# Near Automatic Translation of Autonomie-Based Power Train Architectures for Multi-Physics Simulations Using High Performance Computing

**Tomasz A. Haupt, Angela E. Card, Matthew Doude, Melissa Hannis, Michael S. Mazzola**
Mississippi State University, USA

**Scott Shurin**
US Army TARDEC, USA

**Christopher Goodin**
US Army ERDC, USA

haupt@cavs.msstate.edu

## *ABSTRACT*

*The Powertrain Analysis and Computational Environment (PACE) is a powertrain simulation tool that provides an advanced behavioral modeling capability for the powertrain subsystems of conventional or hybrid-electric vehicles. Due to its origins in Argonne National Lab's Autonomie, PACE benefits from the reputation of Autonomie as a validated modeling tool capable of simulating the advanced hardware and control features of modern vehicle powertrains. However, unlike Autonomie that is developed and executed in Mathwork's MATLAB/Simulink environment, PACE is developed in C++ and is targeted for High-Performance Computing (HPC) platforms. Indeed, PACE is used as one of several actors within a comprehensive ground vehicle co-simulation system (CRES-GV MERCURY): during a single MERCURY run, thousands of concurrent PACE instances interact with other high-performance, distributed MERCURY components. A proof-of-concept implementation of PACE, as applied to a conventional powertrain architecture, was presented at the SAE2016 conference. Since then, a C++ library of components implementing the functionality of the corresponding Simulink subsystems has been developed, followed by streamlining the process of the generation of the C++ code for a particular powertrain; the native Simulink XML representation of the architecture (components and their connectivity) is used for an automatic generation of the simulation workflow and thus effectively reproducing the functionality of the Autonomie models while making it compliant to MERCURY requirements. The resulting PACE models are rigorously verified and validated against results generated by Simulink runs. Furthermore, the whole process is largely automated, thus providing time savings when implementing new vehicle architectures.*

## 1.0 INTRODUCTION

The need for the simulation of military vehicles is important in this time of advanced technological development. There are many ground vehicle architecture ideas that might prove useful in the conduct of military missions. However, the cost of creating these vehicles without adequate assurance that they would be successful is a challenge to be overcome with the help of new computer-aided design tools. Powertrain Analysis and Computational Environment (PACE) is part of the step toward quickly creating multi-physics ground vehicle simulations ready for High Performance Computing (HPC) platforms, such as multi-cluster computers.

MERCURY is a high performance computing simulation system that is designed to simulate military ground vehicles for mobility. There are several components that make up the MERCURY cooperative simulation

framework. PACE is used as one of several actors within a comprehensive ground vehicle co-simulation system (CRES-GV MERCURY). During a single MERCURY run, thousands of PACE instances interact with other high-performance, distributed MERCURY components; an example of which is shown in Figure 1-1.

PACE is a powertrain simulation tool. A proof-of-concept implementation of PACE was presented at the SAE2016 conference [1]. This powertrain simulation tool uses advanced behavioral modeling for the powertrain subsystems of conventional or hybrid-electric vehicles. Since the original report, a number of significant improvements have been made in both the computer science of code generation and the breadth of powertrain modeling. A new work flow has been implemented to nearly automatically create HPC ready C++ code from starting-point Simulink-based powertrain simulations. In the past, all of the Simulink models were derived from Autonomie models. But support for independent model components was always a goal of the PACE project and this is demonstrated by the addition of new capabilities not included in Autonomie, such as modeling of thermal components in the powertrain. Both new developments are discussed in the next section.

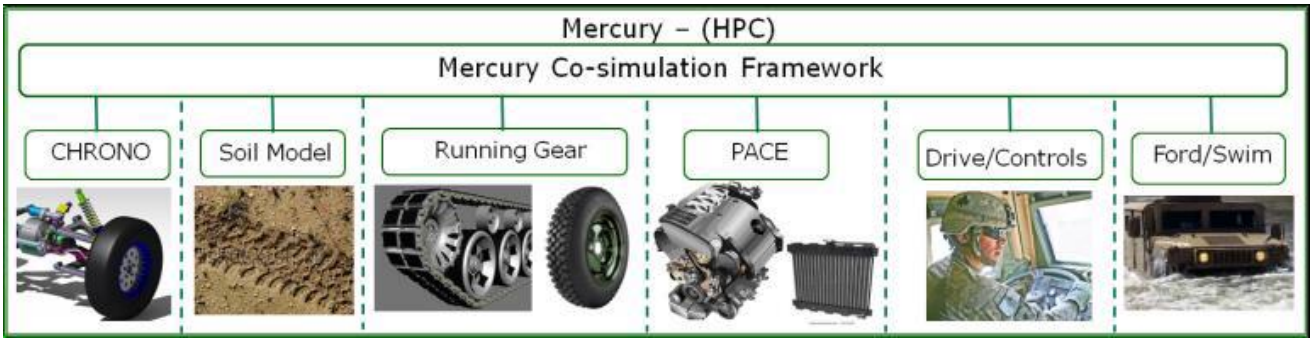


**Figure 1-1: The PACE Federate as an actor in MERCURY [1]**

## 2.0 SIMULINK MODELS

Autonomie is a forward-looking, robust simulation tool for the analysis of a vehicle's performance [2]. Autonomie is used to simulate realistic vehicle performance. It provides validated signal processing analysis, which allows the powertrain engineers to have confidence in the vehicle model.

PACE is a C++ implementation of a vehicle architecture designed by MSU powertrain engineers. Currently, the specifications for a military vehicle architecture are provided to MSU, whose powertrain experts then utilize a work flow that results in a Simulink representation. One method for doing this is Autonomie because it reduces the powertrain simulation to a Simulink representation [3] which is the starting point for creating PACE C++ code using the new auto-code generation workflow. However, any Simulink representation of a powertrain that uses modeling blocks supported by the PACE workflow can be used to create the operative HPC ready C++ code. In other words, PACE is a powerful tool for creating HPC ready code for the MERCURY environment using a potentially wide variety of independent third party Simulink-based simulations.

For example, if the source of a powertrain simulation is Autonomie, it generates MATLAB/Simulink models for the vehicle, which are constructed in a hierarchical format and saved as an SLX file. The top level of a vehicle model's hierarchy is shown in Figure 2. After the creation of the vehicle model, Autonomie is no longer utilized in the PACE implementation. Instead, MATLAB is used next to edit and verify the SLX file.

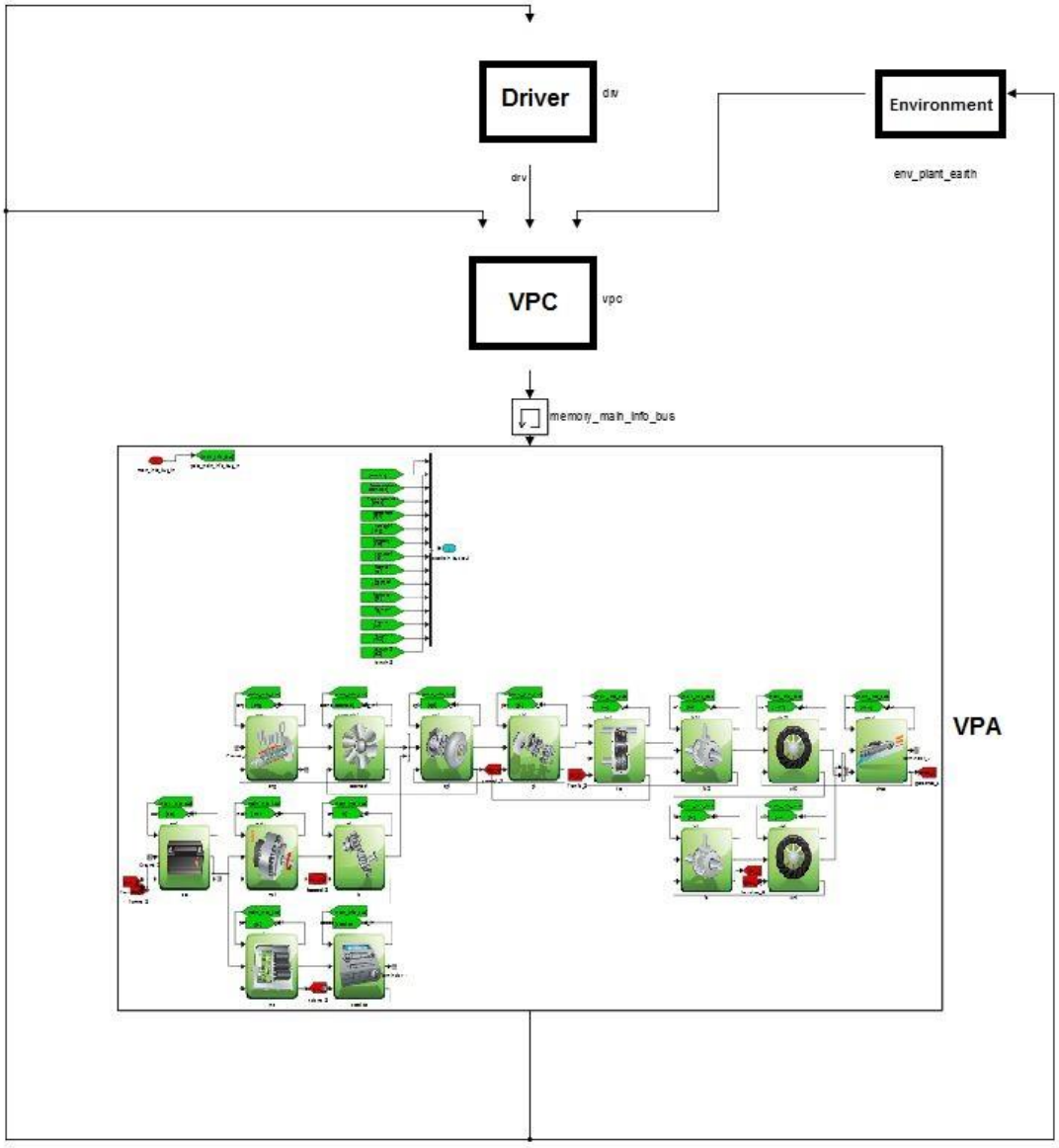


**Figure 2-1: Image of the top level of the Autonomie vehicle model.**

In Figure 2-1, the top level of the vehicle model is shown to have a driver component, an environment component, a vehicle propulsion controller (VPC), and a vehicle powertrain architecture (VPA). Certain aspects of the VPC and the VPA are supported in PACE. The purpose of PACE is to be an actor within MERCURY. Some of the vehicle components implemented in Fig. 2-1 have higher fidelity components in MERCURY. For this reason, some of the elements of the vehicle model in Fig. 2-1 are ignored and internal control connections implemented differently in PACE. This is automatic and transparent to the powertrain modeling engineer.

The components on the second level of the VPA module are the engine, the gearbox, the motor, etc. Within each of the components on the second level of the vehicle powertrain architecture are sub-hierarchies that make up the logical aspects of those components. These aspects always include a plant module, and optional controller

module(s) within a VPA component's subsystem. Each module level is comprised of three blocks: an input block, a logic block, and an output block. An example is illustrated in Figure 2-2. The first block is the input block that connects the input to this module from the other components in the system. The middle block is the powertrain logic module that houses the functionality of the module. The last block is the output block that connects the output of this block to the rest of the system.

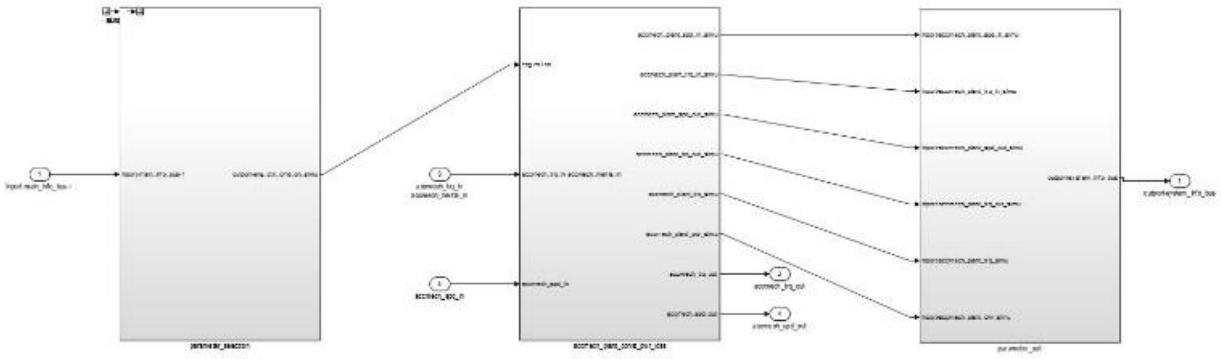

**Figure 2-2: Image of the vehicle model VPA component's plant module**

The significance of editing an SLX file in MATLAB/Simulink is that this allows the independent development of new modules. This means that third party modules can be incorporated into a vehicle model originally derived in part from an Autonomie vehicle model, which results in more functionality to the overall vehicle powertrain simulation.

For example, an independently developed thermal model is reported in a companion paper [4] that has been implemented within a VPA component. The newly incorporated thermal model was developed in order to create temperature constraints on the engine and other electrical components. At each time step, the engine's temperature is checked and if the engine reaches a predetermined set point, a cooling agent is triggered that will bring the temperature of the engine back to an acceptable thermal state. On a separate logical loop in the thermal model, the electric components are also examined each time step such that when the electrical components reach a certain thermal threshold the cooling agent is triggered to bring the component back to an acceptable temperature. The loading on the powertrain from the thermal system is self-consistently calculated.

## 3.0 PACE IMPLMENTATION

PACE provides C++ implementation of powertrain models developed in Autonomie thus making these models available on HPC platforms as components of a large-scale distributed simulation known as MERCURY. The primary concern is to automate the process of building an equivalent C++ code while preserving the model predictions. The latter is verified by direct comparison of output generated by the model as run in Autonomie and stand-alone C++ code. The steps in generating equivalent C++ code are summarized in Figure 3-1.

Our approach is based on the fact that Autonomie is a Matlab/Simulink application, and in particular, that the models created using Autonomie GUI are run as standard Simulink applications. Consequently, our process starts with running Autonomie and letting it generate the Simulink model and populate the Simulink workspace with the values of model-dependent constants and initial values of variables. This results in two files: Simulink

XLS file that defines the model, and initData.mat that stores the workspace variables in a native Matlab format.

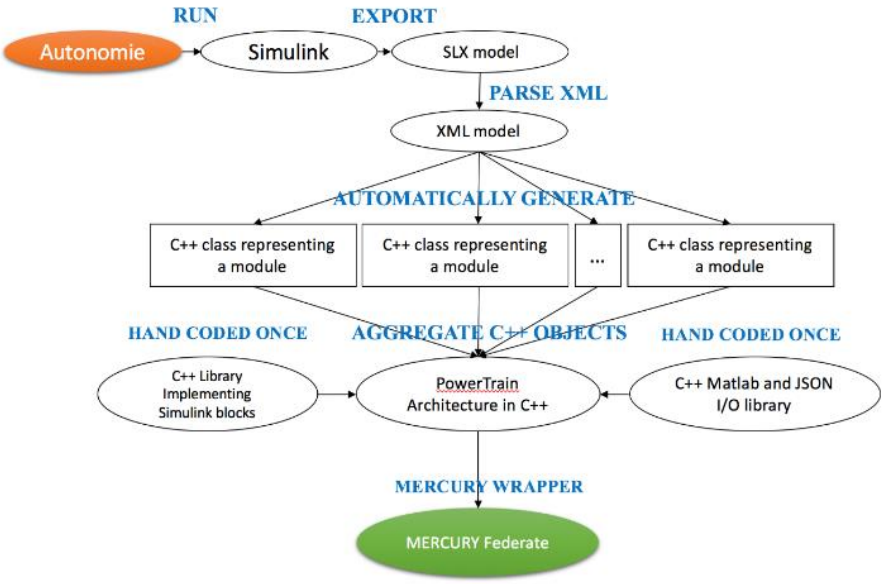


**Figure 3-1: Workflow for generating C++ code based upon a powertrain model developed in Autonomie.**

The information embedded in SLX is all that is needed to restore the Simulink model, including the model architecture, that is, the list of Simulink blocks used by the model, their connectivity, and their parameters, all captured in XML format. The XML document is organized as a tree. It is hierarchically composed of subsystems and basic ("atomic") blocks available from Simulink Library. For example, as shown in Figure 2-1, the root subsystem comprises the highest-level components (each defined as a subsystem): Driver, Environment, VPC, and VPA, as well as a base "Memory" block. Each of these subsystems are further composed of subsystems and base blocks, until all subsystems contain exclusively base blocks (i.e., the "leaves of the tree").

The goal of PACE implementation is not to reproduce a complete Simulink system as generated by Autonomie. Instead, the implementation strategy is to provide a library of C++ classes, with each class modeling a single powertrain component. Selected components are then aggregated and orchestrated by the main PACE driver routine to provide functionality needed by MERCURY. This strategy provides necessary flexibility for the PACE implementation as MERCURY provides its own implementation of some components embedded in the Autonomie model, such as Driver, Wheels, and others (c.f. Figure 1-1). Accordingly, the XML tree representing the complete Autonomie model is split into branches, and only branches describing a single component (such as engine plant or gearbox controller) are further processed, one branch at a time. The root subsystem of each branch is implemented as a C++ class, with all its subsystems implemented as the class' methods.

Implementation of most basic blocks (e.g., lookup table, integrator, transfer function, etc.) is done manually prior to automatic generation of the rest of PACE code described below. The development of the equivalents of basic Simulink blocks is guided by the Simulink documentation and general knowledge of numerical methods. The resulting C++ code is packaged as a library, referred to as libsim.

Each component (that is, a branch) is a subsystem and as such is comprised of Simulink blocks. Each block is described by a list of parameters that specify the block type (a subsystem or a basic block; in the latter case the type is a reference to the Simulink Library), its input and output ports, and block-specific properties, such as initial values or intended behavior, if different than the default. Then the connectivity of the blocks is described by providing links between input and output ports of the subsystem blocks. Finally, subsystem mask parameters are given, if any.

The processing of each branch starts with identification of the constituent blocks and creating lists of its input ports (arguments), and output ports (return values). Some basic blocks, such as "From" and "GoTo" blocks are eliminated, and the connectivity table is adjusted accordingly. The remaining blocks are used to create a list of executable commands using a simplified syntax referred to as a pseudo-code. A "Constant" block becomes an assignment. Some blocks representing very simple operations such as "Sum" or "Gain" are inlined, that is, the corresponding expressions are encoded in the pseudo-code (taking into account that parameters of the Sum block may change addition into subtraction, etc.). "Mux" and "Demux" functionality is inlined as well. The "Stop" block is expressed as a runtime exception that can be caught by PACE or MERCURY as needed (it may represent the situation when the engine stops because of lack of gas). The "Memory" block is encoded as two pseudo-code statements, one for setting the memory value, and the other for retrieving the memory value. The remaining blocks are encoded as function calls, either to the library of basic blocks (described above) or the class' methods resulting from processing subsystems of the branch.

A critical element of the code generation is the analysis of the data dependencies: in general, the blocks cannot be executed in the order they are listed in the XML file because of apparent algebraic loops resulting from feedback. These loops are typically resolved in Simulink by memory blocks, or initial values of some blocks (such as integrators). A "greedy" algorithm is successfully employed in the PACE code generator. This implies that at each step of the computation, all blocks are executed (in arbitrary order) for which the values of all input arguments are known. With each such step, new results are produced, i.e., inputs to more blocks, thus enabling execution of other blocks, until all blocks of the system are executed. Another outcome of the data dependency analysis is the identification of state variables, that is, variables for which the values must be preserved between invocations.

Once the pseudo-code of the branch's root subsystem is created, the same procedure is applied recursively to all its subsystems generating the method of the C++ representing the branch, that is, the complete pseudo-code for the powertrain component.

The next step in the automatic code generation is the rewriting of the pseudo-code using C++ syntax. Among the challenges of this task are determination of the types of variables (C++, unlike Matlab, is a strongly typed language) and the accessing of the Simulink's workspace variables.

The types of variables are determined by their backward propagation: the type of a variable returned by a block determines the type of the variable assigned to it. We know the type of all output ports of a basic block (usually a double or a vector of doubles). If a block (a basic block or a subsystem) has more than one output port, a C++ structure is returned with each filed corresponding to one output port (with a known type). This way the types of all variables can be determined by traversing the tree from its leaves to the root.

The values of the workspace variables (or constants) come from the initData.mat file (generated by running Autonomie, as described above). In order to access these values, PACE includes another library ("MSU_IO") developed "off-line". The functions of that library allow the values of a workspace variable to be retrieved by the name of the variable, and, in the case of vectors and 2D- and 3D-Matrics, it is formatted according to the C++

standards. At the code generation phase, the references to the workspace variables are detected (their names are distinct from names of mask variables), and appropriate calls to the library are inserted.

Reading the workspace variables happens at runtime (thus not during the generation of the code) during the initialization of PACE objects, just after these objects are constructed. To make PACE completely independent form any Matlab/Simulink software, optionally, the initData.mat is translated into JSON format, and read by PACE using a JSON version of the MSU_IO library.

The final step in the code generation is the creation of the class representing a powertrain architecture, a MERCURY federate. This class instantiates all modules that belong to the requested architecture and orchestrate their execution as requested by MERCURY's execution controller. Figure 5 illustrates the PACE interaction with one of the other Mercury federates, CHRONO, that is logically and physically in line with the powertrain because CHRONO calculates wheel dynamics with the ground.

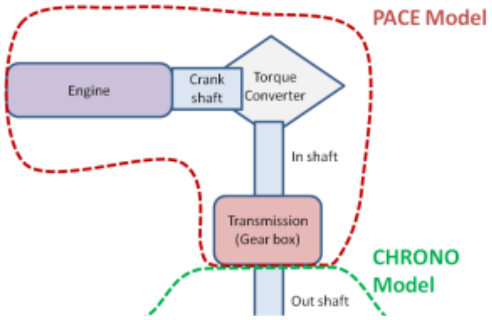


**Figure 3-2: The boundary between PACE and CHRONO defined on the powertrain topology**

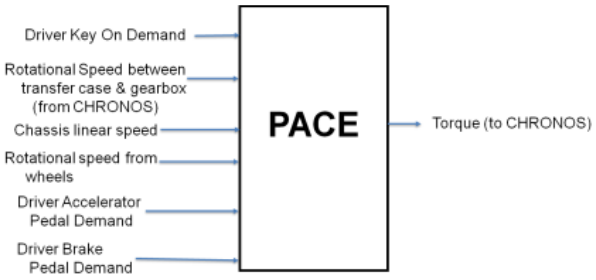


**Figure 3-3: The input/output variables for PACE as a federate within Mercury**

The current version of the PACE code supports two Autonomie powertrain architectures, one of which is a mild parallel hybrid powertrain, while the second is a series hybrid powertrain. Both are consistent with vehicles of interest to the U.S. Army. Figures 3-2 and 3-3 relate to the mild parallel hybrid vehicle powertrain architecture for PACE. Figure 3-2 shows the specific integration boundary between the PACE output, which is the transmission output shaft, and the next federate called CHRONO which computes wheel and suspension dynamics. Figure 3-3 shows the inputs taken by PACE from other federates and the one output that PACE passes to CHRONO, which is the transmission output shaft torque. The integration boundary between PACE and CHRONO is different for the series hybrid architecture. This division does not exist in the corresponding Autonomie simulation and represents a significant accomplishment in the computer science of PACE and a significant accomplishment in the improvement of the fidelity of design studies of ground vehicle mobility that MERCURY represents.

## 4.0 PACE VERIFICATION

PACE undergoes a verification process that provides confirmation to the user that PACE will reproduce a similar output as the MATLAB/Simulink model it replicates. The inputs and outputs of each signal from the modules within the vehicle model are collected from the MATLAB workspace. To enable verification, the vehicle model was instrumented in MATLAB/Simulink by placing "ToWorkspace" blocks at every input signal

and output signal. After running the model simulation in MATLAB/Simulink on a specified drive cycle, the values from that run can be found located in the Matlab workspace. Matlab scripts were created to capture this data. PACE was designed to save corresponding data from the execution of the compiled C++ code as comma separated values (CSV) files.

In the previous version of PACE, the verification process was time consuming and largely qualitative. All signals to and from each module in the PACE C++ data were individually evaluated against that of the MATLAB/Simulink data. These signals were evaluated by comparing the output CSV files from PACE to the output CSV files from MATLAB.

The output files from MATLAB and PACE were then compared by plotting graphs of the data. One signal in a CSV file was a column of data. The data from PACE and Simulink were then plotted against time.

These plots would overlay point by point if the data from PACE matched that of the MATLAB/Simulink output. This was a tedious process and did not compute statistically significant estimates of the correspondence between the data sets.

Figure 4-1 shows three plots of one signal. The top image is both sets of data compared with the Simulink data overlaid on top of the PACE data. The bottom left image (blue plot) in Figure 4-1 is a signal from the PACE output. The bottom right image (orange plot) in Figure 4-1 is a signal from the MATLAB output data. These signals are compared by looking at the side-by-side plots and by placing one plot over the other. There are data points that do not match in this picture, but it is difficult to identify them and by how much they differ.

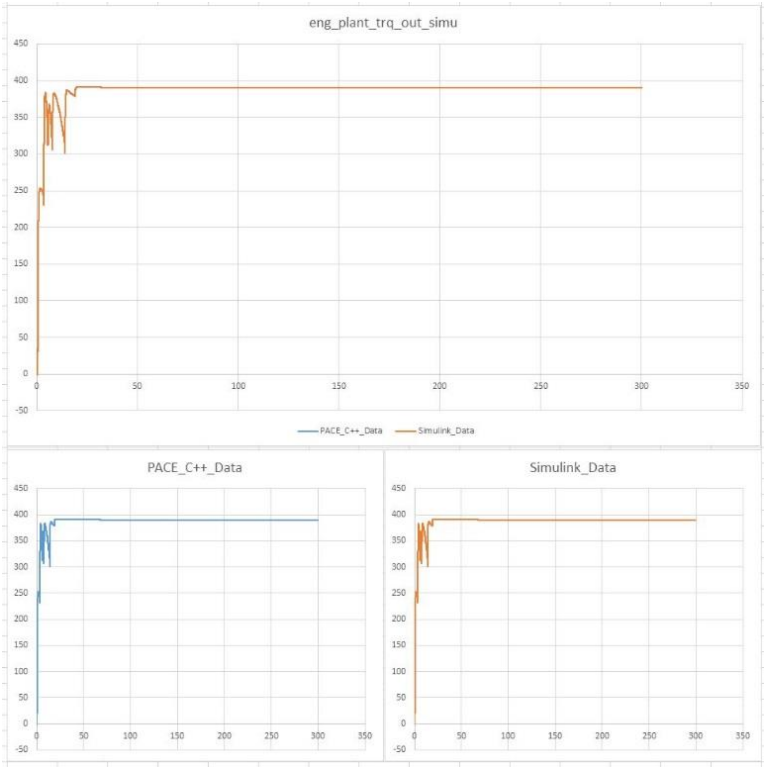


**Figure 4-1: Top plot is a representative overlay of one time series of MATLAB vs. PACE data. The**

**bottom left plot is the PACE output data. The bottom right plot is the MATLAB output data.**

In the previous version of PACE, Figure 4-1 would have been the basis for validation. Each signal would have been validated if no obvious discrepancies were seen. The current implementation of PACE is not considered validated until the data has gone through an additional level of statistical analysis in order to better analyze the potential differences between that of the PACE data and the corresponding Simulink. Figure 4-2 illustrates a scatter plot of the data of a particular module from PACE and Simulink. Before creating the scatter plot, each data point is compared and only data points not found to be equal to the fourth decimal place are plotted. Each blue circle represents a C++ value vs. a Simulink value at a single time step. The data points are expected to lie on the line where the slope is equal to one and the intercept is equal to zero. Discrepancies are easily seen when the circles do not align on the slope intercept. Outliers, if any, are obvious and usually indicate bugs in the PACE code. Slight deviations in the slope or the intercept could indicate that there is a systematic error in a contributing algorithm. In either case, if the points do not align on the slope or intercept, there is a problem, and the PACE code is returned to be improved. Once each signal has passed evaluation by this scatter plot test, a more rigorous statistical evaluation of the remaining differences between the PACE and the Simulink data is produced.

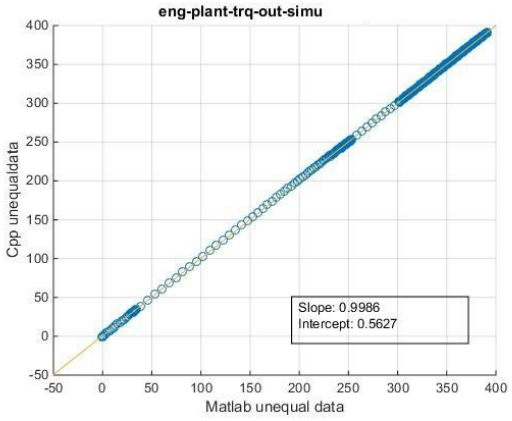


**Figure 4-2: A scatterplot of filtered PACE vs. Simulink data.**

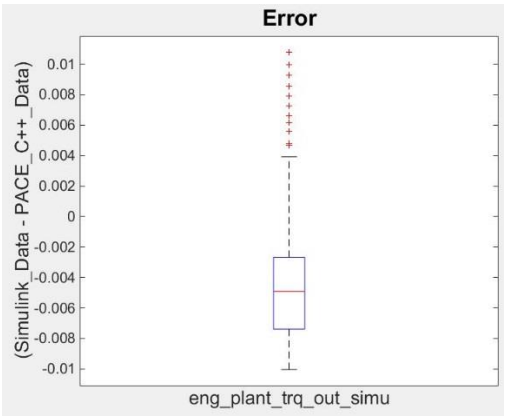


**Figure 4-3: A boxplot of the distribution of error of PACE results against Simulink results for a single signal.**

This final analysis utilizes the MATLAB boxplot function [5]. In each boxplot, the distribution of error is provided for all the individual signals in a plot of a single module. Figure 4-3 illustrates an image of a boxplot of a single signal.

In the figure, the rectangular box in the plot represents a range of data. The Simulink data is subtracted from the PACE C++ data and that output is stored in an array. The line in the middle of the box is the median, also called the second quartile (Q2). This boxplot shows the distribution of error in the sample of data. The top of the box is the third quartile (Q3). The bottom of the box is the first quartile (Q1). The vertical line extending out from each end of the box are called whiskers. The whisker's length at most is 1.5 times the range (75% Quartile - 25% Quartile). Outside this range are the outliers, which are depicted as red plus symbols.

Figure 4-4 depicts a boxplot that has the plots of all the signals of a module in one graph. The range on the y-axis has a significantly small distribution of error compared to the full dynamic range of the data. Therefore, the

plotted signals for the module in Figure 4-4 are considered valid.

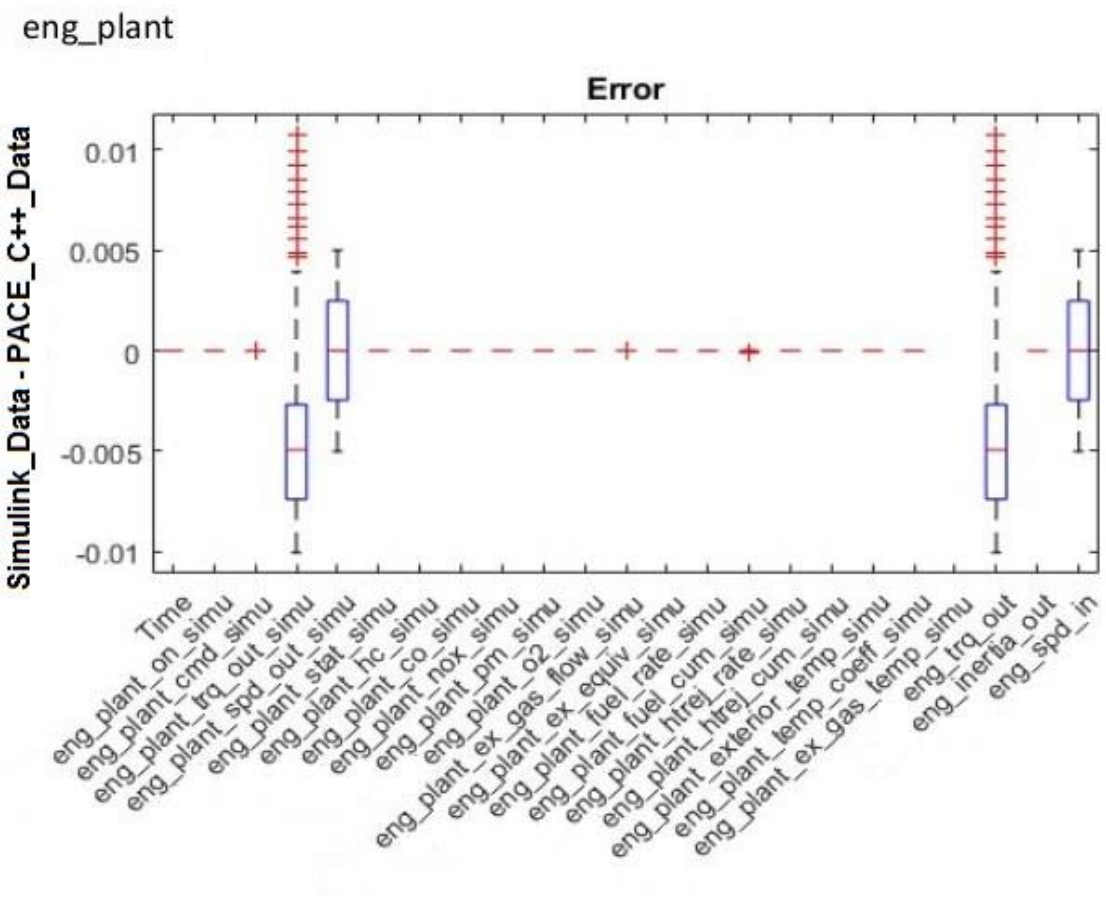


**Figure 4-4: A boxplot of the distribution of error between PACE data and Simulink data for all the signals in a single module.**

## 5.0 SUMMARY AND CONCLUSION

Model based design and high performance computing can be combined to produce high fidelity large-permutation-space concept design exploration for future military ground vehicles. PACE is an important development in addressing one of the major challenges to using HPC for this application, which is the creation of HPC ready source code from high fidelity powertrain models existing in engineering computing environments. This paper reports a major improvement to the PACE workflow in that a nearly automatic process for creating a federate of MERCURY coded in C++ starting from the MATLAB/Simulink engineering friendly simulation environment. In the work presented in [1], it took months to generate this code for a single Autonomie powertrain architecture. With the work flow now available this time was reduced to hours for the same powertrain architecture. This accomplishment opens the possibility of having many permutations of

powertrains to run with MERCURY in the future.

## REFERENCES

[1] T. Haupt, G. Henley, A. Card, M. Doude, M. Mazzola, S. Shurin, A. Hufnagel and C. Goodin, "Powertrain Analysis and Computational Environment (PACE) for Multi-physics Simulations using High Performance Computing," in *SAE International*, Starkville, MS, 2016.

[2] "Autonomie - Vehicle Modeling Approaches," Argonne National Labs, [Online]. Available: http://www.autonomie.net/references/vehicle_mods_25.html. [Accessed March 1, 2017].

[3] "Autonomie," Argonne National Labs, [Online]. Available: http://www.autonomie.net/expertise/Autonomie.html. [Accessed March 1, 2017].

[4] G. J. Monroe, M. Doude, T. Haupt, G. Henley, A. Card, M. Mazzola, S. Shurin, A. Hufnagel and C. Goodin, "Thermal Modeling in Powertrain Analysis and Computational Environment (PACE) for Additional Behavioral Modeling Capability in Autonomie," in SAE International Technical Quality Response Team, Starkville, MS, 2017.

[5] Mathworks documentation, [Online]. Avaiable: https://www.mathworks.com/help/stats/boxplot.html [Accessed March 1, 2017]

## Additional Information

### Contact Information

The corresponding author is Tomasz Haupt who may be contacted at haupt@cavs.msstate.edu.

### Acknowledgments

Material presented in this paper is a product of the CREATE-GV Element of the Computational Research and Engineering Acquisition Tools and Environments (CREATE) Program sponsored by the U.S. Department of Defense HPC Modernization Program Office. This effort was sponsored under contract number W912HZ-13-0037. Public Release, Distribution Unlimited.