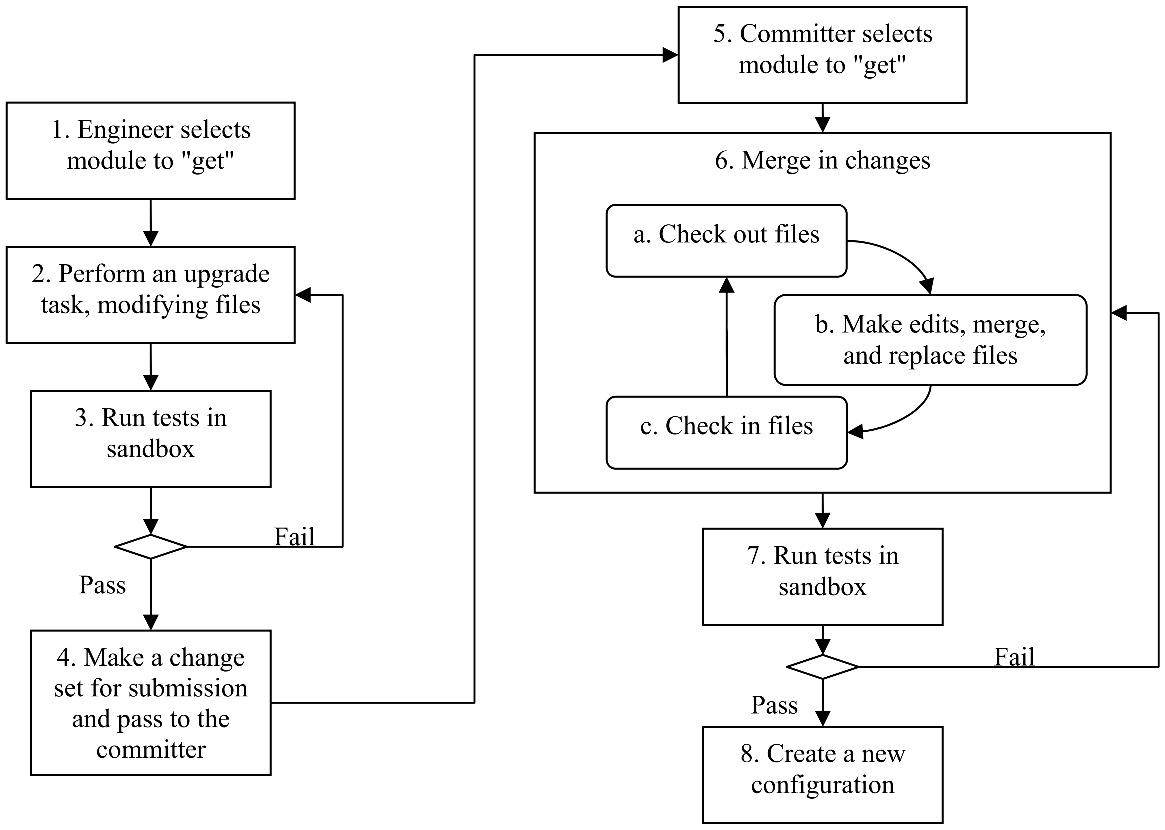
The multi-committer model of configuration management requires strong communication between the teams, as well as regular meetings to review and create new configurations. The multi-committer system also requires that each developer become familiar with the tools, which can be a challenge when there are many people working on the project. In addition, detailed training and procedures must be in place at the start of the project in order for the multi-committer model to succeed. Another factor is that a multi-committer process requires a robust automated testing apparatus to ensure that the many transactions contributed by each developer are repeatedly tested consistently. For example, the modeling team could be tasked with developing the right tests for the model. As an engineer finishes an upgrade task, solid testing is required to determine if the component is in a state that is ready for check in.

In addition to testing, teams must have in place style guidelines and naming conventions to ensure consistency across submissions to CVS. The guidelines can start off with very simple rules for the process of naming files and components, as well as where to make changes when there may be many options available. Style guidelines should grow as needed to manage the complexity of work being done in a project. Guidelines might include a list of blocks that are acceptable in the model. Based on previous experience and requirements such as the end goal for the model, a library could be created which would enable modelers to easily drag blocks into their model from the approved pallet. An example of a block that might be prohibited would be one whose execution depends on MATLAB since it is not supported for code generation.

VI. Understanding the Single-Committer Workflow

Another approach that seemed more scalable for our project was the single-committer workflow (Figure 17 and 18). The single-committer workflow is similar to the multi-committer, however all changes are submitted through one person who integrates them into the current revision of the model. The teams submitting changes provide the single committer with a change set (a bundle of files) which represent the modifications they have made to implement a new feature. The necessary files which had been modified are then checked out, replaced, and then tested in the committer sandbox. The single committer can make a new configuration at any time in the process, and the latest configuration is then provided as a baseline for further development.

For our project, the modeling team took the role of the committer and had responsibility for taking modifications from the control design team and merging them with the latest configuration. The control design team would provide a change set which represented a new aero-brake design strategy.



1. Engineer selects module to "get"
2. Perform an upgrade task, modifying files
3. Run tests in sandbox
Fail
Pass
4. Make a change set for submission and pass to the committer
5. Committer selects module to "get"
6. Merge in changes
a. Check out files
b. Make edits, merge, and replace files
c. Check in files
7. Run tests in sandbox
Fail
Pass
8. Create a new configuration

Figure 17. The single-committer workflow.

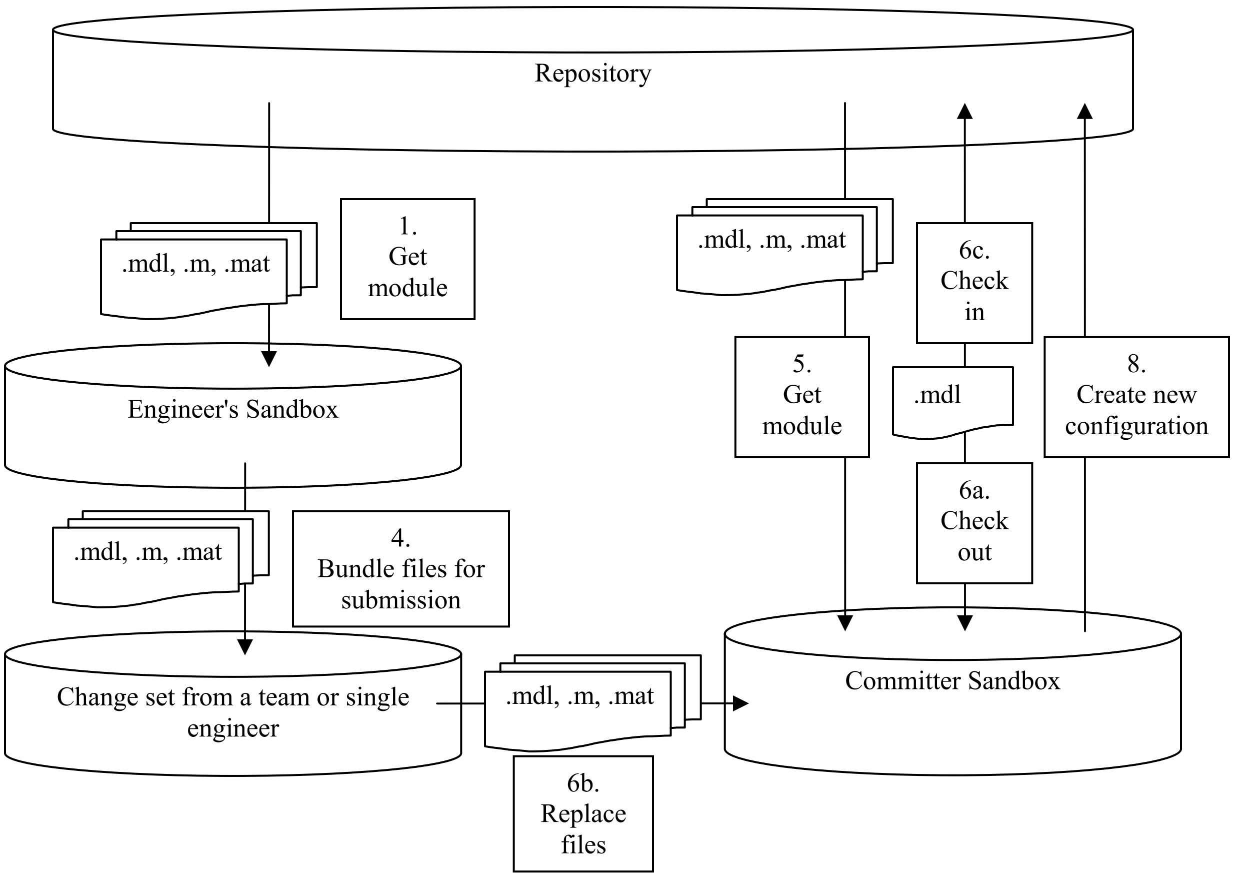
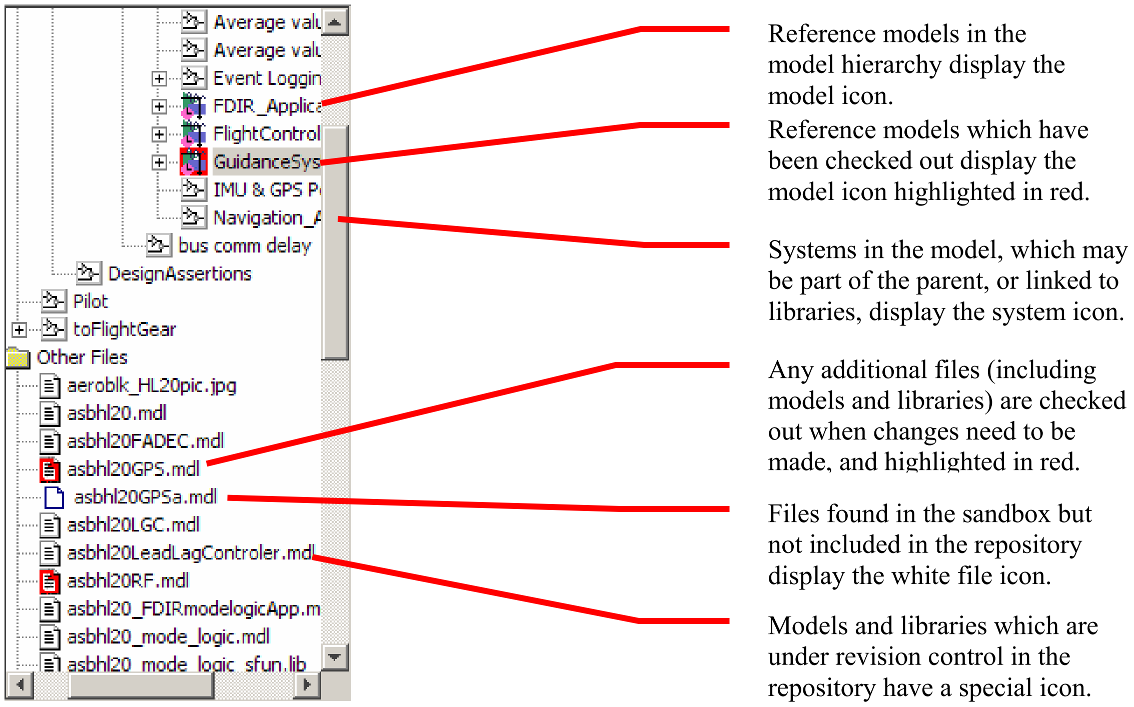


Figure 18. The single-committer file flow

Adding changes and maintaining a consistent model becomes a full time job for the committer, who must review the submission files and determine the best method to check in the latest version. In some cases, the committer can simplify the collection of changes made in order to reduce the number of files that needed to be modified. For example, when the first control design strategy was implemented, new data containing the desired velocity was introduced. The controls team added the data as part of the initialization commands in the root model file. The initialization commands for the root model originally just loaded a data file which represented the collection of all data needed by the model, and now it had been modified to run two more commands which would load the aero-brake controller data separately. The committer was able to simplify this by incorporating the aero-brake controller data directly into the original master data file, rather then changing the initialization code of the root model.

Replacing as few files as possible in the committer sandbox is the objective when merging a submission with the latest configuration. In the Simulink environment, the model represents an executable specification for the system. By compiling the model and running tests, it was very easy to determine if the changes made in the committer sandbox were complete and had added the new functionality. Another tool that could be used to track which files had been modified would be the checksum utilities available within Simulink. These utilities provide a checksum based on the functional components of a model and can be used to identify when a model may have been updated with a change which affects its functionality (as opposed to simple graphical differences like moving a block, which are ignored by the functional checksum).

The SCM interface provides the committer with important information about the current sandbox and the status of files within it, as shown in Figure 19. As a change set is merged with the current sandbox, the best practice is to check out the minimum set of files which require modifications. New files which may have been added to the sandbox can be identified and added to the repository and put under revision control.



Reference models in the model hierarchy display the model icon.

Reference models which have been checked out display the model icon highlighted in red.

Systems in the model, which may be part of the parent, or linked to libraries, display the system icon.

Any additional files (including models and libraries) are checked out when changes need to be made, and highlighted in red.

Files found in the sandbox but not included in the repository display the white file icon.

Models and libraries which are under revision control in the repository have a special icon.

Figure 19. The SCM interface provides information on the current sandbox and the status of files in it.

Within the single-committer model, it is up to the committer to choose when to create a new configuration. This would usually correspond to checking in the file changes needed to gain the newly designed functionality submitted by a team. Our process involved running the tests to determine if the model was still a good one, checking out the baseline data and diary file, and generating a new baseline. The baseline would provide test data artifacts which prove the validity of the new collection of files as a configuration. Once testing was complete and the sandbox was deemed to be in a working state, the configuration was created and a descriptive tag would be added.

The process could also be more involved and use different levels of testing. A central repository could hold untested versions of other users' sandboxes, which would be tagged as UNTESTED. When the quality team or the author who submitted the UNTESTED configuration tested it, a new TESTED configuration could then be made. If it was determined that enough changes had been made to warrant a release, a reviewer or review team would be instructed to get the new FORREVIEW configuration from the repository. If the configuration was determined to be in a releasable state, the new RELEASE configuration would be created. This would then be broadcast to other teams, who would be directed to get the latest release and continue working from that configuration.

VII. Applying Software Design Principles to the Challenge of Parallel Development

Common software design principles [6] were key to the success of this Model-Based Design project. These are listed here:

- Communicate clearly
- Think in modules
- Automate wherever possible
- Test early and often

By most accounts, the HL-20 model is a medium sized model consisting of more than 11,000 blocks. The challenges of collaborative development increase when working on a large model with many components. The best approach to addressing these challenges is to have well defined processes and smart tools for the job. Because our model was already implemented in a modular manner and the teams were communicating, we put our focus on testing.

Continuous and regular testing is a crucial element of Model-Based Design. It drives down the cost of errors by finding them and fixing them as early as possible. There are many types of tests which can be incorporated, and the possibilities for automating the testing are numerous. In the Simulink environment, the flexibility of the MATLAB language is available for scripting of the tests. The types of tests that can be written generally fall into a few categories: smoke tests, functional tests, compliance tests, unit tests, coverage tests, and baseline/analysis tests.

Smoke tests are the simplest and provide only verification that a catastrophic failure was not encountered while running the test. These are generally very efficient to run and may be selected as the minimum requirement for checking in a file. We developed only basic tests which could be automated to allow for immediate verification of a design change. The simplest test we had was tSmokeBasic (Appendix A). This smoke test would do nothing more than open the model, run it, and check for any errors after running the model. If there were no errors returned, then the test passed. This is the basic turn it on and "look for smoke" test.

The functional tests should involve a more elaborate testing of the model. This could be a comparison of the simulation results to a baseline where a tolerance is given, and within that tolerance, the model passes. For the HL-20 we developed a test to determine if the model would land correctly, tHitRunway (Appendix A). This test script would open the model, attach a logging signal to the main vehicle bus, and log signal data while the model ran. This logged data was then analyzed to determine the first point in the simulation where weight on the wheels of the landing gear was detected, and from this determine the point of touchdown on the runway. If the vehicle landed within the first half of the runway the test was deemed successful and passed.

Compliance tests can ensure functional design characteristics as well as enforce style guidelines. A compliance test might check for banned blocks in the model, or notify the user of a mismatch in settings which had been set forth in the modeling guidelines. If style guidelines are used, the Simulink model advisor provides a framework for checking common style rules as well as the ability to customize checks to be run on the model.

Individual components can also be tested with unit tests. The unit test provides verification of functionality for a given unit of the model. When a component is compared against its specification, a series of tests can be developed to measure whether or not the component meets the specification requirements.

When considering a model for its worthiness as the basis for creating mission critical software, it is important to ensure complete testing. Coverage tests can be used to ensure that all logical paths within a model have been tested and to ensure that under operating conditions the software will not fail. This testing is required for certification of flight software and there are many resources available for anyone who wants to engage in this type of testing [7].

The final and most elaborate test we used was built on top of the Hit Runway test, tMeasureEnvelope (Appendix A). This was an analysis test which would initialize the model at different starting conditions over a range of altitudes and velocities, and then run the model to see if it hit the runway. The resulting data gathered was used to produce the plots found in Figure 2 and Figure 22. We did not use the test to determine a pass fail criteria, however the limits of the altitude and velocity could be measured and used to compare against a baseline and this would determine if any regressions in model capability had occurred.

VIII. Benefits of using the CM System

As the number of files involved with a project increase and the number of teams working together increase, the benefits of having configuration management become more obvious. In our project to upgrade the HL-20 model, the primary benefit was that it allowed us to implement different strategies for the glideslope control system, and capture the configuration that represented that complete, working design. We were then able to easily recover each of these configurations for comparison studies.

The modular architecture used in the project enabled easy switching between designs. By using the Simulink model reference feature the glideslope control system was implemented in a separate model that was referenced by the full simulation. This allowed us to quickly change the configuration since we only needed to get a single model file to change the configuration for these comparison studies.

Another benefit of having configurations was that it allowed us to recover configurations after we may have corrupted our sandbox while in an exploratory mode. In many instances ideas for different control strategies did not meet our requirements and we were able to recover the original configuration and start over in the next trial. A meticulous engineer could achieve the same benefit from keeping a backup copy of the files in a separate directory from their sandbox, however the CM system ensures that a known working configuration is the correct one. Simple mistakes like accidentally overwriting the backup copy of a file can result in loss of work. Additionally, accidental corruption of the version control system is less likely because there is never a need to explore the directories which hold the repository, and the files in that area are read only.

Having the single-committer system encouraged regular meetings to discuss modeling strategies to enforce consistency. Because the committer was taking full responsibility for updating the configurations with each submitted change set, the committer is heavily involved with tracking the actual implementation. When questions would arise about the details of a change that was submitted, the committer would need to immediately raise this with the team member responsible for the change. This provides an informal level of design review in the process.

IX. Simulation Results

The new control system was tested by running simulations where the aircraft initial conditions were offset from the initial glideslope. The initial conditions that were perturbed were the altitude and velocity where the nominal conditions were 2295.5 m and 180 m/s respectively. Figure 20 shows the simulation of the aircraft with perturbations of the altitude to 2754.8 m and the velocity to 200 m/s. Figure 21 shows the simulation of the aircraft with a large perturbation of the altitude to 3213.7 m. In these cases the energy management control system was able to successfully land the aircraft. Figure 22 shows the success of the energy management control system in landing the aircraft given perturbations of the altitude and velocity of the aircraft for both strategies. The second strategy was more robust to variations in the initial altitude and velocity, only failing at the high altitude (3213.7 m) and high velocity (240 m/s) conditions.

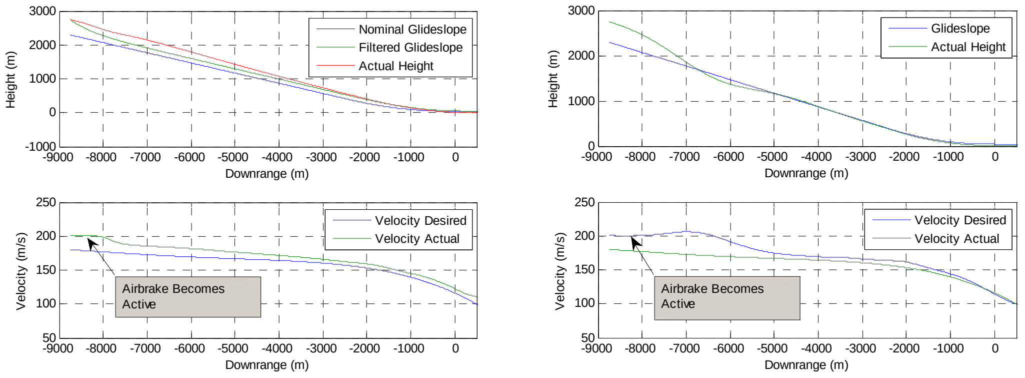


Figure 20. (Strategy 1 left, Strategy 2 right) V0 = 200 m/s, initial altitude 2754.6 m.

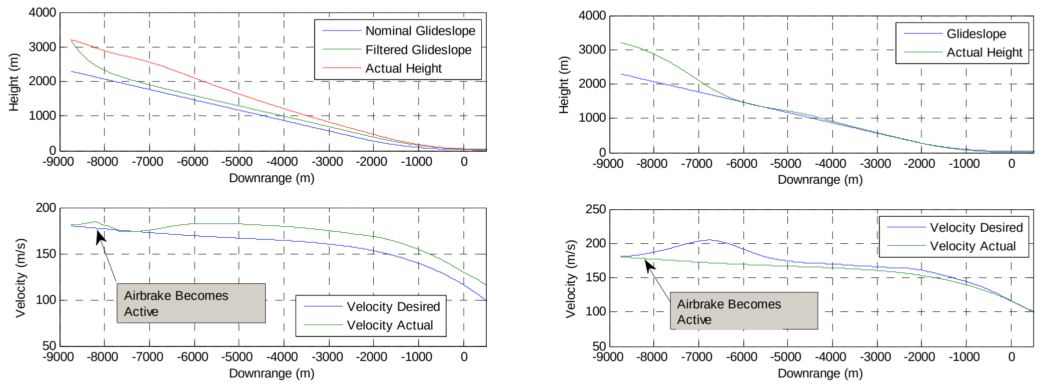


Figure 21. (Strategy 1 left, Strategy 2 right) V0 = 180 m/s, initial altitude 3213.7 m.

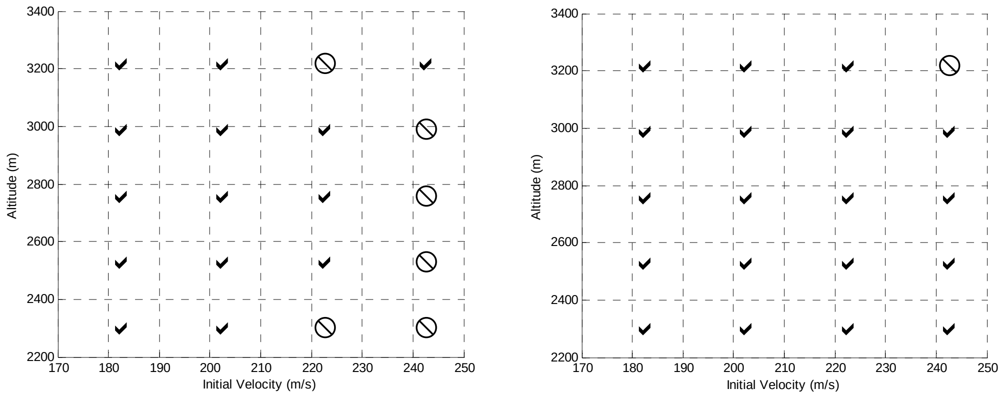


Figure 22. (Strategy 1 left, Strategy 2 right) Landing success for different initial conditions.

X. Next Steps

The tests that have been written provided for the basics of measuring vehicle performance. In the method that they operate today, the smoke and hit runway tests require the model to run a complete simulation. The measure envelope test requires twenty runs of the model. Even when the model runs in less than a minute the total simulation time for this test is nontrivial. Each of the simulation runs is independent and could be distributed amongst a computing cluster. This change to the test harness should be relatively easy, and when complete will save time when running tests and may also enable more detailed measurements of the bounds over which the model will land successfully.

By adopting a systematic strategy to use a configuration management system we are now in a position to more productively continue improving the design of the control system for the aircraft. This system gives us the flexibility to experiment with other control design concepts to improve the guidance control system to expand upon its operation. Our next step is to add the Pre-flare and Outer Glideslope components.

XI. Conclusion

Throughout this project we found that configuration management can be successfully applied to the Model-Based Design process using MATLAB and Simulink. Working in a single tool chain allowed us to utilize a single model as the center for multidisciplinary modeling, control design, and system verification. We were able to set up a system using a single-committer model, in which we found that we could quickly get revisions for experimenting with new control design strategies. This allowed us to also recover a working configuration of a sandbox when we wanted to start fresh during an exploratory phase. For a large team or organization that has adopted Model-Based Design, careful implementation of configuration management will also help minimize errors introduced due to miscommunication between teams. The successful CM strategy involves a workflow using testing as well as predefined styles and naming conventions. We found that testing is crucial to provide immediate confirmation that design changes are valid, and the flexibility of MATLAB allows for different levels of automation to be incorporated into the configuration management workflow.

Acknowledgments

The authors would like to thank E. B. Jackson and C. L. Cruz for publishing the HL-20 Aerodynamic data used in the Simulink model presented in this paper.

References

[1] Gavin Walker, Jonathan Friedman, and Rob Aberg, "Configuration Management of the Model-Based Design Process", SAE 2007-01-1775

[2]Jackson E. B., Cruz C. L., "Preliminary Subsonic Aerodynamic Model for Simulation Studies of the HL-20 Lifting Body", NASA TM4302, August 1992.

[3] Jackson, E. Bruce; Cruz, Christopher I.; Ragsdale, W. A., "Real-time simulation model of the HL-20 lifting body", NASA TM-107580, July 1992.

[4] Glass, J. "Tuning Multi-Loop Compensators to Meet Time and Frequency Domain Requirements", MathWorks Aerospace and Defense Conference June 14th, 2006, Reston VA.

[5] Glass, J, Shenoy, R. "Multi-Loop Control Design in Simulink® — Made Easy", MathWorks Webinar, Recorded August 29th, 2006.

[6] Barnard, P. "Software Development Principles Applied to Graphical Model Development", AIAA-2005-5888, August, 2005.

[7] Potter, Bill. "Model-Based Design for DO-178B", MathWorks Webinar, Recorded July 21st, 2006.

Computer Software

[8]MATLAB, Software Package, Ver. 7.5, The MathWorks, Inc, Natick, MA, 2007.

[9]Simulink, Software Package, Ver. 7.0, The MathWorks, Inc, Natick, MA, 2007.

[10]Real-Time Workshop, Software Package, Ver. 7.0, The MathWorks, Inc, Natick, MA, 2007.

[11]Control System Toolbox, Software Package, Ver 8.0.1, The MathWorks, Inc, Natick, MA 2007.

[12]Simulink Control Design, Software Package, Ver. 2.2, The MathWorks, Inc, Natick, MA, 2007.

[13]Aerospace Blockset, Software Package, Ver. 3.0, The MathWorks, Inc, Natick, MA, 2007.

[14]FlightGear, Software Package, Ver. 0.910, www.flightgear.org, 2007.

[15]TortoiseCVS, Software Package, Ver. 1.9.14, www.tortoisecvs.org, 2007

Appendix A

Test file: tsmokeBasic.m

```
function out = tsmokeBasic(varargin)
% SmokeBasic test for HL20_Top, optional input, sys

sllasterror([])
lasterror('reset')

if nargin==0
    sys = 'HL20_Top';
```

```
else
    sys= varargin{1};
end

out = testReportInit(mfilename);
out.model = sys;

try
    % remove slprj and sfprj directories to get a clean build
    if exist('slprj','dir')
        [stat,msg,mid] = rmdir('slprj','s');
        if ~stat
            error(mid,msg)
        end
    end

    if exist('sfprj','dir')
        [stat,msg,mid] = rmdir('sfprj','s');
        if ~stat
            error(mid,msg)
        end
    end

    open_system(sys)

    % Turn off the simulation pace block
    set_param([sys '/toFlightGear/Simulation Pace'],'SleepMode','Off')
    sim(sys)

    close_system(sys,0) % close the model without saving

    % set the status if it passed
    out.status = 'Passed';

    % capture data from the test
    out.data = [];

catch
    out.status = 'Failed';

end

out.stopTime = now;
out.sllasterror = sllasterror;
out.mllasterror = lasterror;
```

Test file: tHitRunway.m

```
function out = tHitRunway(varargin)
% HitRunway test for HL20_Top, optional input, sys

sllasterror([])
lasterror('reset')

if nargin==0
    sys = 'HL20_Top';
```

```
else
    sys= varargin{1};
end

out = testReportInit(mfilename);
out.model = sys;

try
%%
    open_system(sys)

    % Add signal logging to the airframe data bus
    ph = get_param([sys '/HL-20 Vehicle Systems Model'],'PortHandles');
    set_param(ph.Outport(2),'DataLogging','on')
    set_param(ph.Outport(2),'DataLoggingNameMode','custom')
    set_param(ph.Outport(2),'DataLoggingName','airframedata')
    set_param(ph.Outport(2),'DataLoggingDecimateData','off')
    set_param(ph.Outport(2),'DataLoggingLimitDataPoints','off')

    % Turn off the simulation pace block
    set_param([sys '/toFlightGear/Simulation Pace'],'SleepMode','Off')

    % Save states and signal logging to the workspace
    set_param(sys,'SaveState','on')
    set_param(sys,'StateSaveName','xbase')

    set_param(sys,'SignalLogging','on')
    set_param(sys,'SignalLoggingName','sigsOut')
    % Simulate the complete model
    sim(sys)

    % Examine the position at the first instance where there was weight on
    % the wheels of the vehicle.

    % Find the index to the time at which the vehicle touched down
    iTouchDown = ...
      find(any(sigsOut.airframedata.PlantData.LGData.WOW.Data,2),1,'first');
    disp(['Touch Down time at ' ...
      num2str(sigsOut.airframedata.PlantData.Xe.Time(iTouchDown)),'s'])
    % Get the position
    lon_rad = ...
      sigsOut.airframedata.PlantData.PosLLA.long_rad.Data(iTouchDown);
    lat_rad = sigsOut.airframedata.PlantData.PosLLA.lat_rad.Data(iTouchDown);
    alt_m = sigsOut.airframedata.PlantData.PosLLA.alt_m.Data(iTouchDown);
    % Is the position on the runway?
    lon_deg = convang(lon_rad,'rad','deg');
    lat_deg = convang(lat_rad,'rad','deg');

%% get runway boundaries
    [rwx,rwy,tx,ty,nc,sc] = getRunwayDataSFO;

    % If the lon, lat point is within the runway, pass.
    if inpolygon(lon_deg,lat_deg,tx,ty)
        out.status = 'Passed';
    else
        out.status = 'Failed';
    end
```

```
%%

    % capture data from the test
    out.data = sigsOut;


catch
    out.status = 'Failed';

end

out.stopTime = now;
out.sllasterror = sllasterror;
out.mllasterror = lasterror;



%%
function [rwlon,rwlat,tlon,tlat,nc,sc] = getRunwayDataSFO
% getRunwayDataSFO - returns runway data for SanFrancisco Airport
% [rwlon,rwlat,tlon,tlat,nc,sc] = getRunwayDataSFO
% rwlon is the vector of longitude points for the runway
% rwlat is the vector of latitude points for the runway
% tlon is the vector of longitude points for the target (1st half of runway)
% tlat is the vector of latitude points for the target (1st half of runway)
% nc is northmost center of the runway
% sc is the southmost center of the runway
%
% This can be used to make a patch from the points:
% Example:
% patch(rwlon,rwlat,'k')
% hold on
% patch(tlon,tlat,'r')
% plot([sc(1),nc(1)],[sc(2), nc(2)],'b*-',rwlat,rwlon,'k')
% text(sc(1),sc(2),'28R')
% text(nc(1),nc(2),'10L')
% axis equal


%% build runway boundries

% Sources:
% http://en.wikipedia.org/wiki/San_Francisco_International_Airport
aplat = 37.618889;
aplon = -122.375;
% Centers of ends:
% http://earth.google.com/
nc = [-122.393719, 37.628874]; %     north center
sc = [-122.357194, 37.613553]; %     south center

rwlen = 11870; %feet
rwwid = 190;   %feet

ratwid2len = rwwid/rwlen; % ratio of length to width
vlen = nc - sc;  % runway length vector
vwid = [-vlen(2), vlen(1)] * ratwid2len; %width vector (orthogonal to length)
% bounds of runway
%           28R
%       nl  nc   nr   nc = 122.393719 W, 37.628874 N
%        *---*---*
%        |  /|\  |
%        |   |   |
%        |   |   |
%        |   |   |
%     ml *   |   * mr  mid
%        |   |   |
%        |   |   |
%        |   |vlen
% vwid  <-------|
%        *---*---*
%       sl  sc   sr    sc = 122.357194 W, 37.613553 N
%           10L

sl = sc + .5 * vwid;
sr = sc - .5 * vwid;
nl = sc + vlen + .5 * vwid;
nr = sc + vlen - .5 * vwid;
ml = nc - .5 * vlen + .5 * vwid;
mr = nc - .5 * vlen - .5 * vwid;

% Runway polygon
corners = [nl; nr; sr; sl; nl];
rwlon = corners(:,1);
rwlat = corners(:,2);

% target polygon
target = [ml; mr; nr; nl; ml];
tlon = target(:,1);
tlat = target(:,2);
```

Test file: tMeasureEnvelope.m

```
function out = tMeasureEnvelope(varargin)
% test to measure the successfule initial operating conditions
% of HL20_Top, optional input, sys

% Keep the some variables in sync with the base workspace (used by the model)
global Vinit Vmw height_init height0
evalin('base','global Vinit Vmw height_init height0')

sllasterror([])
lasterror('reset')

if nargin==0
    sys = 'HL20_Top';
else
    sys= varargin{1};
end
out = testReportInit(mfilename);
out.model = sys;


try
```

```
    % make sure the model is closed, because it needs to initialize it's
    % workspace
    bdclose(sys)
    open_system(sys)

    failed = zeros(0,2);
    passed = zeros(0,2);

    Vinit = [180;200;220;240];
    height_init = [1;1.1;1.2;1.3;1.4]*height0;

    for ct1 = numel(Vinit):-1:1
        for ct2 = numel(height_init):-1:1
            % Set the initial velocity
            Vmw(1) = Vinit(ct1);
            height0 = height_init(ct2);
            try
                teststruct = tHitRunway('HL20_Top');
                if strcmp(teststruct.status,'Passed')
                    passed(end+1,:) = [Vinit(ct1),height_init(ct2)];
                else
                    failed(end+1,:) = [Vinit(ct1),height_init(ct2)];
                end
            catch
                failed(end+1,:) = [Vinit(ct1),height_init(ct2)];
            end
        end
    end

    figure
    plot(passed(:,1),passed(:,2),'*')
    hold on;plot(failed(:,1),failed(:,2),'o')
    legend('Successful Landing','Failed Landing')
    xlabel('Initial Velocity (m/s)')
    ylabel('Altitude (m)')


    % set the status if it passed
    out.status = 'Passed';

    % capture data from the test
    out.data.passed = passed;
    out.data.failed = failed;

catch
    out.status = 'Failed';

end
clear global Vinit Vmw height_init height0
out.stopTime = now;
out.sllasterror = sllasterror;
out.mllasterror = lasterror;
```