2007-01-1775

# Configuration Management of the Model-Based Design Process

**Gavin Walker, Jonathan Friedman, and Rob Aberg**
The MathWorks



## ABSTRACT

Today, many leading automotive OEMs and Suppliers are adopting Model-Based Design for the development of embedded systems applications. In this paper, the authors review the challenges of performing configuration management that is adequate for use in a production environment of the models and associated files central to Model-Based Design.

## INTRODUCTION

A robust configuration management process allows teams to manage the development and use of the models within a large team project, including the various model files themselves; the configuration of a model built from many subcomponent models; and the model data (parameters and signal specifications).

In addition to these traditional configuration management requirements, within a production environment a configuration management tool may be used to store artifacts that document Model-Based Design, including simulation results, test harnesses used to verify the design (both inputs and outputs), generated code and logs, and reports of formal coverage tests. To ensure reproducibility, the system may catalogue the work environment – that is, the versions of the OS, modeling tools, compilers, etc.

Lastly, configuration management tools can be used to manage the many model variants that are created to capture both the different systems being controlled, as well as the complexity of the controllers. For example, on a given vehicle program, engineers may need to develop controllers for luxury-, mid-, and base-level vehicles, with differing requirements for the North American, Asian, and European markets.

In this paper, the authors discuss how configuration management tools can be used throughout Model-Based Design.

## CONFIGURATION MANAGEMENT (CM)

A configuration can be defined as "an arrangement of parts or elements." In this paper, we focus on those parts or elements associated with Simulink models. By *managing* the configuration of a model, we hope to provide an example of how one might manage arbitrary combinations of files as a configuration.

In particular, we aim to provide a mechanism that allows us to easily define sets of files that are compatible with each other, so that when other team members obtain such a configuration of files from revision control they are confident that the combination will work together. This confidence can enable the sharing of work within a company, and the parallel development of large Simulink models. Additionally, by recording additional information, or *metadata*, we aim to make it possible to extract from the revision control system the precise set of files that were sent to a supplier or customer, or which are simply the latest *good* configuration of files.

We define a few terms before proceeding. In particular, we avoid the term *version* and instead use *revision*.

- Revision Control Software: Software that stores files with unique revision numbers, and allows them to be retrieved by name and revision number.
- Repository: The place where the Revision Control Software stores files.
- Revision: A specific instance of a file stored within a Revision Control Software tool.
- Configuration: A collection of specific revisions of a number of files that work together.
- Root Model: We take a Simulink-centric view of configuration management in this paper. We assume that each project contains a root model, which is the model that a user will open to begin work on the project. This model may link to other models.

An overview of a typical configuration management process is given in Figure 1. We note the use of the terms “check in” and “check out”, which we use in this paper. Note that the meaning of these terms varies

between different revision control and configuration management tools.

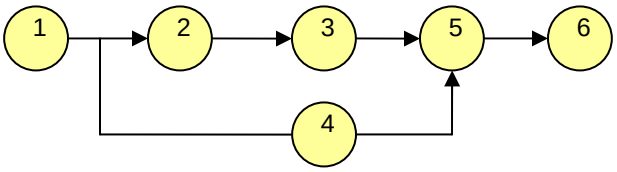


**Figure 1:** Steps in a typical configuration management process.

1. **Get a Configuration.** Get a number of files, which have been denoted as forming a mutually consistent "configuration", from the revision control repository.
2. **Create a File.** The user creates a new file on his or her local computer.
3. **Add to Revision Control.** Make the revision control system aware of the new file. The only instance of this file is on the user's local computer. It is not in the repository. The user can still modify it.
4. **Check Out.** The user wishes to work on an existing file from revision control. The instance of the file on the user's computer becomes editable. Other users are made aware that this user is editing this file.
5. **Check-In**. Place the modifications made by the user into the revision control repository. Give this file a new, unique revision number to indicate that it is different from its predecessors. The user's local copy is no longer editable. This new revision is available to all users.
6. **Create a New Configuration**. Label a number of files as being a new, mutually consistent "configuration". Other users can get this set of files from the revision control system.

To clarify how we differentiate configurations from revisions, consider the following example. We have a project that consists of just three files: `Project.mdl`, `importantLibrary.mdl`, and `newFeature.mdl`.

In **Figure 2**, we illustrate the development progress of the project. We note the following points:

- The first configuration, defined when the project was created and denoted *Rev1*, contains only `Project.mdl`, revision 1 and `importantLibrary.mdl`, revision 1.
- The same revision of a file can appear in a number of different configurations: `importantLibrary.mdl`, revision 1 is part of both the "Project Creation" and "Release 1" configurations.
- Some revisions of a file will may appear in any configurations: for example, `newFeature.mdl` revisions 3 and 4.
- Configurations are being used here to denote major events, such as external releases of the project, as well to denote which is the current "good" set of files. The implication of **Figure 2** is that `importantLibrary.mdl`, revision 5, is not part of the "current working" configuration, suggesting that it is perhaps untested, contains a bug, or is simply not compatible with the latest revisions of `Project.mdl` and `newFeature.mdl`.

| `Project.mdl` | `importantLibrary.mdl` | `newFeature.mdl` |
|---|---|---|
| *Rev1* | *Rev1*,**Rev1** | Rev1 |
| **Rev2** | Rev2 | ***Rev2*** |
| ***Rev3*** | Rev3 | Rev3 |
| Rev4 | ***Rev4***,REV4 | Rev4 |
| REV5 | Rev5 | REV5 |

Configurations: Project Creation – *Italics*, Release 1 -- **Bold**, Release 2 – ***Bold-Italics***, Current Working – ALL CAPS

**Figure 2:** Illustration of revisions and configurations.

## LINKING TO A CM TOOL

To implement confirmation management within Model-Based Design, we need to set up an interface from MATLAB® and Simulink® to a revision control (RC) tool. There are a number of ways to do this, which depend upon the particular RC tool, the operating system that the tools will be used with, and the sophistication required from the interface.

The most straightforward way to put the files associated with Model-Based Design under the control of revision control software is to treat them as two separate tools. The user will have access to the full functionality of the revision control software through its standard user interface. Why might that be suboptimal?

- Engineers have to use two tools, and remember to use them. A common failure mode is for an engineer to begin work in Simulink without checking out the required files first.
- Analysis can be performed within MATLAB and Simulink to determine the dependencies of various files upon each other. A third-party RC tool is unlikely to understand such dependencies.
- Having two separate tools makes it harder to enforce a workflow and to guide the user to do the correct thing. By writing a layer between the revision control software and the Simulink models and associated files, we can enforce such a process.

There are several available methods to link from MATLAB into a revision control programs. The following list is illustrative and is not meant to be exhaustive.

### THE MICROSOFT SOURCE CONTROL COMMON INTERFACE (SCCI) API

One published API for revision control is the Microsoft® Source Code Control Interface (SCCI) API, supported on Microsoft Windows. This API allows communication between any revision control package and application that implements the API. MathWorks tools use this

interface to perform basic revision control operations directly from the MATLAB current directory browser, from the Simulink file menu, and from the MATLAB command-line.

Any source control package implementing this interface on a Windows PC will appear in the Source Control section of the MATLAB Preferences User Interface. For a full description of this interface see the MATLAB documentation section entitled “Source Control Interface on Windows Platforms.”[1] A limitation of this interface is that, to the authors’ knowledge, it does not support tagging or labeling of a number of files to create a configuration set.

CUSTOM INTERFACE TO REVISION CONTROL TOOL

Many revision control packages publish an API that allows other tools to interface to them directly. For example, The MathWorks uses the published interface into IBM® ClearCase® on UNIX platforms.

The advantage of such interfaces is that the full functionality of the revision control software can be made available. There are, however, disadvantages. The client side interface has to be written from scratch, and then maintained. The interface has to checked and updated with each release (major, minor, or bug-fix) of the source control tool. Such maintenance costs can be high.

HINTS, TIPS, AND COMMON STUMBLING BLOCKS

Setting up a source control tool to work with MATLAB and Simulink is straightforward. We note the following points:

- Simulink .MDL files are stored as plain, ASCII, text. Some revision control tools will automatically attempt to merge changes to text files if two people have edited the same file. This behavior may not always be successful for Simulink files, and we recommend setting the revision control tool to not merge .MDL files.
- Recall that MATLAB .MAT files are binary. Many revision control tools will recognize them as such. However, to ensure that the desired behavior, we recommend setting the revision control tool to explicitly not attempt to merge .MAT files.
- Some older revision control tools may have problems with MATLAB object directories – i.e., directory names that being with an “@” symbol. Most can be configured to allow this. (The “@” symbol is used in some CM tools for macro substitution.)

APPLICATIONS FOR THE INTERFACE

Once an interface exists between MATLAB and Simulink and a revision control or configuration management tool many tasks can be performed from the MATLAB command line, by GUI-applications developed within MATLAB, or using utility functions within MATLAB and Simulink. A MATLAB GUI application might make configuration management tasks more straightforward by:

- Providing a GUI-interface to common RC and CM tasks
- Using a GUI to guide users through the correct CM process by enabling and disabling GUI functions as necessary
- Extracting information from RC tool and display it in a user-friendly manner within Simulink, without requiring the user to interact with, or have any detailed knowledge, of the RC tool. This is illustrated in **Figure 3**.

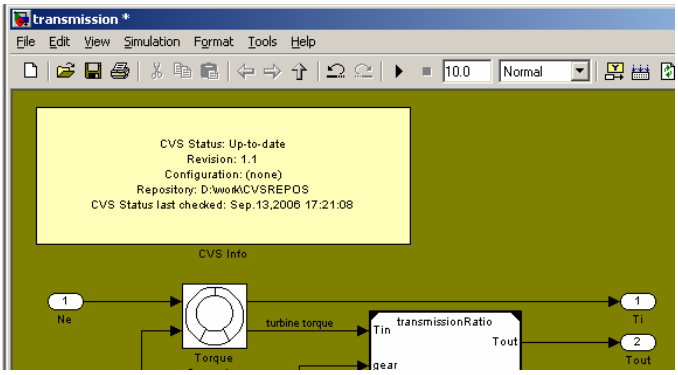


**Figure 3:** Automatically updated revision control information available on a Simulink block.

## COMPONENT REUSE VIA REVISION CONTROLLED PALETTES

The first step in using a revision control system within Model-Based Design is to architect the model in such a way that each controlled portion of the model has only one primary owner or editor. To accomplish this goal, one needs to create a balance to achieve a component that is large enough to be useful but small enough to be reusable. This task is by no means trivial and needs to be supported by a process that allows for the evolution of the components as content is added to the model components. Once the model architecture has been established, there are different mechanisms that can be used to store and integrate the components.

USING SIMULINK LIBRARY LINKS

Consider the Simulink library shown in Figure 4. This model contains a number of alternative implementations of a particular control system. In general, this might be just one of a series of libraries of useful components, grouped by application or type.

We note that each of the subsystems in `control_palette_libs` is itself a library link. Each of these subsystems is stored in their own individual Simulink libraries, thus achieving the goal of “one item of functionality per file”. When a subsystem, such as

`Analog Control`, is copied from `control_palette_libs` into a model, Simulink follows the link back to the file where the subsystem is actually implemented. Hence the model that the user creates contains a link back to the library where Analog Control is actually implemented, in this case, `AnalogControlLib`.

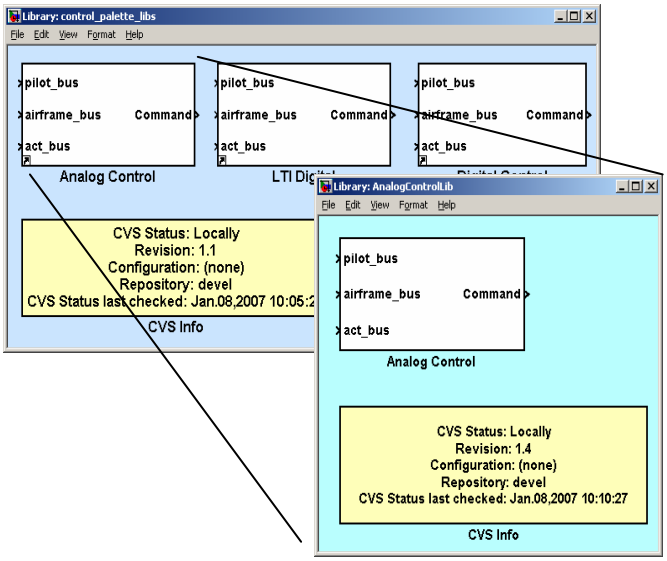


**Figure 4:** Revision controlled “palette” using library links.

This structure achieves two things:

- An engineer can control what items go in the palette. There might be many implementations of controllers with revision control, but only approved ones end up on the palette. The palette is itself within revision control. (For example, in the illustration here, the palette is at revision 1.1.)
- The implementations of the individual control algorithms are stored in libraries that contain just one item of functionality. This allows each of the control algorithms to be separately version controlled. (For example, in the illustration here, the palette is at revision 1.4.)

USING MODEL REFERENCE LINKS

An alternative to using libraries of libraries is to use a library of model reference components, as illustrated in Figure 5. If one of the Model Reference components is copied from the palette into a Simulink model, then subsequent analysis will show only a reference from the item in the new model to the corresponding referenced file. As for the implementation using library links, there will be no references stored to the palette.

We recommend that Model Reference is used in preference to library links to partition a large model into components that are to be individually stored under revision control.

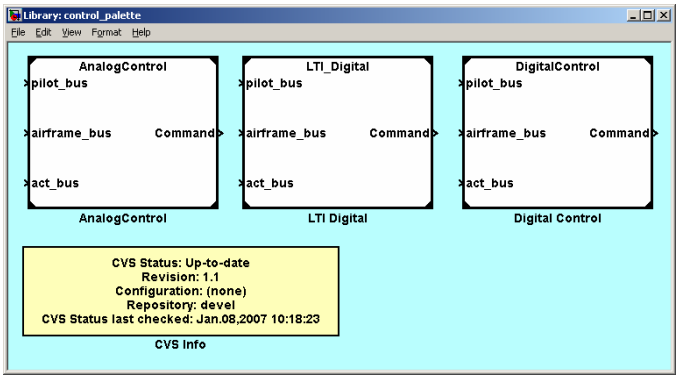


**Figure 5:** Revision controlled "palette" using model reference.

## MODELS, DATA, AND ARTIFACTS

There are many different answers to the question “What do we need to store within a revision control system?” The minimum requirement might be to store only the files required to ensure that a new user getting only these files from revision control would have a “fully functional” model. The definition of “fully functional” might change for a model as it progresses through a typical Model-Based Design process. Initially it might be sufficient for the model to simulate. Later it might require additional files or libraries to support code generation.

To guarantee reproducibility at some point in the future, or for future root cause analysis or similar activities, it is common to additionally store “derived files” at specific points within the development process, such as the generated code, simulation or model validation results, linearized approximations to a non-linear model, etc.

There can be additional benefits to storing files that could, in principal, be entirely reproduced. For example, when using Model Reference it can be useful add the simulation target MEX to the configuration. When another user gets the set of files from the configuration management system they will not have to wait for the MEX files to be rebuild. We note that if this is done then a straightforward mechanism must be provided to allow a user to ensure that these files can be made writable, and hence rebuilt, if required. This is typical of the type of trade-off that must be made between the often conflicting demands of the complexity of a configuration management system, the level of reproducibility it offers, and the opportunity for additional benefits.

Other derivable files (“process artifacts”) relating to the process, rather than the primary use of the models, that might be stored include model checking reports used to demonstrate review readiness, the code generation logs, coverage reports, etc. The decision about what files to store at what stages of the process will depend upon the working practices of individual groups, companies and industries.

To illustrate this concept, we use an MD5 checksum algorithm. By generating checksums automatically before and/or after key steps, we can make this information more trustworthy. For example, prior to creating a release or exporting a configuration we might create a manifest file, such as the one shown in Figure 6.

```
Exported on Sep.13,2006 14:41:57

Filename:                Rev.:     Checksum:
.sandboxroot             1.1.1.1   1e4f2a5e29cfc2806e7f758e99cf6a4b
EngineModel.mdl          1.3       afd710f2a2bdf3b32c046f1e539e9c88
m20060908T182816.mdl     1.3       2a36b6903e84662a65b8eb54688a6e56
transmission.mdl         1.1       9202b54deb6642f93e95cdb3fcb61c91
transmissionRatio.mdl    1.2       69949fe144b364035309c546c4f92531

(C)The MathWorks Inc. 2006
```

**Figure 6:** Checksum Manifest File

MD5 checksums can also be included, straightforwardly, into model advisor reports, test reports, coverage reports, etc., and even into the generated code and header files (using a Real-Time Workshop® Embedded Coder source code template). In this way, the traceability of the files throughout an entire Model-Based Design development process can be greatly enhanced.[2]

### META DATA

In addition to the files we store, it is useful to also store "meta data" with the configuration. Such information is particularly useful for deciding which configuration to get from the repository without actually having to get it. Most revision control software automatically stores various items of metadata, such as who last modified a file, when they did it, and perhaps a comment about what they did.

For a configuration management system to be relevant to Model-Based Design we typically store more information than this. What is stored depends upon the task the models are being used for, the stage in the development cycle, and can vary from company to company. Some examples include:

- Project status (e.g. "current working", "release candidate", "delivered");
- Application suitability ("experimental", "robustness studies", "fixed-step solvers", "linearization", "Rapid-Prototyping", "Production Code", etc.);
- Classification ("internal use only", "suitable for export", or "imported from customer");
- The version of The MathWorks tools used to develop these files.

## COMMON TASKS

A number of common tasks can be greatly enhanced by making use of the additional information available when a configuration management system is in use. We describe some of the more common tasks.

### CREATING ADDITIONAL INFORMATION FOR THE AUDIT TRAIL: CHECKSUMS

It can be useful to add additional information to the configuration to give future users confidence that they have the same instances of the files as those that were checked in.

### CREATING VARIANTS

A common workflow, especially in the automotive world, is to take a baseline configuration of a system, and to then create variants of it by swapping components for alternatives that implement the same type of functionality, but that are suitable for a different application.

Consider the example of a vehicle engine control unit (ECU). One of the components of the ECU is the "idle-speed control" component. Suppose that the baseline configuration of the ECU is suitable for a vehicle with automatic transmission and a normally aspirated engine. A later model of the vehicle has a manual transmission and a turbo-charged engine. To configure the ECU to be suitable for this new vehicle we might need to exchange the idle-speed control component for one that is suited to this new vehicle. We can do this by deleting the Simulink subsystem that implements the baseline idle-speed control and replacing it with a more suitable implementation. In doing so, we have created a new variant of the original baseline configuration.

In reality, many components in the ECU would have to be swapped to make the ECU suitable for this new vehicle, and the creation of a variant may require a large number of components to be swapped.

One way to enable the creation of variants within the framework described in this paper is as follows:

The baseline model of the ECU is created. The various alternative implementations of the idle-speed control unit are created and tested. All the models are stored within the same project. All the different idle-speed control models are placed in a revision controlled palette, which is also placed in the same project. The entire project is marked within the configuration management system as the baseline configuration.

When a new team member creates a local copy of the project files to edit, often called a sandbox, from the baseline configuration, he or she has all the files need to create new variations. This user can check-out the ECU model, replace the idle-speed control unit with one of the alternatives from the revision controlled palette, and hence create a new variation of the original model.

In a fuller implementation there might be many such revision controlled palettes.

Part of the motivation of using a configuration management system is to be able to trace the development of a configuration, and to be able to reproduce its outputs, whether it is generated code, simulation results, linearized models, etc.

## PARALLEL DEVELOPMENT OF LARGE MODELS

It is possible to perform a top-down design within Simulink. This workflow fits well with the Model-Based Design philosophy of starting with simple models of the parts of a design, and elaborating them with more detail as the design evolves. This section suggests some practical guidelines for this process.

### DEFINING THE INTERFACES

Defining the interface of a software component, whether it is a C or M-code function, or a Simulink subsystem, is a key first step for others being able to use it. There are a number of reasons for this:

- Agreeing on interfaces is an important first step in deciding how to break down the functionality of a large system into subcomponents. For example, if every output from a component is an electrical current apart from one signal, which is a pressure signal, then this occurrence may suggest that the component has missing functionality, such as a model of a sensor that converts pressure to current, or perhaps should not generate the pressure signal at all.
- Once component interfaces have been agreed upon they can be developed in parallel. If the interface remains stable then integrating those components into a larger system is straightforward.
- Changing the interface between components is expensive. To do so requires changes to at least two components (the source and receivers), and to their test harnesses. An interface change also makes all previous revisions of those components incompatible with the current and future configurations.

However, the above discussion is not meant to imply that interfaces should be set in stone. If an interface chance is required to support new design requirements, then the Configuration Management system can be used to support the interface change and minimize the impact on other team members.

Consider “Component A”, revision 1, used in Project 1 and Project 2. An engineer working on Project 1 changes the interface of Component A to make it compatible with Component B, used only in Project 1.

**Without CM:** The engineer from Project 1 checks the updated Component A, now at revision 2, back into revision control. The next time that the engineers on Project 2 perform a “get updates from revision control” operation, they find that they have an incompatible version of Component A.

**With CM:** The Engineer from Project 1 behaves as before, but additionally creates a new *configuration* of Project 1. Engineers on Project 2 check only for new configurations of Project 2. They may be aware of the new revision of Component A, but until an engineer from Project 2 integrates it into their project and creates a new configuration of Project 2, they will not get the new revision.

### GUIDELINES FOR DEFINING THE INTERFACES

We offer the following suggestions for defining the interfaces of subsystems for a new project:

- Base the boundaries of the subsystems on those of the real systems. This guideline is especially useful if a model contains both physical plant and control system elements, and control elements run at different rates.
- Keep model elaboration in mind. For example, if the development process requires the addition of sensor models at a future step, one could start with an empty subsystem that either passes signals straight through, or performs a unit and/or name conversion so that the sensor interfaces are captured.
- Review the potential reuse of the component in support of the current design and future designs – some elements will have more ubiquitous reuse such as “sleep state” feature.
- Define and use a signal naming convention.

Within Simulink, the process of defining interfaces is aided by appropriate use of Simulink Signal Buses, as discussed in the Simulink documentation. [3]

### DEFINING THE MODEL STRUCTURE

As far as possible, try to avoid a mix of implementation detail and subsystems or components in the same level

of a model. This philosophy makes testing, and further subdivision into subcomponents, straightforward.

Getting the right level of granularity for the components of a model is important. We offer the following guidelines:

- Pick granularity so that only one engineer is likely to need to edit each model at a time.
- For interface definition, consider the suggestions highlighted in the previous section.
- Group by rate of update.
- No decision should be set in stone. Components can and should be subdivided as they increase in size and complexity.

A BRIEF NOTE ON THE PARAMETERIZATION OF COMPONENTS IN SIMULINK

In the following sections we discuss the various methods for partitioning a Simulink model into a number of components. We do not discuss the various methods of partitioning the data that each of these components requires. This topic could easily be the subject of a paper in its own right. Instead, we highlight the following points.

Global Parameters

A common approach in the automotive industry is to completely separate the problem of parameter storage from model storage. In this scenario, the parameters for a model come from a database of calibration data, and the specific calibration file used becomes part of the configuration. The calibration data is treated as global data, and resides in the base MATLAB workspace.

Non-Global Parameters

In this scenario, each component stores and loads its own parameters. Various methods exist for storing such local parameter data, and include:

- Mask Workspaces, with or without the use of Mask Initialization functions
- With Model Reference, Model Workspaces
- Using parameter files (.m or .mat) and callbacks of the individual Simulink models (e.g., preload function)

However, combining a number of components that store their own parameter data runs the risk of parameter name collisions. Specifically, if a naming convention for parameters, or alternatively a data dictionary of unique parameter names and definitions, is not used then there is the risk that two components will use a parameter having the same name but with different meanings.

See the Simulink documentation [3] for more information on these features. Finally, we note that in Release 2006a from The MathWorks it is possible to control the scope of data for a given subsystem via the SubSystem Parameters, Permit Hierarchical Resolution dialog.

DEFINING COMPONENT INTERFACES

There are various ways in which signals can be passed between different Simulink subsystems or components. The choice of which method is the most appropriate will vary from company to company, and project to project. Picking the most suitable method requires trading-off readability, robustness, and flexibility. We offer the following suggestions.

Signal Buses

Signal Buses have been a standard feature of Simulink for many releases. For more information on Signal Buses, see the Simulink documentation.[3] Signal Buses, especially when used in conjunction with Bus Objects, offer built-in interface checking. Signal Buses are particularly well-suited for use at the high levels of models, where components often either

- Have a very large number of signals going into, and out of them; or
- Do not use all the signals available.

An additional benefit of signal buses is that, should a change in the interface definition be required, the change can be performed by modifying the Bus Object. As a result, one avoids the need to make structural changes such as the addition or removal of ports and signal lines to the model.

Inports and Outports for Each Signal

At the lower levels of a model, where the components are specifications or implementations of algorithms the use of individual inports and outports can improve readability compared to signal buses.

We note that using ports to create interfaces has a greater risk of connection problems, because it is difficult to check the validity of connections, other than their data type, size, etc.

## CONCLUSIONS

In this paper, the authors have illustrated a number of practical approaches to performing configuration management within Model-Based Design. We have discussed the different ways in which the files associated with Model-Based Design can be linked to third-party configuration management and revision control packages, and the mechanisms within MathWorks tools that allow large projects to be split into a number of files suitable for use with such third-party packages. Starting with features that are applicable to a single user, we have discussed features and workflows appropriate for small teams, through to those which are

appropriate for larger teams working in parallel on a project.

## REFERENCES


1. MATLAB Users Guide - http://www.mathworks.com/access/helpdesk/help/techdoc/

2. "Using MathWorks Tools to Generate Code for DO-178B Applications," Bill Potter, The MathWorks, presented at The MathWorks Aerospace and Defense Conference 2006.

3. Simulink Users Guide - http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/