Design Data Management in Model-Based Design

Saurabh Mahapatra[1]
Priyanka Gotika[2]
MathWorks Inc, Natick, MA, 01760

One of the keys to successfully harnessing the benefits of graphical modeling in Model-Based Design is the effective management of associated design data. Design data in the form of model parameters and simulation configuration settings are as important to the simulation as the algorithm itself. Together they determine the form and range of the simulation outputs and internal states of the model. However, control engineers have been inclined towards graphical editing and layout than design data management leading to several key design issues especially for large scale systems. These are separation, logical partitioning, change detection, dependency analysis and traceability. In this paper, we present an approach to address these issues using a demonstration within Simulink, an exemplary Model-Based Design environment. A tradeoff decision on embedding of design parameters in the graphical model as opposed to keeping them separate can determine the ease of carrying out a parameter sweep. Another advantage is in the case of modular design platforms where the control variables required to switch variant configurations must be separated out from the design data. Moreover, the use of multiple configuration settings can be used to optimize the performance of the simulation depending on the inherent dynamics present in the model. Lack of logical partitioning of the data can present understandability and data corruption issues within a collaborative environment. Parallel development of graphical components without clear data separation increases the risk of design error. Furthermore, innovative partitioning schemes require a careful understanding of sharing and logical relationships among the data which can be independent of the graphical model componentization. Another decision point in logical partitioning is regarding the storage of design data in a centralized or a decentralized repository system with each offering unique advantages. For example, a decentralized system consisting of multiple files can be put under source control and differenced to earlier versions to detect changes. A change detection workflow that integrates changes in the graphical design with those in the data is key to getting a complete understanding of the changes in the system. As the design begins to scale, it may become necessary to store the design data in a decentralized repository system. However, this does present additional challenges. For example, the tracking of the interdependencies between the graphical model components and the design data partitions becomes critical. An analysis of such dependencies must be carried out to identify any missing components from the project. At a lower level, the traceability of the individual design data to the components is critical to establish the necessary and sufficient conditions for executing a component. Such analysis can also help with the logical partitioning schemes and removing redundant data. One of the big challenges of utilizing the recommendations presented above is in migrating a legacy system to use them. Such migration presents constraints on how much porting can be done in practice. In this paper, we outline several recommendations for addressing these key questions. In conclusion, we offer a set of best practices that engineers can use to manage their data effectively within a team-based environment.

I. Introduction

In several development workflows in Model-Based Design, it is not uncommon for the management of design data associated with a model to be an under resourced activity for creating executable specifications. Since design data is

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

accessed during the simulation initialization step, an approach that requires minimal effort is to simply centralize the data in a global storage repository. This guarantees that well-defined design data is accessible to the entire model as required. This works well during the early stages of the design with low fidelity models and fewer design data. But as the model acquires scale and fidelity, lack of organization makes it increasingly difficult for engineers to trace, track or manage it. This leads to several issues outlined below:

a. Lack of separating the design data:
    i. The risk of deleting any data that may be used by the model results in keeping unnecessary data that is no longer used by the model. This practice leads to redundancy and increases the size of the data in the global storage repository.
    ii. A prerequisite of team collaborative workflows is the componentization of the model that allows various contributors to work in parallel. Since design data stored in a global repository and such separation does not exist, there is an increased risk of data corruption by a team member resulting in incorrect simulation results.
    iii. When a subset of the design data needs to be varied requires special tooling in place such as scripting to change a subset of the data while storing the previous data. This tooling can possibly corrupt the design data. For example, it may be necessary to switch between variant configurations[1] with the associated data.

b. Lack of logical partitioning and dependency analysis: Componentization techniques aid in the development of a model architecture that provide ease of understanding, independent development, and unit testing as benefits. However, such benefits are reduced without a clear understanding of the data associated with the components.

c. Lack of change detection: As the size of the model increases, it becomes increasingly difficult to keep track of the changes to the data over time.

d. Burden of traceability: Diagnosing an issue in a model may warrant a need to trace the data and its associated requirements with a specific component. Since there is no logical organization of the data, such traceability operations can be time-consuming.

This paper takes the viewpoint that creating the executable specification is not limited to designing and managing algorithms but also the associated data. Without such an investment, the benefits accrued through effective algorithm design would be offset by the associated data issues that creep in over the long term.

II. Design Data Management Framework in Simulink

Simulink®[2] is a commercial block diagram environment that enables Model-Based Design[3,4,5]. The system behavior can be modeled as differential and difference equations by using graphical block constructs and signal line interconnections to represent interrelationships among them. Control logic behavior can be visualized by using statecharts. By using graphical abstractions, a complex functional hierarchical decomposition of the overall system can be obtained which can be numerically simulated to verify and validate text-based algorithm requirements. Thus, the Simulink model forms an executable specification of those requirements. Automatic code generation transforms this specification to C/C++, VHDL (Verilog Hardware Description Language) or PLC (Programmable Logic Controller) code that can be deployed on the hardware for rapid prototyping or hardware-in-the-loop simulations for testing real-time requirements of the algorithm. Since the graphical model forms the sole truth of the design in the system, collaboration is enhanced across organizational teams which can rapidly iterate through their designs or refine requirements. In this section, we use Simulink as an exemplary environment to illustrate how some of the above issues can be addressed. Also, the concepts used within the environment can be extended to other tools.

A. Introduction to Simulink Data Dictionary

First, we introduce the concept of a Simulink data dictionary[2, 6] which is a persistent repository of global design data a Simulink 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, which are inputs or outputs of model simulation. The MATLAB workspace is a centralized global repository accessible to both MATLAB programs and Simulink models. Note that the model can still use the MATLAB workspace to store the global design data. 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 and incremental loading leading to performance improvements.

A Simulink® data dictionary is made up of two parts.

- Global Design Data: Contains the named design data that define parameters, signals, and other data objects that define the behavior of the model. Data created or imported in a dictionary are stored in this part.
- Configurations: Contains configuration sets that determine how the model is configured during simulation and code generation. These objects control attributes such as sample time and simulation start time. This part can also store variant configuration objects[1], which store information about variant configurations, active and default variant settings, and definitions of the control variable associated with each configuration.

B. Defining Associations Between a Model and Data Dictionary

A Simulink data dictionary can only be linked to a model or a model reference component. It cannot be associated with a subsystem or a library block. Furthermore, it is possible for multiple models to use the same dictionary but the converse is not allowed. As shown in Figure 2, it is possible to link design data sets stored in separate data dictionaries (*SLDD File A*, *SLDD File B*, *SLDD File C*) at a time to the model (*Model 3*) thus preventing data corruption errors.

Once a model is associated with a data dictionary, it can no longer access design data in the MATLAB base workspace. This is schematically shown in Figure 1 where the MATLAB base workspace is grayed out once the linkage to a data dictionary has been established. The simulation I/O data which is excluded from the data dictionary can still reside in the MATLAB base workspace and be accessible to the models.

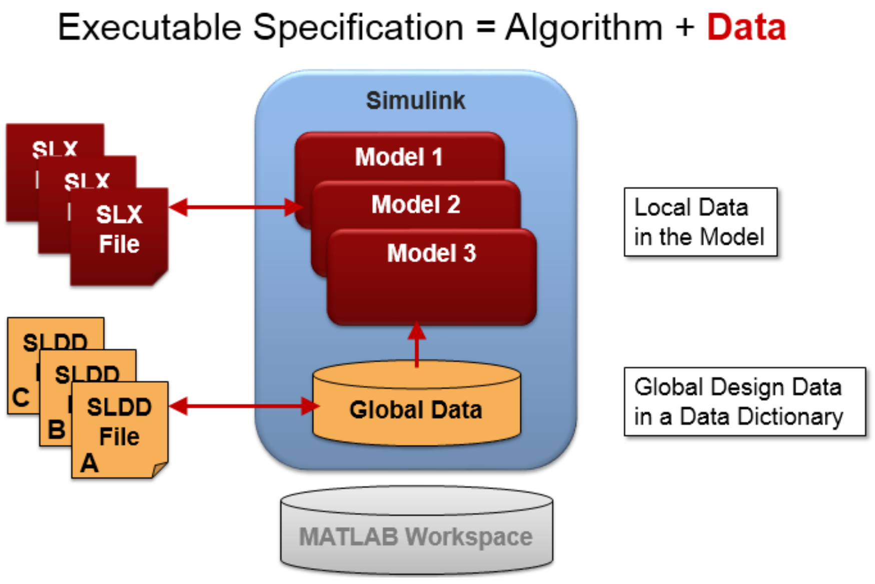


Figure 1: Association between a Simulink model and a data dictionary.

C. Logical Partitioning Using Data Dictionary Hierarchies

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. Figure 2 shows a model hierarchy on the left which does not correspond to the same data dictionary hierarchy. The parent model *System* has two subcomponents *Sub 1* and *Sub 2*. *Sub 2* further has two subcomponents *Part 1* and *Part 2*.

Sub2, *Part 1* and *Part 2* components are linked the *DD2* data dictionary which references a *Shared* subdictionary. *DD2* data dictionary is private with respect to the aforementioned components. However, it uses data that is shared by *Sub 1* which is linked to its own private data dictionary *DD1*. Thus, the design data has separation as *Sub1* component cannot access the data contained in data dictionary *DD2* while *Sub2* component and its subcomponents *Part1* and *Part2* cannot access data dictionary *DD1*. The parent model *System* is linked to a parent data dictionary *DD* that references *DD1* and *DD2* thus ensuring that the data required by all components is accessible. Thus, it is possible to have a wide variety of logical partitioning schemes for a given model reference hierarchy as long as each component is able to access its data.

D. Scope of Design Data

The scope of the design data in the Simulink data dictionary is global in nature. In other words, no two design data elements can have the same name. In Figure 2, every element in the data dictionaries for the system level model *System* has a unique name. This is a limitation imposed by the current implementation. It is quite possible that during the parallel development of *Sub1* and *Sub2* components, there may be design data element names that are coincidentally identical. Since there is a separation of the design data, there would be no issues as long as they are worked on separately. However, a name conflict would arise at the system level model *System*. Any such conflicts are reported to the user and thus resolved by the choice of unique names.

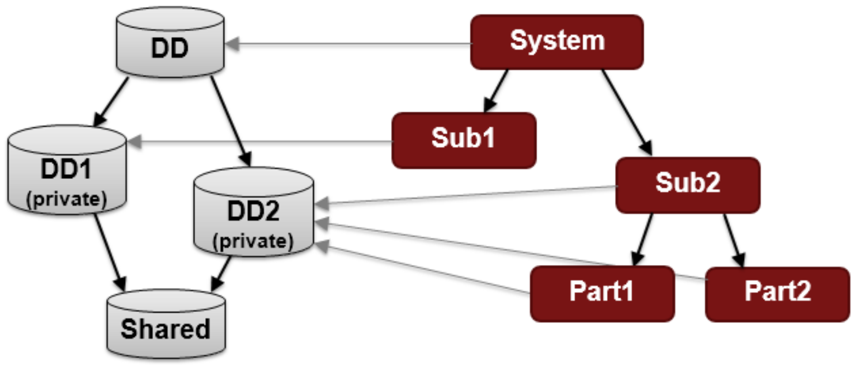
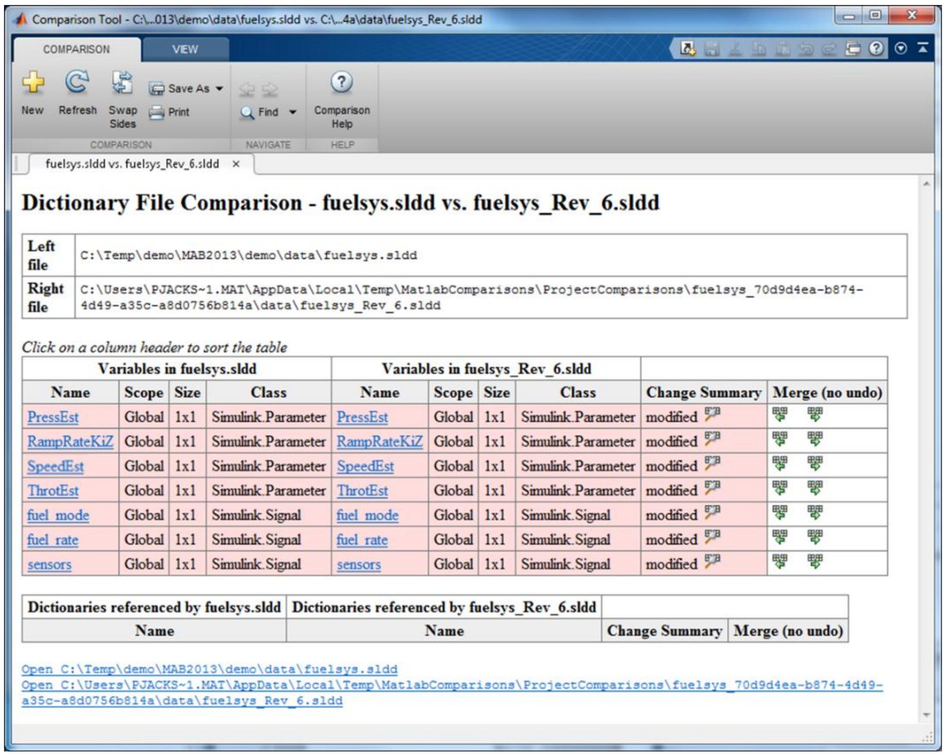


Figure 2: A data dictionary hierarchy does not necessarily have to correspond to a model reference hierarchy.

E. Change Detection

One of the challenges of dealing with design data is the frequent churn that it must undergo during the design phase. Hence, it is necessary that changes to design data be tracked. There are two possible workflows that need to be considered:

- <u>Source control workflow:</u> Since changes to design data to a data dictionary file would be checked into a source control repository, merging tools would be required. MATLAB provides functionality[7] to compare and merge changes between any two data dictionaries as shown in Figure 3.
- <u>Non-source control workflow:</u> In Simulink, it is possible to detect immediate changes made to the design data without a file comparison operation.


Comparison Tool - C:\...013\demo\data\fuelsys.sldd vs. C:\...4a\data\fuelsys_Rev_6.sldd
COMPARISON
VIEW
New
Refresh
Swap Sides
Save As
Print
Find
Comparison Help
COMPARISON
NAVIGATE
HELP
fuelsys.sldd vs. fuelsys_Rev_6.sldd

Dictionary File Comparison - fuelsys.sldd vs. fuelsys_Rev_6.sldd

Left file	`C:\Temp\demo\MAB2013\demo\data\fuelsys.sldd`
Right file	`C:\Users\PJACKS~1.MAT\AppData\Local\Temp\MatlabComparisons\ProjectComparisons\fuelsys_70d9d4ea-b874-4d49-a35c-a8d0756b814a\data\fuelsys_Rev_6.sldd`

Click on a column header to sort the table

Variables in fuelsys.sldd				Variables in fuelsys_Rev_6.sldd					
Name	**Scope**	**Size**	**Class**	**Name**	**Scope**	**Size**	**Class**	**Change Summary**	**Merge (no undo)**
PressEst	Global	1x1	Simulink.Parameter	PressEst	Global	1x1	Simulink.Parameter	modified	
RampRateKiZ	Global	1x1	Simulink.Parameter	RampRateKiZ	Global	1x1	Simulink.Parameter	modified	
SpeedEst	Global	1x1	Simulink.Parameter	SpeedEst	Global	1x1	Simulink.Parameter	modified	
ThrotEst	Global	1x1	Simulink.Parameter	ThrotEst	Global	1x1	Simulink.Parameter	modified	
fuel_mode	Global	1x1	Simulink.Signal	fuel_mode	Global	1x1	Simulink.Signal	modified	
fuel_rate	Global	1x1	Simulink.Signal	fuel_rate	Global	1x1	Simulink.Signal	modified	
sensors	Global	1x1	Simulink.Signal	sensors	Global	1x1	Simulink.Signal	modified	

Dictionaries referenced by fuelsys.sldd	Dictionaries referenced by fuelsys_Rev_6.sldd		
Name	**Name**	**Change Summary**	**Merge (no undo)**

Open `C:\Temp\demo\MAB2013\demo\data\fuelsys.sldd`
Open `C:\Users\PJACKS~1.MAT\AppData\Local\Temp\MatlabComparisons\ProjectComparisons\fuelsys_70d9d4ea-b874-4d49-a35c-a8d0756b814a\data\fuelsys_Rev_6.sldd`

Figure 3: Data dictionary file comparison in Simulink

F. Dependency Analysis[8]

Although a large scale model can be componentized into model references and a data dictionary hierarchy, there is no requirement to have a strict partitioning in place. Therefore, tooling may still be required to identify the relationships between the model and the data dictionary files. First, the relationship among the component files and the relationship between the component files and the relationship among the data files need to be mapped. Second, relationships among the data dictionary files would also need to be identified. Furthermore, there are two kinds of dependency analyses that would need to be provided:

- 'Required by' relationship: This relationship identifies all the upstream dependencies for a model or a data dictionary.
- 'Impacted by' relationship: This relationship identifies all the downstream files that would be impacted by a change to a model or a data dictionary file.

An implementation of impact analysis in Simulink is shown in Figure 4. The impact analysis graph shows the same dependencies as those we had specified for the design and introduced in Figure 2.

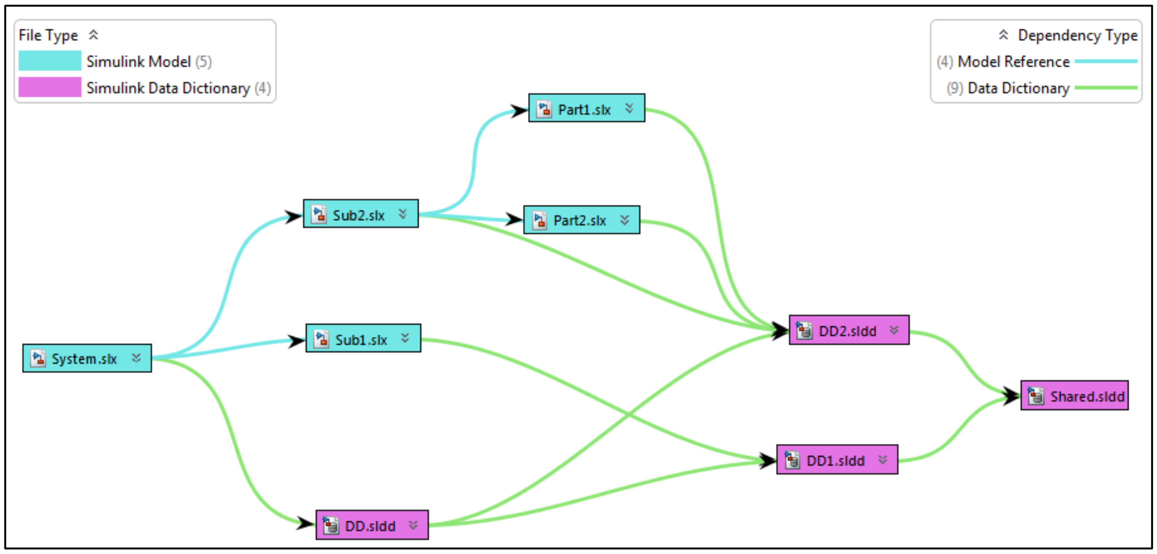


Figure 4: Impact analysis graph in Simulink Projects for the model and data specification described in Figure 2

G. Requirements Traceability[9]

One of the key activities of Model-Based Design is the linking of requirements to specific parts of the design models[10]. Seldom are requirements linked to the design data. The lack of such traceability lowers the understanding of the entire model. For example, a fixed point implementation of the design data may be required during the deployment of the final algorithm on a fixed point processor. Since this data is not linked to any requirements, it can lead to the design engineer developing the model with floating point data. In the Simulink data dictionary, it is possible to link each design data variable to a requirement document authored in a requirements management tool like IBM® Rational® DOORS®, Microsoft® Word or Microsoft® Excel®.

III. Best Practices for Design Data Management

In this section, we cover best practices for creating data dictionaries and managing them within a team environment.

A. Migration Workflows

Consider the example shown in Figure 5A. On the left side of the figure, the system level model uses 8 variables *(a,b,c,d,e,f,g,h,i)* stored in the MATLAB workspace. The components and their associated variables are shown on the right. The task is to create a data dictionary hierarchy that maps to the model hierarchy as shown in Figure 5B. A methodology needs to be developed that identifies common variables between multiple components in a model and stores them in a shared data dictionary. As shown in Figure 5B, variables *(d,f,h)* which are used by *Sub1* and *Sub2* should be isolated.

Here we propose an algorithm that can be used to place the design data into the data dictionaries for a specified hierarchy. The steps are outlined below:

- **<u>STEP 1:</u>** Link each model with its corresponding data dictionary as shown in Figure 5B.
- **<u>STEP 2:</u>** For each model in the model reference hierarchy, find all the variables used by that model excluding its children. The result of this operation is shown on the right hand side of Figure 5A.
- **<u>STEP 3:</u>** For each model, place the variables obtained in **STEP 2** in the linked data dictionary. Note that at this step there will be duplicate variables across data dictionaries as shown in Figure 6
- **<u>STEP 4:</u>** Identify the children of the top level data dictionary. List all possible combinations of the data that can be shared by these children. For example, consider a data dictionary with three subdictionaries. There will be four possible combinations, three of which will correspond to the data shared by any two data dictionaries while the fourth will correspond to the data shared by all the data dictionaries. For each such combination, identify the common variables in the corresponding data dictionaries. This is equivalent to the intersection operation on the variables contained in these data dictionaries.

In Figure 6, the top level data dictionary *DD* has two children *DD1* and *DD2*. There is only one possibility of shared data between *DD1* and *DD2*. Thus the intersection operation of data dictionaries *DD1* and *DD2* will give *(d,f,h)* as shared variables.

- **<u>STEP 5:</u>** Traverse the combinations in a specific order that corresponds to the maximum number of data dictionaries that can share data. For example, if a data dictionary hierarchy has four subdictionaries, *X1*, *X2*, *X3* and *X4* then the order in which they should be traversed is: **intr**(*X1*, *X2*, *X3*, *X4*), **intr**(*X1*, *X2*, *X3*), **intr**(*X2*, *X3*, *X4*), **intr**(*X1*, *X3*, *X4*), **intr**(*X1*, *X2*, *X4*), **intr**(*X1*, *X2*), **intr**(*X2*, *X3*), **intr**(*X3*, *X4*), **intr**(*X1*, *X4*), **intr**(*X1*, *X3*) and **intr**(*X2*, *X4*) where **intr** returns the common variables shared by the data dictionaries listed as its arguments.
- **<u>STEP 6</u>:** Find out if the corresponding data dictionaries share a common data dictionary. If they do, place the shared data in the shared data dictionary. If they do not, then there is an error in the data dictionary hierarchy as any two dictionaries cannot have duplicate variables. In Figure 6, *DD1* and *DD2* reference the *Shared* data dictionary where the common variables *(d,f,h)* can be placed. This operation will result in the variable placement of data dictionaries as shown in Figure 5B.
- **<u>STEP 7:</u>** Repeat **STEPS 4, 5**, **6** and **7** for each of the reference subdictionaries until the entire data dictionary hierarchy is traversed.

Note that **STEP 3** guarantees that all the design daata required by a component in a model reference hierarchy available despite the data being pushed down into the data dictionary hierarchy in **STEPS 4, 5**, **6** and **7**. It is also possible to construct a compact data dictionary hierarchy where each data dictionary is linked to a model in a model hierarchy such that each data dictionary has no redundant variables i.e. variables not used by the model.

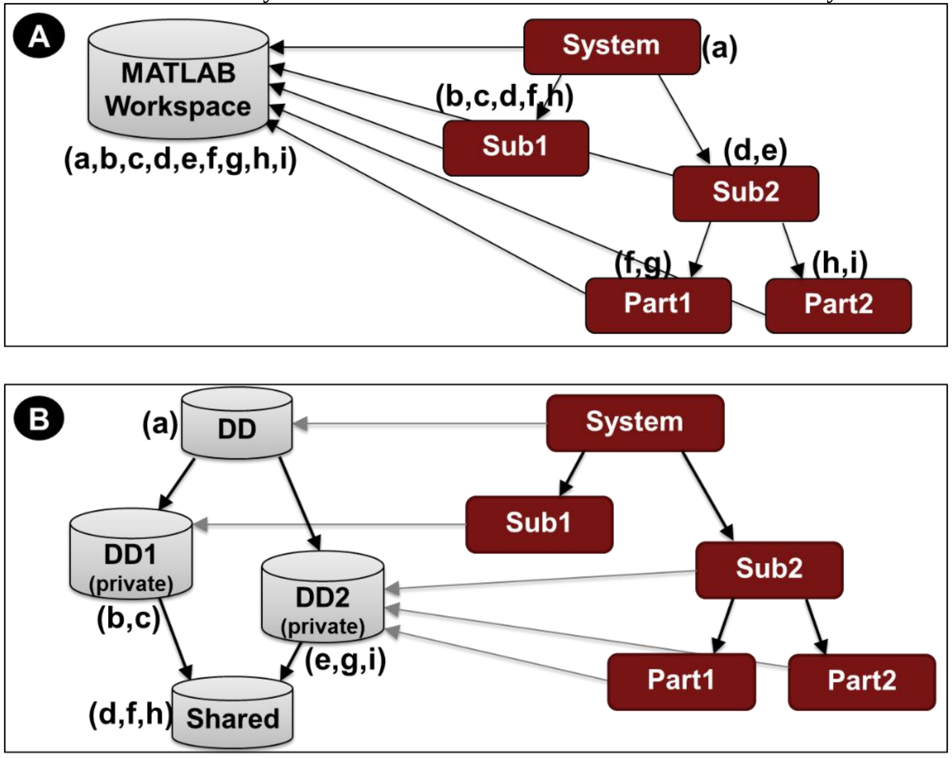


Figure 5: Migration of variables in MATLAB workspace to a specified data dictionary hierarchy

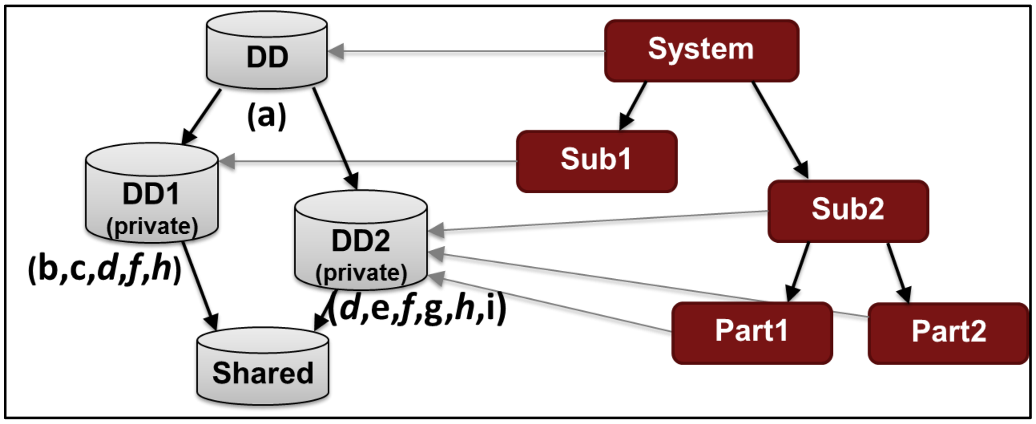


Figure 6: Variables placed in the data dictionary as described in Step 3

B. Logical Partitioning Scheme Development

In this section, we propose exemplary data dictionary schemes within a team environment that allows for collaboration and isolation. The guiding principles in creating these examples should motivate the reader to construct their own data dictionary hierarchy that addresses the team's data usage needs.

a. Models with no shared data: Here we cover three possible scenarios as outlined below:

- Temporary prototype model: During the prototyping phase, it is not uncommon for an engineer to make rapid design changes to the model and the associated design data. It is also typical for engineers to work in isolation during this activity. As shown in Figure 7A the recommendation is to store the design variables in a global workspace that can be accessed by all programs inside the design environment. In our example, this would correspond to using the MATLAB base workspace.
- Stand-alone system model: As the individual engineer's design progresses and matures beyond the conceptual stage, the number of design data variables would increase. Since the engineer is still working in isolation, it is recommended that the model be linked to a single data dictionary.
- System model with library subcomponents: Once the engineer's model becomes large, it may be necessary to componentize the model to manage design complexity. As shown in Figure 7C, the system level model uses the components stored in libraries. Since these components would be used across multiple models and the engineer is working in isolation, it is recommended that the design data used by each instance be placed in the data dictionary associated with the system level model.

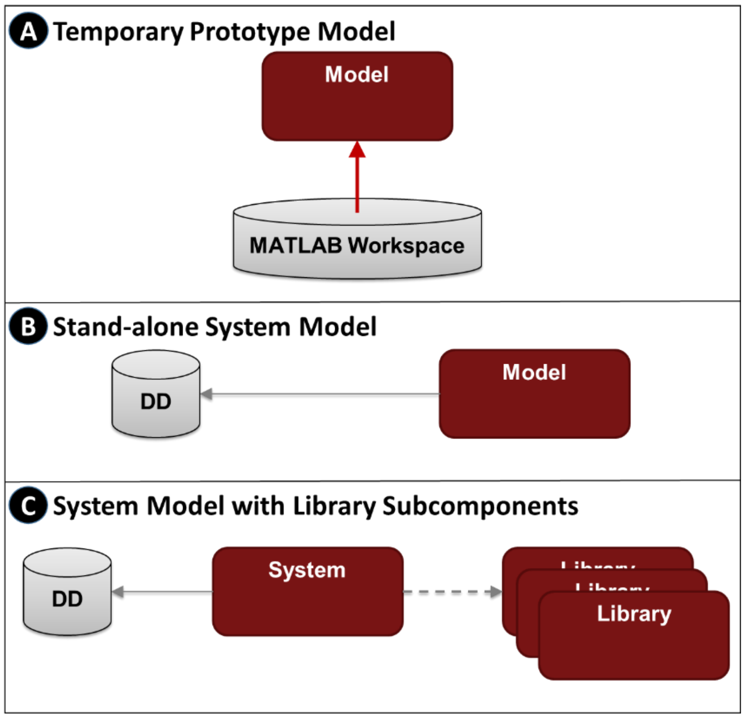

A Temporary Prototype Model
Model
MATLAB Workspace
B Stand-alone System Model
DD
Model
C System Model with Library Subcomponents
DD
System
Library
Library
Library

Figure 7: Data dictionary schemes hierarchy for prototyping stand-alone model and componentized development.

b. Subcomponent models with shared data: We present scenarios with models using shared data and multiple owners

- Single user: Since the user works in isolation, the risk of data corruption owing to team collaboration is nonexistent. In such a scenario, it is recommended that all the components in the model hierarchy be linked to a top level data dictionary. As shown in Figure 8, the top level data dictionary can have a hierarchy for logical partitioning of the design data.
- Multiple users working closely together: In this scenario all the users are collaborating closely and hence it is recommended that a shared data dictionary be used in the data dictionary hierarchy to identify design data that can impact multiple components. Also, non-shared data should be placed in private data dictionaries that reference the shared dictionary as shown in Figure 8B. Since, the private dictionaries are files themselves this allows for the isolation of the design data that is only being used by a component. There is still risk of design data corruption in the shared data dictionary owing to an error made by a team member. However, close collaboration would entail that the team members agree upon a process for changing the data in the data dictionary.
- Multiple users working independently: As shown in Figure 8C, there are two teams each of which desire to have some degree of isolation between them. However, there is close collaboration within each team. To promote close collaboration for the team developing *Sub2* component, all the shared data are placed in a shared data dictionary called *DD2s.* The team developing *Sub1*, will not have access to this shared data dictionary. For the data shared between the two teams, will be placed in the shared data dictionary *DD1s* which is referenced by the data dictionaries linked to *Sub1* and *Sub2, (DD1and DD2 respectively)*. From the perspective of the teams, *DD1s* will be a shared public data dictionary for both teams to access and *DD2s* will be a shared private data dictionary that only the team developing *Sub2* will access.

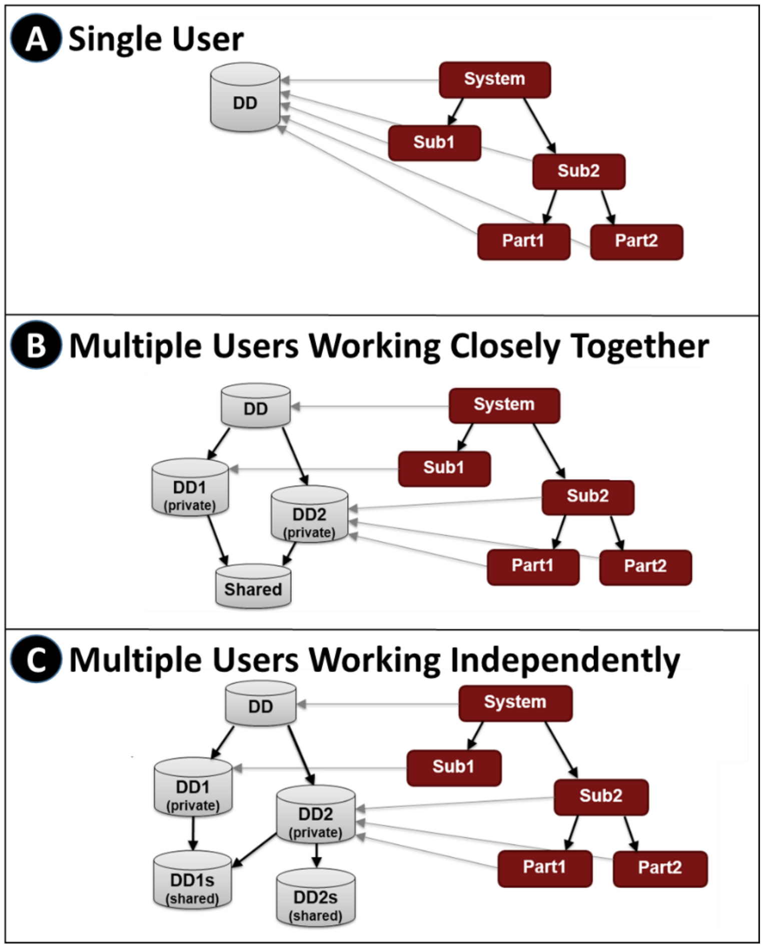


Figure 8: Data dictionary schemes for managing subcomponent models with shared data.

c. Manage global data outside design environment: It is possible to store the design data in an external dictionary that is linked to a Simulink data dictionary. As shown in Figure 9A, there would be read and write operations that define the interaction between these dictionaries.

d. Simulate multiple independent systems: As shown in Figure 9B, *System1* and *System2* use *DD1* and *DD2* data dictionaries respectively. Since it is possible to store configuration set data that may contain simulation settings, it is possible to simulate each of these components independently of each other. For example, if *System1* has stiff elements in it, then its simulation settings may require stiff ODE solvers to improve simulation performance. However, note that the system level model that references these components would need to use a single configuration set that is consistent across all components.

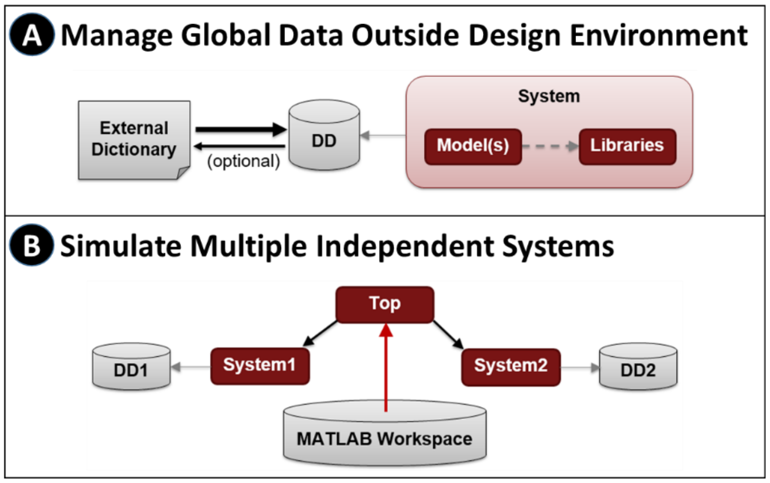


Figure 9: Data dictionary schemes for managing data outside design environment and simulating multiple independent systems.

C. Team Collaboration Workflows for Modular Design Platforms

A modular design platform is a finite set of components and their associated interfaces that can form a fixed common structure while allowing for some variability[1]. In Simulink, the approach taken to represent variable components is either through the use of variant subsystems or model reference variants. The former provides a mechanism in which variant configuration can be incorporated within a single model file while the latter allows for them to be implemented in separate files. This framework requires the use of three kinds of data:

- <u>Control variables:</u> These variables would be used for activating the Boolean logic statements contained in variant objects for activating a configuration within a variant configuration data object.
- <u>Variant objects:</u> For each variant in the model, a variant object that contains a Boolean logic statement can be activated by control variable settings.
- <u>Variant configuration data objects:</u> For each combination of control variable setting, it is possible to create a configuration in the variant configuration data object. When a configuration is activated then the variant choices corresponding to those control variable settings would get activated in the Simulink model.

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 model references 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 model references and hence separate files.

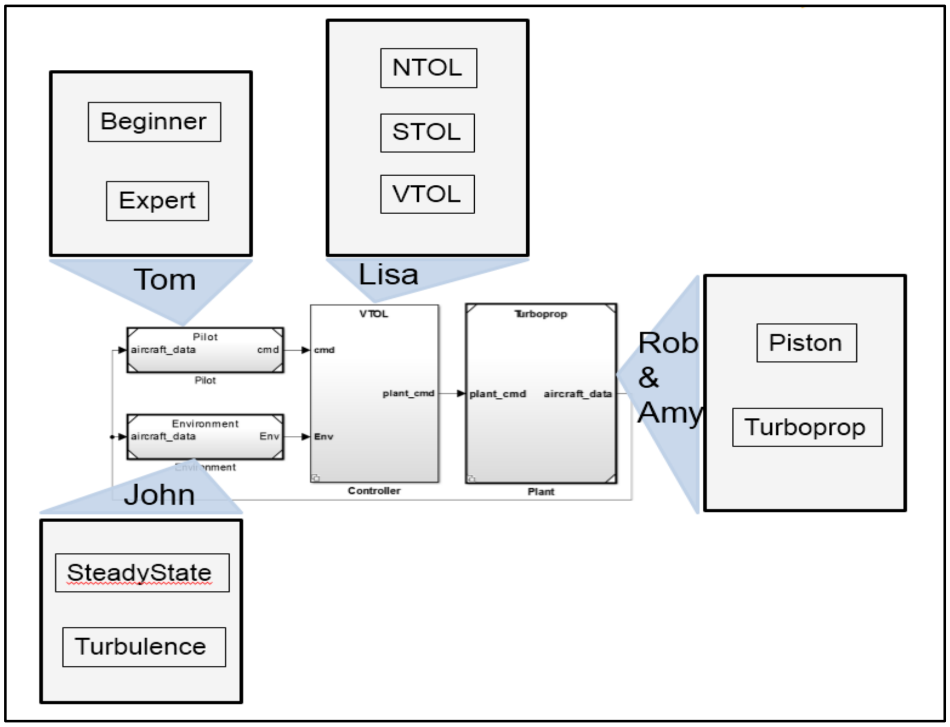


Figure 10: Ownership of the variants in the system level model

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. Data dictionaries for component level are partitioned based on the type of data 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 four 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 *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.

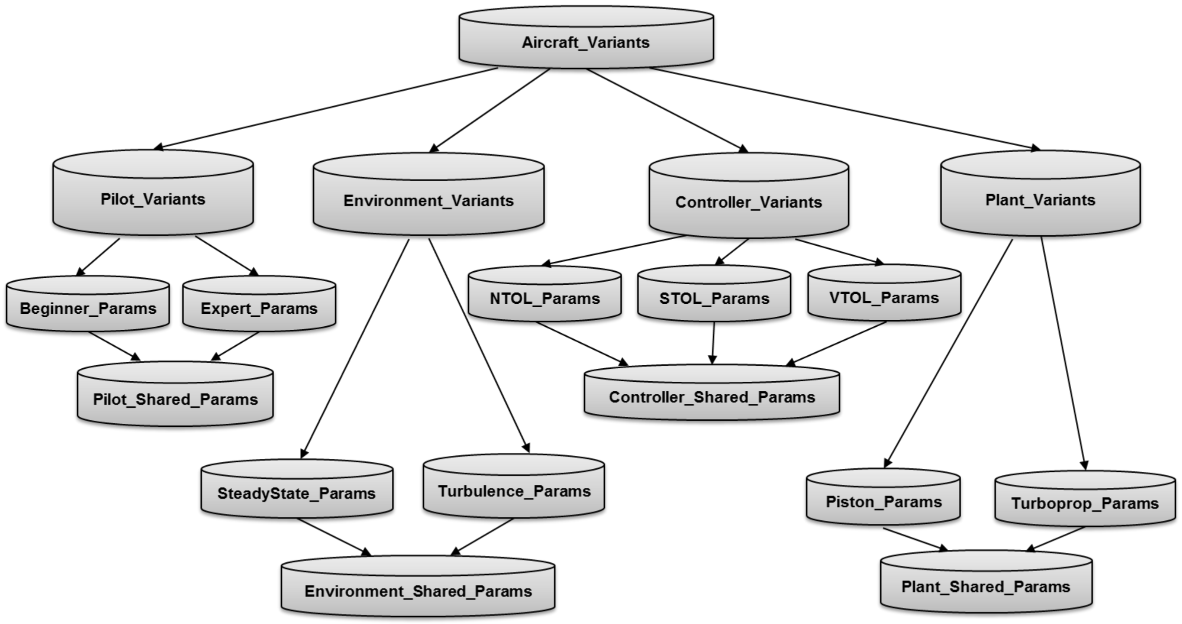


Figure 11: Data dictionary hierarchy

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 four 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. *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.

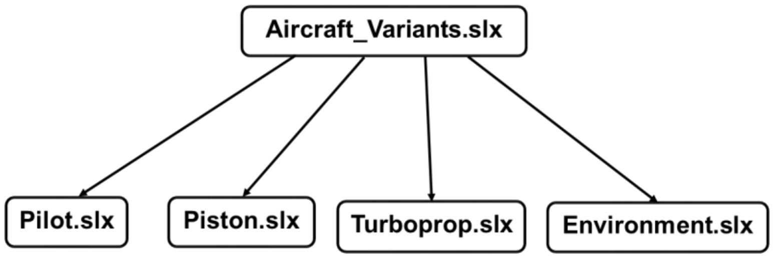


Figure 12: Model reference hierarchy

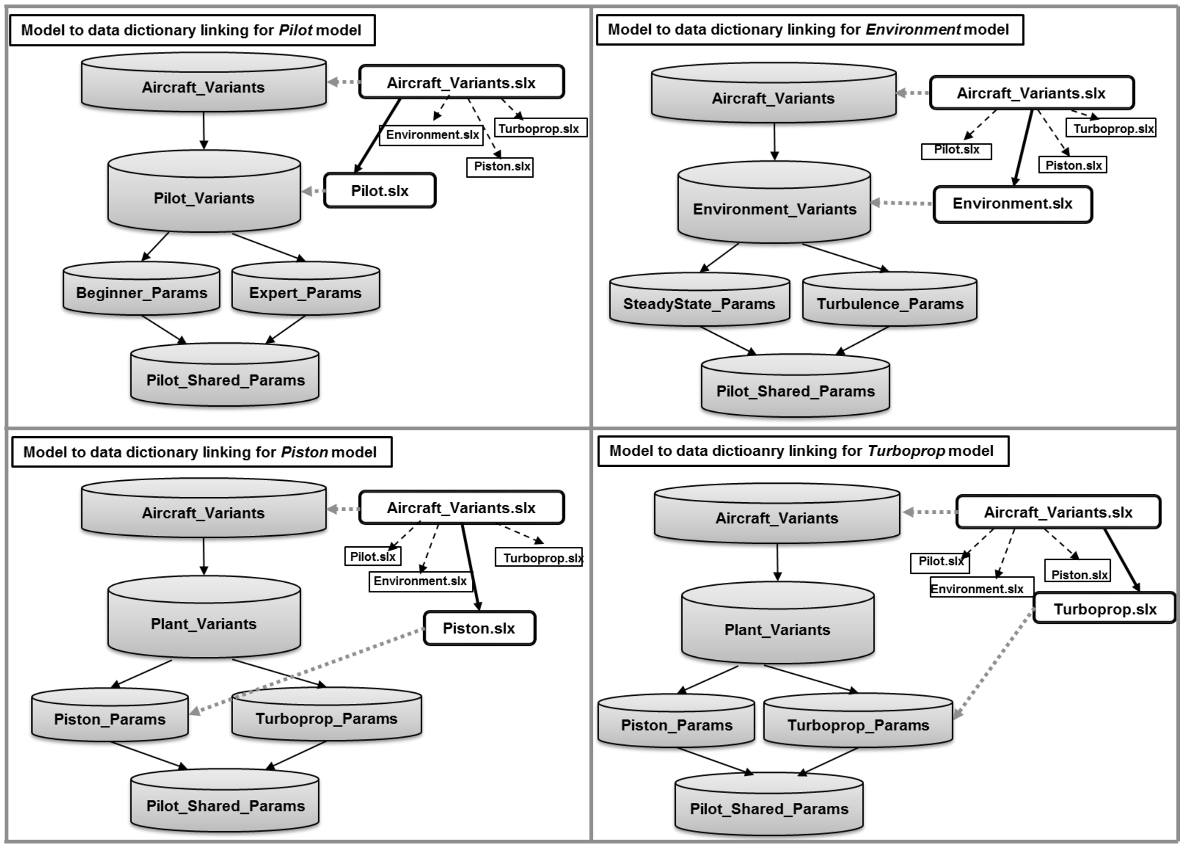
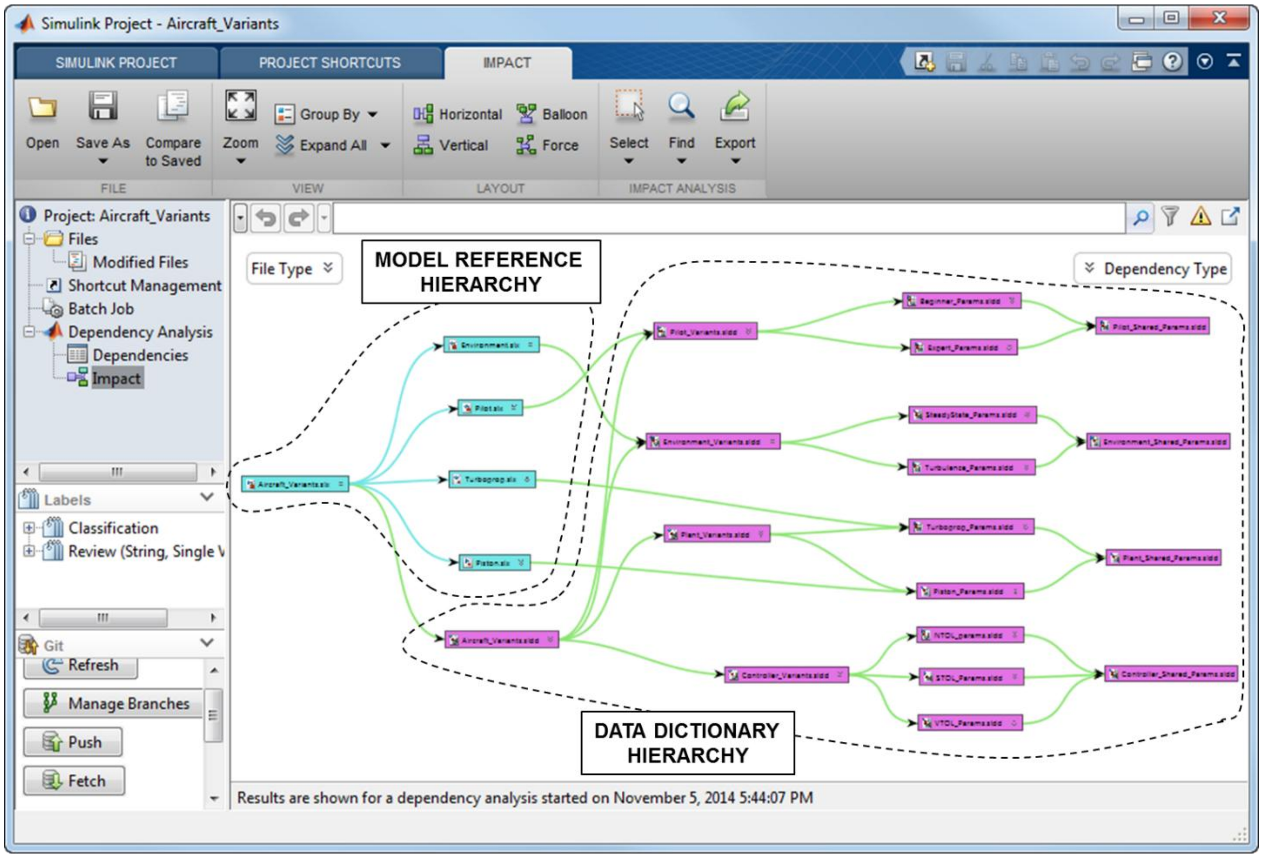


Figure 13: A data dictionary hierarchy does not necessarily correspond to model reference hierarchy

Despite componentization, interdependencies exist among team members contributing to a system level design within a project-based 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[4]. 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.



Results are shown for a dependency analysis started on November 5, 2014 5:44:07 PM

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

IV. Conclusion

In developing large scale system models, control engineers have been focused more on graphical editing and layout than on design data management. This has led to several key design issues, including separation, logical partitioning, change detection, dependency analysis and traceability. The concept of a data dictionary brings design data on par with graphical modeling.

In this paper, we presented an approach that addresses these issues through a Simulink example by introducing the Simulink data dictionary. We showed how the logical partitioning of the data increases understandability and avoids data corruption issues within a collaborative environment. Parallel development of graphical components can also be facilitated with clear data separation reducing the risk of design error. Furthermore, we introduced partitioning schemes based on sharing needs and logical relationships among the data. We also demonstrated a change detection workflow that also includes data differencing for enhanced understanding of all the changes in the system.

As the design begins to scale, the tracking of the interdependencies between the graphical model components and the data dictionaries becomes critical. The dependency analysis tool in Simulink projects identifies any missing components from the project. At a lower level, the traceability of the individual design data to the components is critical to establish the necessary and sufficient conditions for executing a component.

We also introduced logical partitioning schemes based on team structure that balances collaboration and isolation requirements. We also proposed an algorithm that migrates a legacy system to use these schemes. Such migration presents constraints on how much porting can be done in practice.

Another advantage is in the case of design of modular design platforms where the control variables required to switch the variant configurations must be separated out from the design data. Using a Simulink example, we offer a set of best practices that engineers can use to manage such data effectively within a team-based environment.

References

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

[2]Simulink, Using Simulink, MathWorks, Natick, MA, March 2014.

[3]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.

[4]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

[5]Barnard, P., "Graphical Techniques for Aircraft Dynamic Model Development," AIAA Modeling and Simulation Technologies Conference and Exhibit, Providence, Rhode Island, Aug. 2004, CD-ROM. [6]Vitkin, L. and Fallahi, A., "The role of the Data Dictionary in the Model-Based Development", *SAE World Congress & Exhibition 2009*, Detroit.

[7]Simulink Report Generator, About Simulink Model XML Comparison, MathWorks, Natick, MA, October 2014.

[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

[9]Simulink Verification & Validation, Requirements Traceability, MathWorks, Natick, MA, October 2014.

[10]Mosterman, P. J. and Ghidella, J., "Requirements-Based Testing in Aircraft Control Design" *Proceedings from the AIAA Modeling and Simulations Technologies Conference and Exhibit 2005*, August 15-18, San Francisco, California, 2005.