Design Variant Management in Model-Based Design

Priyanka Gotika[1]
Saurabh Mahapatra[2]

MathWorks Inc, Natick, MA, 01760

Modular design platforms require the customization of a single design to meet diverse customer requirements dictated by considerations such as application, cost, and operational considerations. Many of these dynamic changes in nature have required design component variations on top of a fixed master design. The concept of modularity applied intelligently to meet such needs has proved to be a cost-effective and efficient paradigm to meet these challenges. In the paper[1], we introduced variant semantics and their usage within a graphical modeling environment such as Simulink. Also, we introduce a scripting methodology for efficiently mapping a custom design to a permutation of variants and their subsequent abstraction for ease of understanding. However, this approach required ad hoc management of variants and the associated. For casual users, this increases the risk of introducing errors into the development workflows. In this paper, we introduce new concepts that addresses those challenges, namely, an intuitive graphical user interface, variant configuration data objects, data dictionary and project based environment for collaboration. Simulink examples are provided are proposed to illustrate these concepts.

I. Introduction

In various organizations, the reuse of software and hardware components continues to be a central design theme[2]. With ever increasing product complexity, high specificity of customer requirements and cost pressures make it imperative for engineers to take a design reuse-centric point of view. It is not surprising to observe that modular product architectures and production have long been used in the civil aircraft industry. The variants in Boeing's 747 product family show the same reuse pattern with very similar design parameters but serving different requirements[3]. The 747-200B was a pioneer in low-cost air travel for the masses. The 747-200F was a freighter version with an upward hinging nose. The 747 SP was an extra-long-range-variant featuring a taller tail and short fuselage. The E-4B Command Post variant was equipped to become the wartime emergency base for the US President and his advisors.

Variants present a variety of uses in the context of Model-Based Design[4, 5] workflows. They enable the creation of modular design platforms facilitating reuse and customization. Design exploration where several alternatives exist for a component can now be managed efficiently to simulate every design possibility in a combinatorial fashion for a given test suite. For large-scale problems, these could be distributed on a cluster of multicore computers for overall speedup with our scripting methodology. Alternatively, different test suites could also be mapped for efficiently managing relevant tests for a design. Maintenance activities of existing aircraft may require the upgrade of several components with no deterioration in existing performance requiring the testing of these upgrades in the model. Design elaboration and integration is a challenging activity where low fidelity components are replaced by more specialized ones. Since the order in which these components are integrated influence design quality and subsequent iterations, it is possible to carry out several separate integrations that increase confidence. Based on the evaluation criteria, a subset of these designs could be shortlisted for rapid prototyping or hardware-in-the-loop testing. With automatic code generation, variant components in the software model are mapped to C function code variants that can be switched by simply modifying the preprocessor definitions. Conversely, if there be hardware variants such as floating or fixed-point microprocessors, they will require the use of variants upstream with different modeling implementations.

In our previous paper[1], we outlined strategies on how organizations can leverage these possibilities to reuse while enhancing their existing knowledge to meet the design challenges of the future. The cornerstone of our approach was a 2-tier variant framework that used a scripting methodology. However, a key limitation of this approach was the

[1] Product Marketing Manager, Simulink Platform, 3 Apple Hill Drive, Natick, MA, and AIAA member. .
[2] Senior Product Manager, Simulink Platform, 3 Apple Hill Drive, Natick, MA, and AIAA member.

management of control variables and the associated variant objects. Since this variant metadata is present in the MATLAB workspace which is a central repository for data across all models, there is a risk of unintentional data tampering. To alleviate this, our recommendation was to encapsulate this information within a MATLAB script with suitable checks. Since the script had access to the MATLAB workspace, it can be used as a utility script within the development environment. We also recommended a better encapsulation strategy which was to create a MATLAB class where the variant-related information would be declared as private members and the appropriate method used to activate the desired variants. However, such an approach is also fraught with readability issues at the model level because an object's members would have to be accessed using the dot notation. A compact naming scheme may have alleviated the issue but it does not scale well as the number of variants increase. The root cause of these issues was the lack of a variant-related data namespace which would allow for logical partitioning and separation of variant metadata.

Another limitation of our approach was the definition of the sets of control variables that would be used to activate a configuration in a model. This would pose understandability issues and introduce errors as the definitions would reside in the MATLAB scripts. Encapsulating and managing of these sets of control variable data is essential to effective variant configuration management. Furthermore, constraints would also need to be imposed to ensure that invalid configurations will not be allowed. Although this functionality can be implemented in a MATLAB script, it still be burdensome for the casual and intermediate users.

In a team-based environment, it may be necessary to architect a model into a component file hierarchy with their associated data. However, the placement of variant metadata in the MATLAB workspace will increase the risk of corruption by a team member. Thus the needs of collaboration would need to be balanced with the need of isolation with respect to variant metadata. Again, MATLAB scripts can be used for separating variant definitions but they still do not reduce the risk of data corruption as these would need to be loaded to the MATLAB workspace.

In this paper, our goal is to address the limitations posed by the scripting methodology by introducing an intuitive graphical user interface. We modify the variant handling framework with the definition of variant configurations. We also introduce the concept of a Simulink data dictionary for managing variant metadata and a team-based environment for collaboration called Simulink Projects. It is our belief that these concepts taken together will improve upon the ideas that we presented earlier.

II. Variant Framework in Simulink

In this section, we reproduce a modified introduction to the variant handling framework in Simulink®[6] as presented in our previous paper[1]. The understanding of this framework will lay the foundation for the variant management section of this paper. Interested readers are encouraged to read our paper[1] covering about other aspects of variant management not covered here. A modular design platform[1] is a finite set of components and their associated interfaces that can form a fixed common structure while allowing for some variability. The example in Figure 1 shows a two-dimensional user-specified market segmentation model containing twelve segments onto which are mapped two component variants each containing two choices. Since only four variant configurations are possible, the redundancy across various segments can potentially result in higher economies of scale thereby reducing both sunk and incremental costs. For the sake of demonstration, it is assumed that each segment represented by *(S1, S2)* has only one variant configuration associated with it. This is based on the assumption that sound segmentation schemes, possibly higher dimensional will yield this mapping.

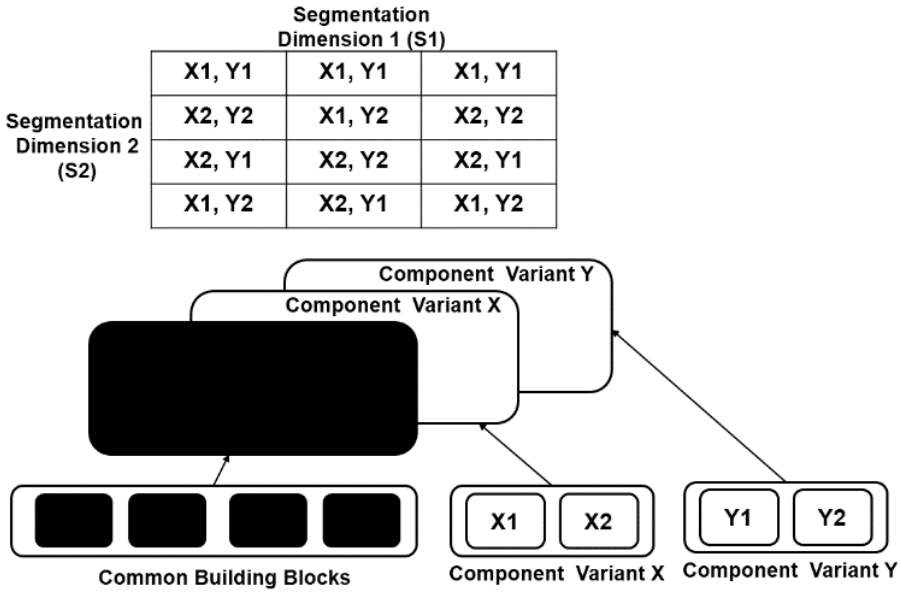

Figure 1: Redundant variant configurations created from variant choices mapped to market segments.

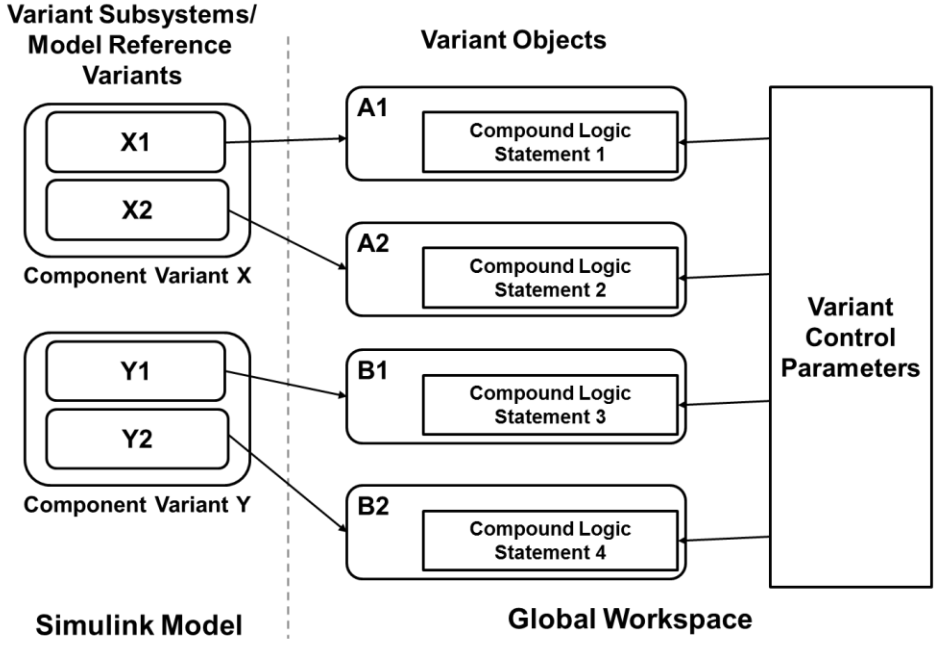

Figure 2: Design variants in a Simulink model are associated with variant objects that encapsulate atomic Boolean statements based on control variables defined in a global workspace.

In Simulink, variable components can be represented either as variant subsystems or model reference variants. Variant subsystems allow variant configurations to be incorporated within a single model file whereas model reference variants allows for implementation in separate files. A schematic of the implementation is shown in Figure 2. The variant choices can have a many-to-one mapping onto a variant object which encapsulates a compound logical statement based on the control variables. This mapping is defined at the model-level and is independent of the placement of the variant in the hierarchy. The encapsulation is atomic in nature and only allows the association of a Boolean logical statement with a variant object. The variant object and the corresponding component in the Simulink model are activated at compile time prior to simulation when the encapsulated Boolean statement evaluates to TRUE.

An implementation of the design variants in Simulink is shown in Figure 3 that can be used to represent the four design derivatives shown in Figure 1 which are mapped onto 12 segments represented by *(S1, S2)* market segment coordinates. There are four variant objects named *X1Y1*, *X1Y2*, *X2Y1*, and *X2Y2*. Each of the variant objects is mapped to its corresponding files by instantiating them in the global workspace. Within the tool, a global workspace can either be a MATLAB workspace or a Simulink data dictionary[6]. The variant objects are shared across the two model reference variants. For example, X1Y1 is mapped to the file *choice_X1* file in component variant X and is also mapped to the file *choice_Y1* file in component variant Y. Observe that the control variables are the market segment coordinates *(S1, S2)* are mapped to the variant objects by compound logical statement construction. By using Boolean algebraic theorems, simplifications can be carried that result in compact representation and better readability. However, such flexibility does come up with the associated risks of creating unsound compound logical statements that may erroneously activate multiple variant choices within a single variant or choose the wrong variant. Checks may need to be incorporated to ensure that the control variables are mapped correctly to the logical statements. We address these concerns in the next section.

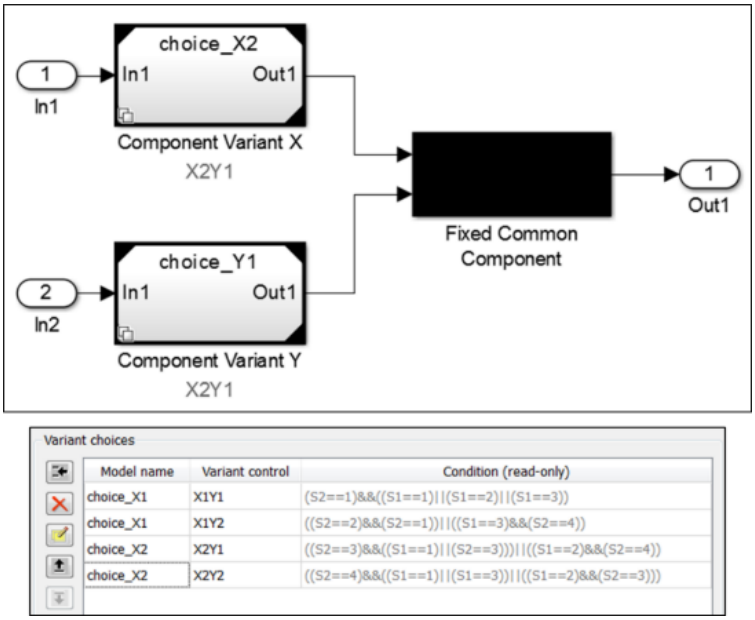

Figure 3: Implementation of the variant configurations outlined in Figure 1 using model reference variants in Simulink.

III. Managing Variants in Simulink

In this section, we cover some of the salient features of a variant management implementation using Simulink as an exemplary environment.

A. Intuitive Graphical User Interface

Despite the advantages of automation and scalability of scripting methodology[1], the approach does present adoption challenges for non-expert users. Thus, an intuitive user interface can be an enabling factor that drives widespread adoption within an organization. In the following example, the two approaches are shown comparatively.

In the example shown in the Figure 4, the *Plant* subsystem has 2 variants, called *Piston* engine variant and the *Turboprop* engine variant. The *Controller* subsystem has 3 variants, namely *STOL* (Short Takeoff and Landing), *VTOL* (Vertical Takeoff and Landing), and *NTOL* (Normal Takeoff and Landing). Further, the *Environment* model consists of the *Steady State* and *Turbulence* variants. The *Pilot* subsystem also has two variants, *Beginner* and *Expert.* In Figure 5, we define the valid configurations by mapping customer requirements from fictitious customers – Company A, Company B and Company C. From the standpoint of serving the customer, there are two enumerated control variables *tol* and *engine*. The specific combinations of these variables are mapped to the customer needs. For Example, Company A has requested a *VTOL* aircraft with a turboprop engine. Company B has requested a *STOL* aircraft with a *Piston* engine. While Company C has requested a *NTOL* aircraft regardless of what the engine type is. These requirements can be translated into Boolean logic statements by defining 2 enumerated control variables that can take

3 and 2 values each i.e., *tol = (vtol, stol, ntol)* and *engine = (piston, turboprop)*. From the view point of internal testing and ensuring design robustness, 2 environment conditions would need to be permuted with 2 pilot behaviors giving rise to 4 test cases for each customer requested aircraft configuration. Again, this can be handled through 2 enumerated control variables, *env* and *pilot* i.e *env = (steadystate, turbulence)* and *pilot = (beginner, expert)*. Although, a total of 24 variant configurations are possible for 3 *Controller* variants and 2 *Plant* variants to be tested against 2 *Environment* variants and 2 *Pilot* variants, only 16 are valid based on customer requests as shown in Figure 6.

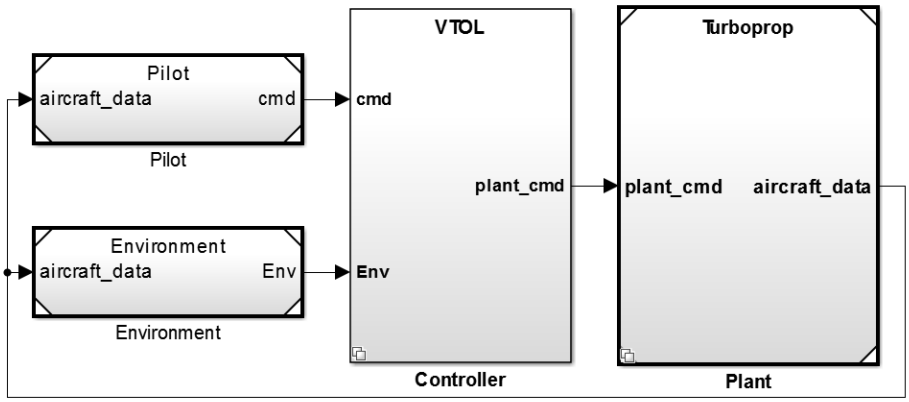

Figure 4: System level model *Aircraft_Variants* showing the configuration requested by Company A

Controller Type / Engine Type	VTOL	STOL	NTOL
Piston		Company B	Company C
Turboprop	Company A		Company C

Figure 5: Customer requested configurations

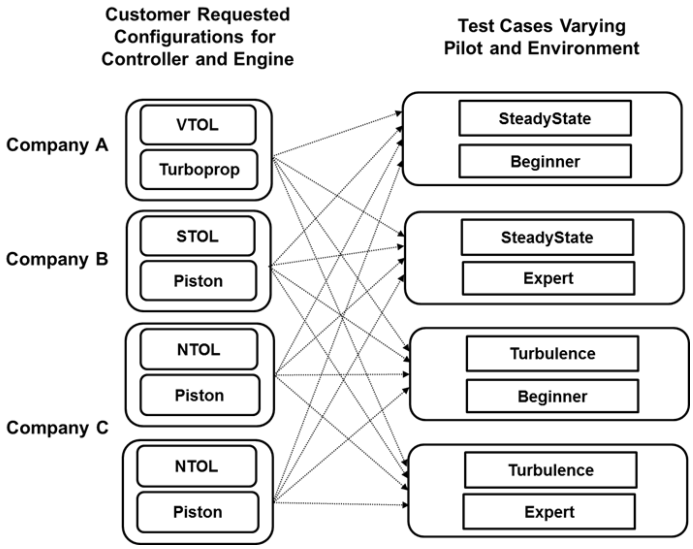

Figure 6: 16 different variant configurations

Using the MATLAB scripting methodology[1], we create a variant object for each variant. Associated with each variant object is a Boolean expression. This is shown in Figure 7. To activate a valid configuration, we create a single variant configuration data object with 16 variant configurations. i.e. 4 customer requested variant configurations

((VTOL, Turboprop), (STOL, Piston), (NTOL, Piston) and (NTOL, Turboprop)) with 4 test cases each as shown in Figure. Clearly, the scripting approach offers the flexibility of incorporating additional code that can be used as a trigger or a decision point for activating different configurations. Additionally, the scripting approach is also attractive for dealing with a large number of configurations. However, there are several issues associated with this approach. There is no top-level information available about the variants and the associated subsystems which can lead to errors.

Variant Object	Condition
NTOL	tol == controller_type.ntol
STOL	tol == controller_type.stol
VTOL	tol == controller_type.vtol
SteadyState	env == environment_type.steadystate
Turbulence	env == environment_type.turbulence
Beginner	pilot == pilot_type.beginner
Expert	pilot == pilot_type.expert
Piston	engine == engine_type.piston
Turboprop	engine == engine_type.turboprop

Figure 7: Variant object definitions

The graphical user interface approach takes a different view of this problem by enforcing the following 4 steps as shown in Figure 8:

1. Visualize, explore and set variant controls: Panel *A* can be used to visualize and explore all variants in a central location. Variant objects and associated conditions can be defined in this panel. These variant objects are saved to MATLAB base workspace instantly as they get created
2. Set configurations**:** Panel *B* shows a variant configuration data object *aircraft_configurations* that shows 16 different configurations as a result of 4 test cases for the 4 customer requested aircraft configurations. Each configuration is setup by assigning appropriate values to the control values in panel *C* such that they satisfy the conditions associated with the variant objects for the variant subsystems and model variants.
3. Validate and set default active variants: Each of these configurations can then be validated by the click of a button and the errors would appear in panel *D*.
4. Export and save: As a last step, the variant configuration data object should be exported and saved to file for future use.

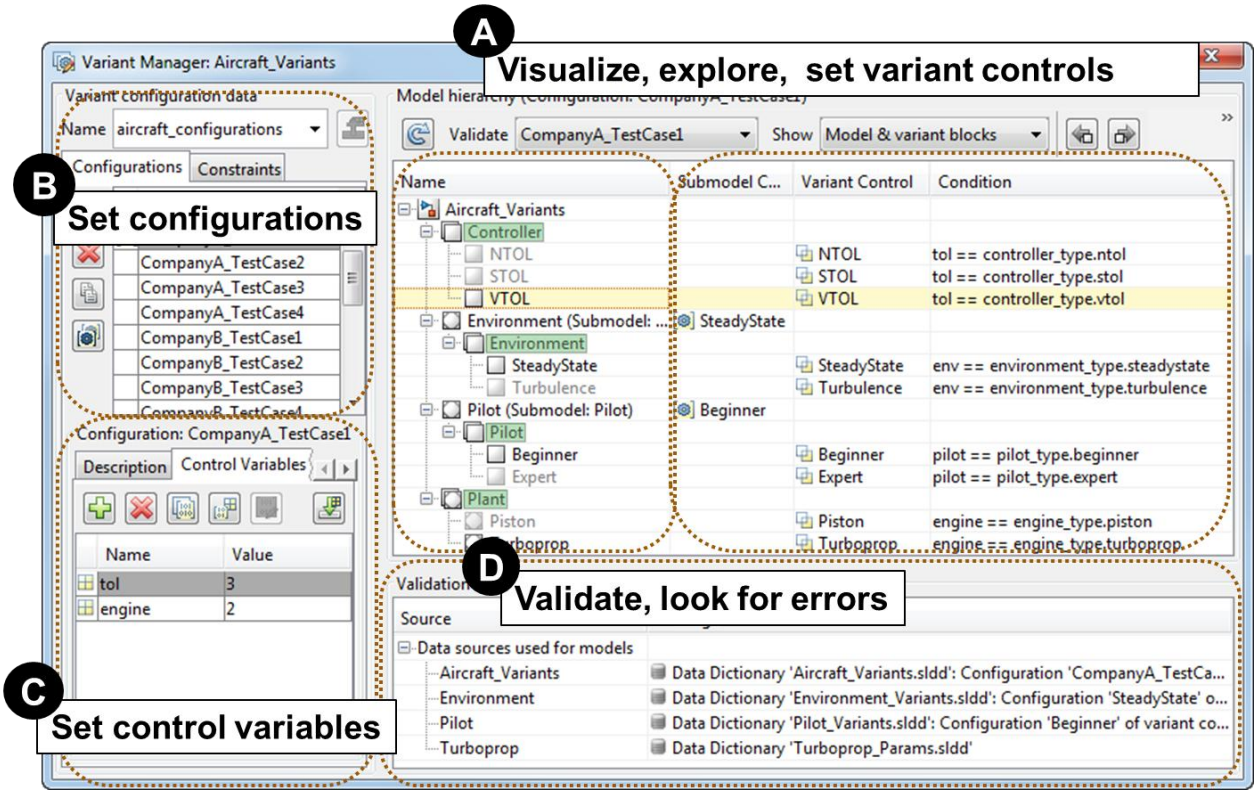

Figure 8: Variant Manager user interface

B. Managing Constraints

It may be undesirable to have certain configurations enter a development or testing phase. For this reason, it is required to impose constraints on these configurations by defining them within the variant configuration data object itself. A constraint can be defined by a name and an associated Boolean logical statement.

For example, an aircraft configuration that has *STOL* controller with a *Turboprop* engine is not requested by any of the customers and so it is unnecessary to test this configuration against the *Environment* and *Pilot* variants. A constraint *Restrict_STOL_Turboprop* is created with an associated condition "`~((tol==2)&&(engine==2))`". This means that whenever the variant configuration is set up such that this condition is not satisfied, an error appears in panel *D* upon validation as shown in Figure 9. Constraints can be placed with a variant configuration object using MATLAB scripts as well. Similarly, no customer wants a VTOL aircraft with a piston engine, so an additional constraint *Restrict_VTOL_Piston* with an associated condition, "`~((tol==3)&&(engine==1))`"can be added to address this scenario.

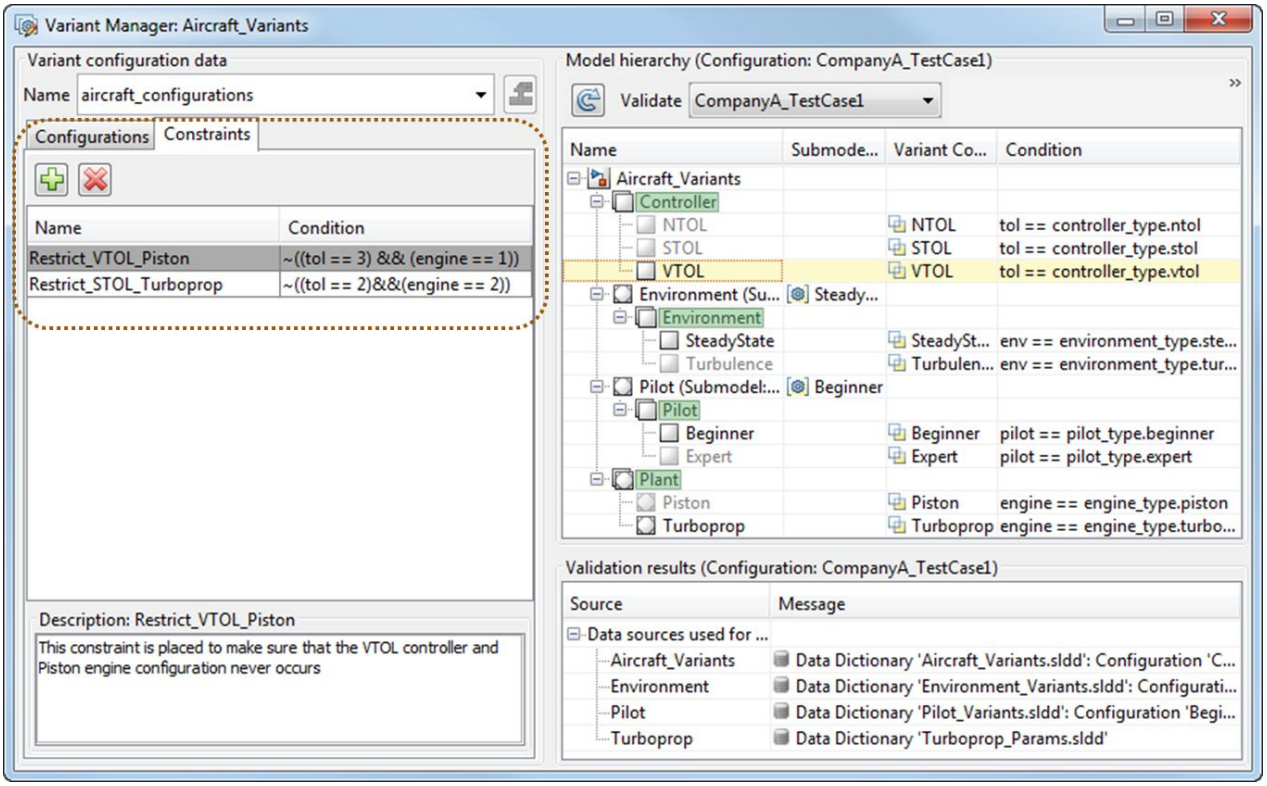

Figure 9: Define constraints using Simulink Variant Manager user interface

C. Enabling Team Collaboration Workflows

Figure 10 shows the example of a team collaboration environment where the engineers would like to work independently and parallel with each other and at the same time integrated into the system level model. For this scenario, we would need to evolve a mechanism for managing the variant objects and the variant data. Furthermore, we would also require that the team is able to work collaboratively. First, let us understand the ownership of the various components in this team.

- The *Environment* component has two variants, *SteadyState* and *Turbulence* that will be worked on by team member *John*. *Tom* keeps the ownership of the component *Pilot* which has two variants *Beginner* and *Expert.* Since *John* and *Tom* would like to work independently, their respective components are referenced models and hence separate files.
- The *Controller* component is owned by *Lisa* who is a system level engineer and who would need to integrate all the components. Since *Lisa* is the owner of the system level model, she does not need to maintain the *Controller* variants, *NTOL*, *STOL* and *VTOL* as separate files.
- The *Plant* component contains two variants *Piston* and *Turboprop* developed by *Rob* and *Amy* respectively. Since they would like to work independently of each other, the variants themselves are referenced models and hence separate files.

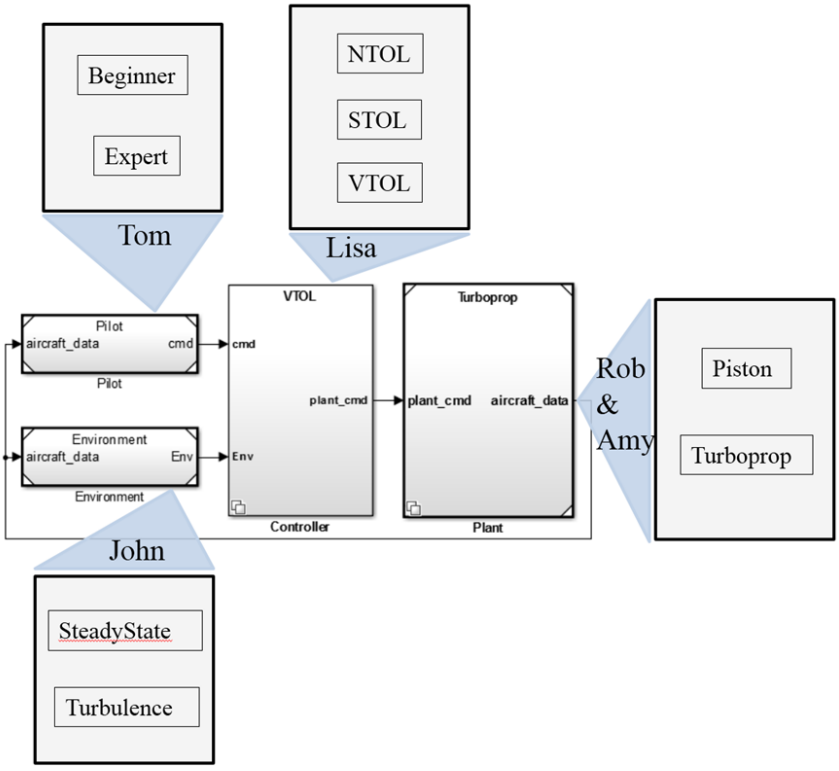

Figure 10: Ownership of the variants in the system level model

A Simulink data dictionary[7] is a persistent repository of global design data that a model uses. The dictionary only stores design data, which define parameters and signals, and include data that define the behavior of the model. The dictionary does not store simulation data, that is, any inputs or outputs of model simulation. It can also store variant objects, variant configuration objects and definitions of the control variable associated with each configuration. A Simulink data dictionary can only be linked to a model. However, in a model reference hierarchy, it is possible to associate a separate data dictionary for each component. Thus, design data separation is possible at the component level. However, the parent model requires that the data dictionaries used by all its components be referenced by its own dictionary. It is not necessary that the data dictionary hierarchy correspond exactly to the model reference hierarchy. On the disk, the Simulink data dictionary exists as a file with a *.sldd* extension. Being a file, it offers several advantages such as access to the data without a network connection, working within a configuration management system.

For this team, we would like to create a data dictionary hierarchy as shown in Figure 11. The system level data dictionary, *Aircraft_Variants.sldd* contains system level parameters required for simulating the system level model. Component data dictionaries are partitioned into variant data dictionaries and design parameter data dictionaries. The references are made such that the top level model should reference the variant data dictionaries which will in turn reference design parameter data dictionaries. The variant data dictionaries contains variant objects, control variables and variant configuration data objects for respective variant systems (variant subsystems and model reference variants). Similarly, the design parameter data dictionaries contain design parameter definitions. This approach gives rise to four variant data dictionaries: *Pilot_Variants.sldd*, *Controller_Variants.sldd*, *Plant_Variants.sldd*, *Environment_Variants.sldd* and nine design parameter data dictionaries: *Beginner_Params.sldd*, *Expert_Params.sldd*, *SteadyState_Params.sldd*, *Turbulence_Params.sldd*, *NTOL _Params.sldd*, *STOL_Params.sldd*, *VTOL_Params.sldd*, *Piston_Params.sldd* and *Turboprop_Params.sldd*. Although *Controller* component is modeled as a variant subsystem and does not require file separation in the form of a model reference, a separate data dictionary file, *Controller_Variants.sldd* is created for the sake of logical partitioning. Each of these 4 variant data dictionaries reference the design parameter data dictionaries. For example, *Pilot_Variants.sldd* references *Beginner_Params.sldd* and *Expert_Params.sldd* where *Beginner* and *Expert* are the variants of the *Pilot* component. Furthermore, for each of these variant components there may be parameters that may be shared between the variants. For this reason, another

data dictionary is created which contain shared parameters. For example, *Pilot_Shared_Params.sldd* is referenced both by *Beginner_Params.sldd* and *Expert_Params.sldd* and it may contain parameter data that may be shared by both the variants. There are four such shared parameter dictionaries for the project.

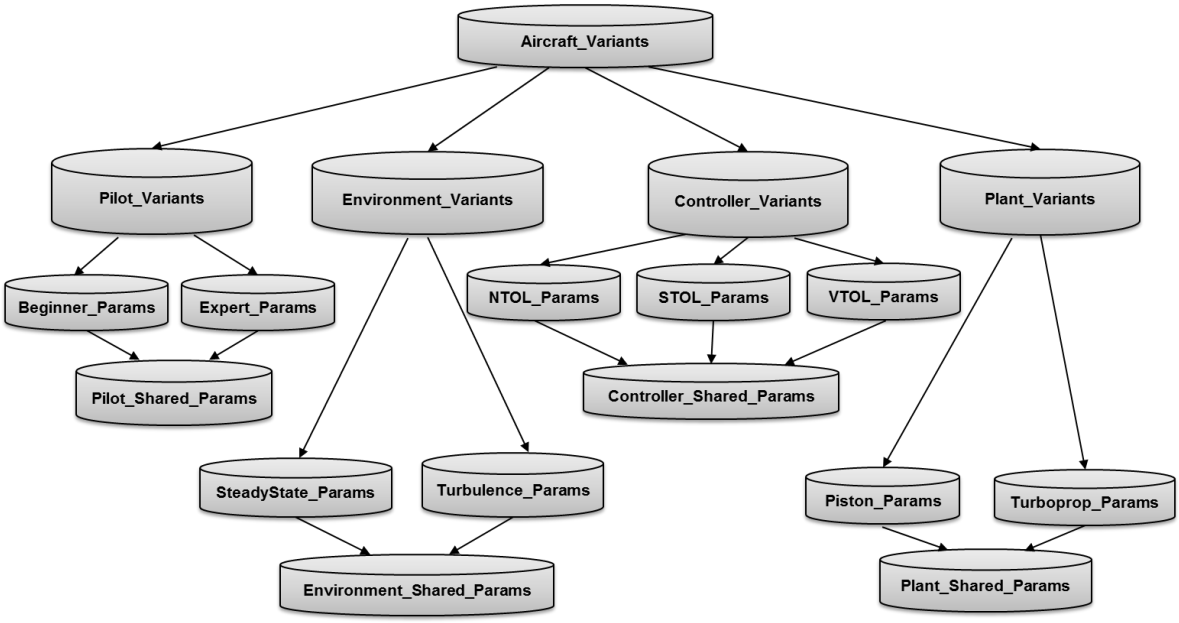

Figure 11: Data dictionary hierarchy

Model reference hierarchy for this example is shown in Figure 12. Figure 13 shows how each of the data dictionaries are linked to model references in the context of model reference hierarchy. Each cell shows how each of the model references of the system level model, *Aircraft_Variants* are linked to the corresponding data dictionaries. *Aircraft_Variants* model itself is linked to the system level data dictionary *Aircraft_Variants.sldd* which has design parameter data required at system level. In addition, it references all 4 variant data dictionaries to access the required component level data for the purpose of simulation. *Lisa*, who is the system level engineer is responsible for ensuring this referencing so that all the required data is available to simulate the system level model.

Notice that the data dictionary hierarchy does not necessarily correspond to the model reference hierarchy. As you can see, *Pilot* model is linked to *Pilot_Variants.sldd* and can be developed and simulated independently by *Tom*. Similarly *Environment* model is linked to *Environment_Variants.sldd* and *John* can independently work on this model in parallel with *Tom.* Notice that for *Plant* model variants, *Piston* and *Turboprop* are linked to their respective design parameter data dictionaries *Piston_Params.sldd* and *Turboprop_Params.sldd* instead of variant data dictionaries. This is because the *Piston* and *Turboprop* models which are model variants of the *Plant* model do not require variant information in order for them to be simulated or developed independently. Hence, by linking the respective design parameter data dictionaries *Rob* and *Amy* are able to work independently on their models. By partitioning the data dictionaries and models this way team collaboration can be enabled in a way where all team members can work independently and in parallel.

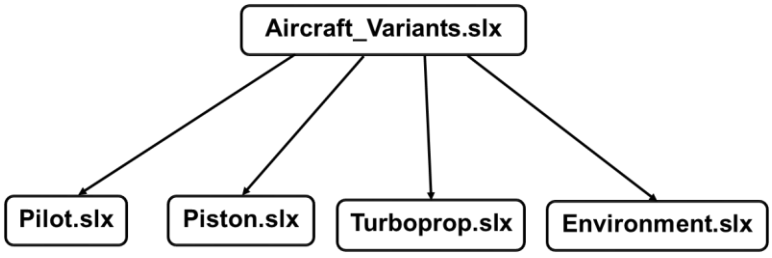

Figure 12: Model reference hierarchy

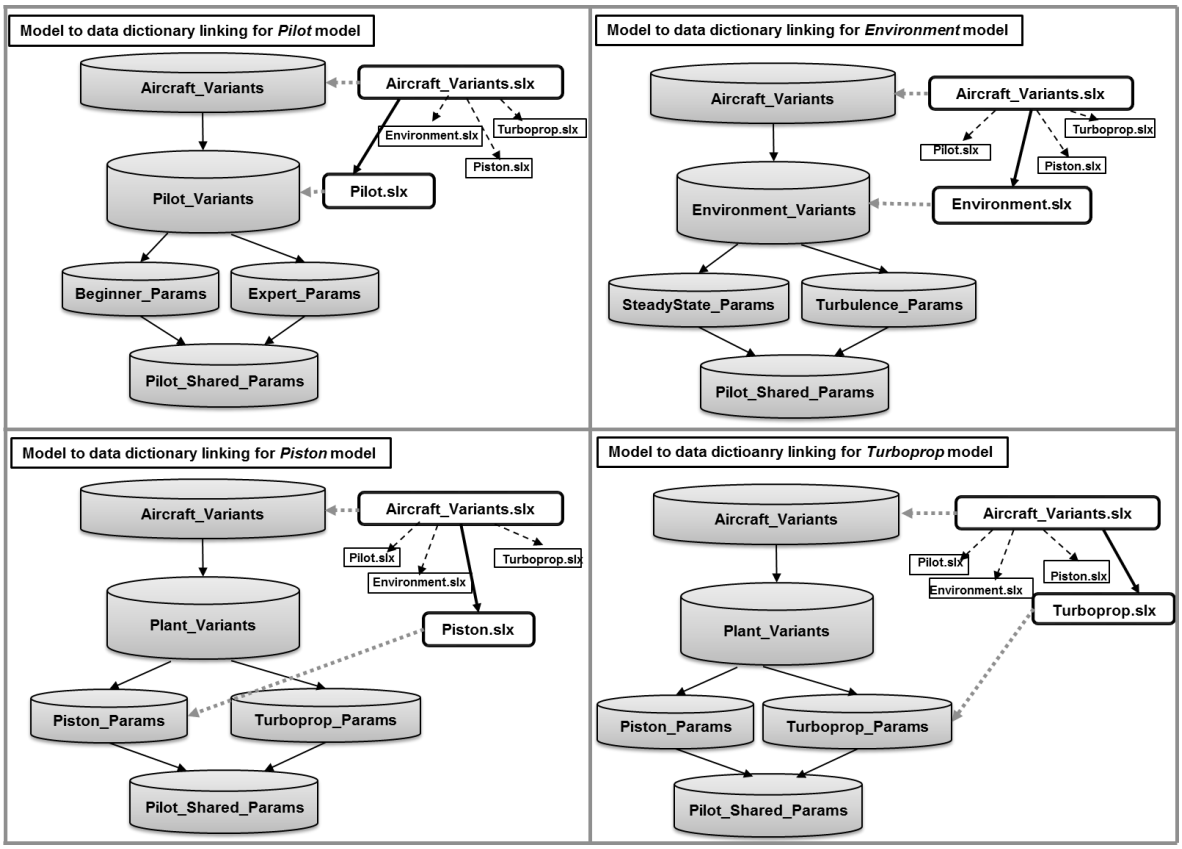

Figure 13: A data dictionary hierarchy does not necessarily correspond to model reference hierarchy

Despite componentization, interdependencies exist among team members contributing to a system level design within a project setting. There is a risk of ad hoc project management where engineers have to learn to work with source control tools or depend heavily on a configuration management specialist within the team for basic tasks[8]. This can lead to process bottlenecks being created, or the abandonment of the process altogether. Simulink Projects is an interactive tool in Simulink for managing project files and connecting to source control software. As shown in Figure 14, it takes a design-centric approach in which the file and project management tasks are exposed to the engineer from within the design tool. By providing flexibility to connect the design tool to various source control tools via an authoring application program interface (API), the amount of the latter tool's exposure for common tasks engineers perform can be managed, while other critical project management tasks still remain with the configuration management specialist.

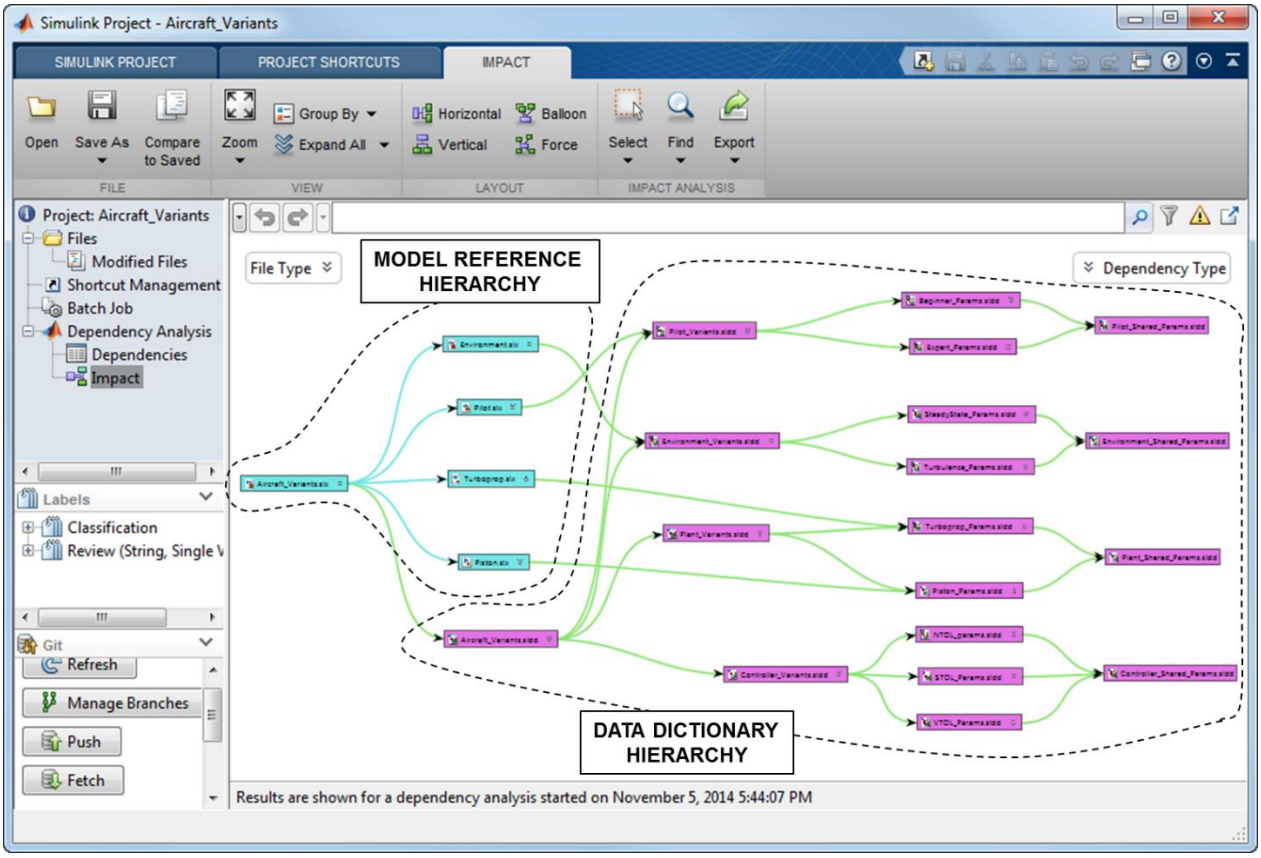

Figure 14: Impact analysis within Simulink Projects interface shows model reference and data dictionary hierarchy as outlines in the example.

D. Variants As An Alternative to Branch Management

One of the challenges of using branching in source control is the cost of merging a branch to the trunk once the feature is completed. This work is redundant if the updates made by a design engineer are restricted to a well-defined component i.e. model reference. Alternatively, she can create a model variant with two components- one that represents the head of the trunk version and the other, as the updated version. Such an approach allows the design engineer to have access to the latest updates to the system level model while working on her feature. In contrast, in the branching workflow the system level model version that she would have to work with would correspond to the head of the trunk when the branch is created.

IV. Conclusion

As discussed, modular design platforms with the ability to incorporate component variants enables reuse for large scale systems. In the context of Model-Based Design, the traditional scripting methodology[1] presents adoption challenges as mentioned in Section III.

In this paper, we presented an approach that addresses these issues through a Simulink example by introducing the Simulink Variant Manager user-interface. In contrast to the scripting methodology[1], Simulink Variant Manager offers an intuitive user interface for variant representation. It provides a view of variants present in the system level hierarchy, with the associated variant objects that contain Boolean logical statements. Variant configurations allow the setting of control variables for switching variants across the model. Constraints prevent the activation of undesirable variant configurations. Furthermore, a validation tool is provided to ensure variant representation consistency. It is important to have a clear understanding of how the variant designs will ultimately map to the market needs. This may require agreement on market requirements involving cross-functional groups spread across marketing, sales, manufacturing, and development.

An interesting application of variant management occurs in a team-based setting where tradeoffs between isolation and sharing of variant data need to be carried out. In this paper, we outlined best practices for managing this

complexity with the use of data dictionaries. As our example showed, it is possible to create a system level model with a hierarchy of separate model reference components and data dictionaries with associations defined between them. It is also possible to create a hierarchy that allows for separation of concerns for parallel development. Since these files would reside in a source control system and accessible through Simulink Projects, engineers can work independently while keeping track of updates made my other members. However, there is no silver bullet and the solution is highly context dependent. Our hope is that organizations will use the best practices outlined in this paper as a foundation to further evolve them. At the very least, we are of the opinion that such approaches will aid in the creation of robust architectures that manage variability more effectively and foster a spirit of collaboration that encourages sharing while respecting the boundaries.

References

[1] Mahapatra, S., "Enabling Modular Design Platforms using Variants in Model-Based Design," AIAA Modeling and Simulation Technologies Conference and Exhibit, Minneapolis, Minnesota, Aug. 2012

[2] Meyer, M.H., Lehnerd, A.P., *The Power of Product Platforms: Building Value and Cost Leadership*, 1st ed., Free Press, 1997, New York, Chap. 2.

[3] Winchester, J., *Civil Aircraft-Passenger and Utility Aircraft: A Century of Innovation*, 1st ed., Amber Books, 2010, London.

[4] Nicolescu, G., Mosterman, P.J., *Model-based design for embedded systems: Computational analysis, synthesis, and design of dynamic systems*. CRC Press, 2009, Boca Raton, FL.

[5] Mosterman, P. J., Zander, J., Hamon, G., Denckla, B., "A computational model of time for stiff hybrid systems applied to control synthesis," *Control Engineering Practice*, vol. 19, 2011

[6] Simulink, Using Simulink, MathWorks, Natick, MA, June 2011.

[7] Mahapatra, S., Priyanka, G., "Design Data Management in Model-Based Design," AIAA Modeling and Simulation Technologies Conference and Exhibit, Kissimmee, Florida, Jan. 2014, (submitted for publication)

[8] Mahapatra, S., Ghidella, J., Walker, G., "Team-Based Collaboration in Model-Based Design," AIAA Modeling and Simulation Technologies Conference and Exhibit, Portland, Oregon, Aug. 2011