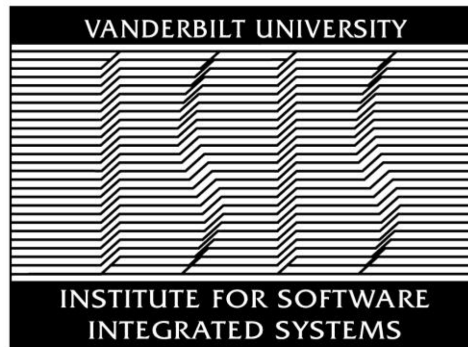

Institute for Software-Integrated Systems

Technical Report

TR#: ISIS-15-110

Title: Integration Platform Technology Components in the META Toolchain

Authors: Zsolt Lattmann, James Klingler, Patrik Meijer, Ted Bapty, Sandeep Neema and Jason Scott

This research is supported by the Defense Advanced Research Project Agency (DARPA)'s AVM META program under award #HR0011-13-C-0041.

Copyright (C) ISIS/Vanderbilt University, 2015

Table of Contents

List of Figures iv

List of Tables vi

1.0 Job Manager and Remote Execution 1

1.1. Summary 1

1.2. Objective 1

1.3. Architecture/Data flow 2

1.4. Detailed Description 2

1.5. Future Enhancements 6

1.6. AVM Involvement 6

1.7. Conclusions 6

2.0 Project Analyzer/Dashboard 6

2.1 Summary 6

2.2 Objective 7

2.3 Architecture 8

2.4 Data flow 9

2.5 Detailed Description 9

2.6 Future Enhancements 12

2.7 AVM Involvement 12

2.8 Conclusions 13

3.0 Parametric Tool Exploration 13

3.1 Summary 13

3.2 Objective 14

3.3 Architecture 14

3.4 Data flow 14

3.5 Detailed Description 15

3.6 Validation 19

3.7 Future Enhancements 20

Tel (615) 343-7472 Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

3.8 User Guides ... 20
3.9 AVM Involvement and Conclusion ... 20
4.0 External Analysis Tool Integration ... 21
4.1 Objective ... 21
4.2 Architecture ... 22
4.3 Data Flow ... 23
4.4 Detailed Description ... 23
4.5 Future Enhancements ... 25
4.6 AVM Involvement ... 25
4.7 Conclusion ... 25
5.0 Performance Optimization and User Interaction Enhancement ... 25
5.1 Summary ... 25
5.2 Objective ... 26
5.3 Detailed Description ... 28
5.4 Validation ... 31
5.5 AVM Involvement ... 31
5.6 Conclusions ... 31
6.0 WebGME Development ... 32
6.1 Summary ... 32
6.2 Objective ... 33
6.3 Architecture ... 35
6.4 Data flow ... 37
6.5 Detailed Technical Description ... 38
6.5.1 AVM Component Model ... 39
6.5.2 AVM Design Model ... 42
6.5.3 AVM Test Bench Model ... 47
6.5.4 Domain Specific User Interface ... 50
6.6 Future Enhancements ... 54
6.7 AVM Involvement ... 54

6.8 Conclusions 55

Tel (615) 343-7472 Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

List of Figures

Figure 1: Job Manager - architecture 2
Figure 2: Screenshot of the Job Manager - shows jobs failed, succeeded etc. 3
Figure 3: Screenshot of the Configuration - show remote setup with username and pass 4
Figure 4: Showing jobs from multiple users 5
Figure 5: Project Analyzer architecture diagram 8
Figure 6: Project Analyzer data flow 9
Figure 7: Response Surface for Power Take Off Module Temperature w.r.t. Grade and Coefficient of Rolling Resistance 11
Figure 8: Vehicle speed vs. time 12
Figure 9: Parametric Exploration Tool – architecture 14
Figure 10: Parametric Exploration Tool - data flow 14
Figure 11: A CyPhy PET model with a DOE driver generating a Response Surface Surrogate Model 15
Figure 12: Analysis tool architecture 22
Figure 13: Analysis tool data flow 23
Figure 14: WebGME high-level architecture diagram (http://webgme.org/WebGMEWhitePaper.pdf) 35
Figure 15: WebGME Data Flow 37
Figure 16: ADMEditor language concepts 38
Figure 17: A Component representing a Cross Drive in CyPhyML in GME 40
Figure 18: META model of AVM Component Model in ADMEditor in WebGME 41
Figure 19: AVM Component Model in ADMEditor in WebGME 42
Figure 20: A Hierarchical Design-Space in CyPhyML. Shown are Connectors, Components, Compound and Alternative Containers. 43
Figure 21: META model of AVM Design Model in ADMEditor in WebGME 44
Figure 22: A Hierarchical Design-Space in ADMEditor. Shown are Connectors, Components and Containers. 45
Figure 23: Data flow during Design Space Exploration from the Domain Specific WebCyPhy UI. 46
Figure 24: Design Space with generated Configurations visualized in the Domain Specific WebCyPhy UI 46
Figure 25: META model of AVM Test Bench Model in ADMEditor 47
Figure 26: META model of AVM Test Bench Model with a Top Level System Under Test containing only the interfaces of an AVM Design Model. 48
Figure 27: Selecting an AVM Design Model as Top Level System Under Test of a Test Bench (before). 49
Figure 28: Selecting an AVM Design Model as Top Level System Under Test of a Test Bench (after). 49
Figure 29: Input Artifacts generated by the Test Bench Runner. 50
Figure 30:Listing of all available workspaces 52
Figure 31: Creating a workspace 53
Figure 32: Workspace details showing available Components, Design Spaces and Test Benches 54

Tel (615) 343-7472 Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

Tel (615) 343-7472 Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

VANDERBILT
UNIVERSITY

List of Tables

Table 1: Integrated analysis tools.. 25

1.0 Job Manager and Remote Execution

1.1. Summary

The Job Manager is a CyPhy Test Bench execution utility used by the OpenMETA system to isolate the process of composing an analysis job from executing that job on computer hardware. The Job Manager can perform execution jobs locally on user's machine as well as on remote servers. It provides a user interface to manage the Job Manager's configuration (i.e., select local or remote execution), monitor the status of jobs, and get information about the available remote resources. All jobs are parallelized according to the available resources (local or remote).

The Job Manager runs as a separate standalone process so the GME application thread is not blocked by the Job Manager, and model editing can continue in GME. The execution jobs can be posted to the Job Manager using the Job Manager Library and API. Currently, GME is the only system that uses these API's and library. The Job Manager is an executable GUI application implemented in .NET (C#).

1.2. Objective

The META design flow approach has reduced the design time, one effect of which is the increased the number of simulations per time period. Using traditional design approaches, designing a single configuration yields a few simulations. Users traditionally manually prepare, configure, execute, and analyze these simulations for single design points. Since the OpenMETA design flow considers entire design spaces, including multiple architecture choices, the collective analysis and simulation of all design points has become a computationally intensive process. In fact, it became infeasible for an OpenMETA user working on a single workstation with a typical design space to execute all required analyses linearly (i.e., one simulation after another). The first reason being that some execution tools used in the FANG competition are incompatible with Windows; second, in some typical design scenarios there were hundreds or thousands of analyses to perform, and some individual analyses took several hours to execute. Parallel execution of analyses can significantly decrease the net execution time and get analysis data back to the user in a timely manner.

Modern computers have processors with more than one core available for utilization by our application. The Job Manager supports parallelizing the executions of independent analyses both on the user's machine and on remote server cloud nodes depending on the user's configuration. The remote execution framework introduced flexibility to use tools which target different platforms and operating systems. Thus, if the user's machine does not have a particular analysis tool (e.g., the tool is not compatible with their platform)

then they can use the remote execution service to perform the analysis on a remote machine, which has all the required tools. Since FANG-1 competitors and beta/gamma testers used the VehicleFORGE platform for collaboration, our solution was implemented to support the VehicleFORGE authentication, job posting, deleting, monitoring, artifact uploading, and artifact downloading APIs.

1.3. Architecture/Data flow

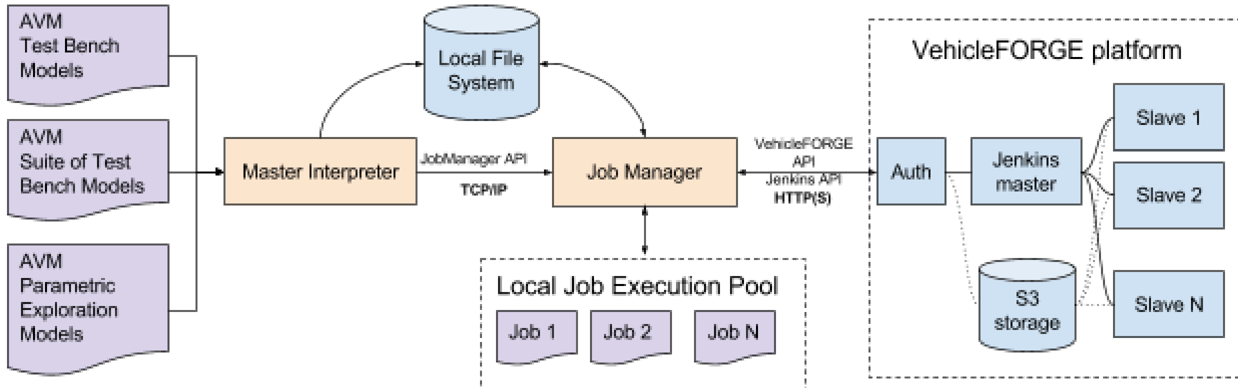

Figure 1: Job Manager - architecture

Users invoke the Master Interpreter on an AVM Test Bench model, an AVM Suite of Test Benches model, or an AVM Parametric Exploration model. The Master Interpreter prepares the temporary results directories on the local file system, opens the Job Manager (if it is not already running), and then posts the analysis jobs by using the Job Manager API. When the Job Manager opens, the user is prompted to choose either a local execution pool or a remote cloud-based execution pool running on VehicleFORGE. If the local pool is chosen then the Job Manager executes all jobs locally and writes the results directly back to the local file system. If the Job Manager is configured for remote execution, after successful authentication, the temporary result directories are uploaded to VehicleFORGE and the statuses of the jobs are updated through the VehicleFORGE API. When the remote jobs are completed the results are downloaded to the local file system by the Job Manager (see Figure 1).

1.4. Detailed Description

The Job Manager is a Windows-based application implemented to facilitate the allocation, execution, and monitoring of jobs such as CyPhy Test Benches, Parametric Exploration models, and Suite of Test Benches (see Figure 2). Each job is an execution job and has a set of properties, listed below:

- Id: Unique identifier of the job
- Title: Description of the analysis

- TestBenchName: Name of the Test Bench that spawned the job
- WorkingDirectory: Directory for generated files
- RunCommand: Command the Job Manager will run
- Status: indicates the current status of the job. Possible values are as follows: WaitingForStart, Ready, QueuedLocal, RunningLocal, UploadPackage, ZippingPackage, PostedToServer, StartedOnServer, QueuedOnServer, RunningOnServer, DownloadResults, RedownloadQueued, AbortOnServerRequested, Succeeded, FailedToUploadServer, FailedToDownload, FailedAbortOnServer, FailedExecution, Failed.
- Labels: Description of the combination of software require to run the job
- ResultsZip: the name of the Python script used to clean up the execution workspace after the job is completed on the remote slave machine.

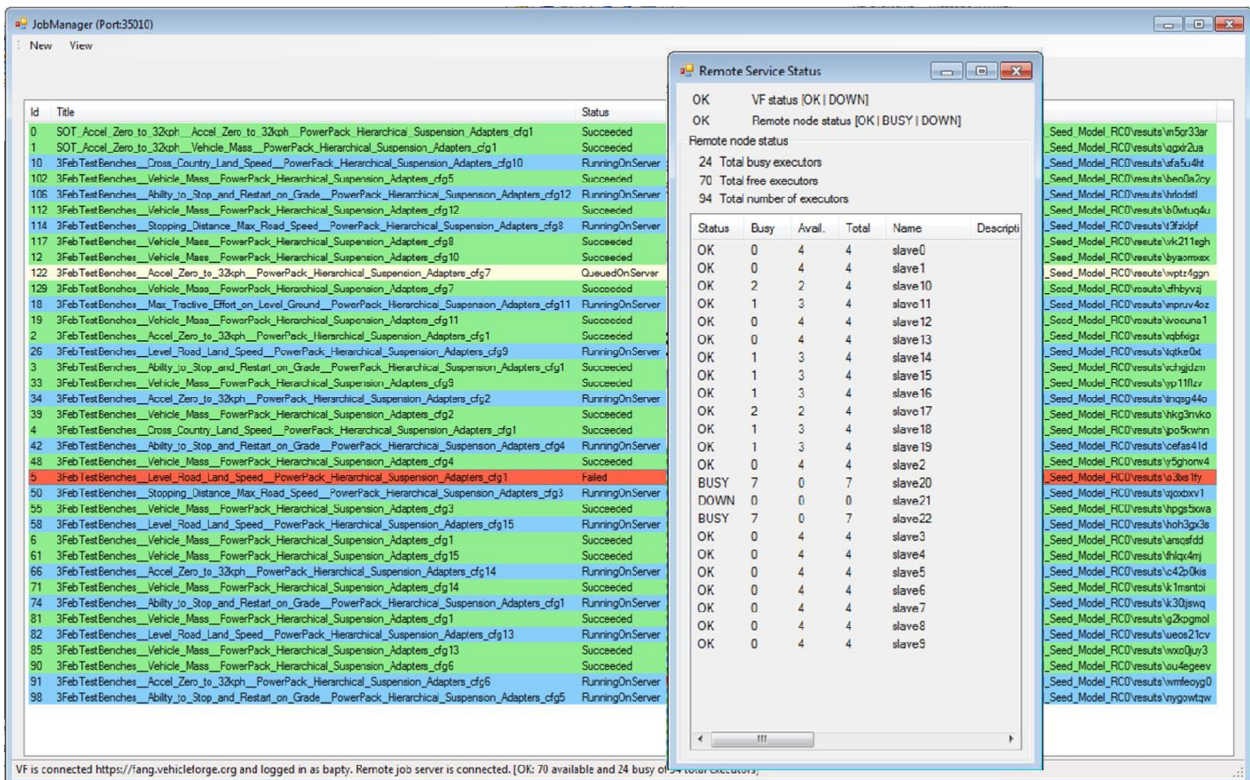

Figure 2: Screenshot of the Job Manager - shows jobs failed, succeeded etc.

The Job Manager supports two modes of operation: (a) local execution of jobs on the user's machine and (b) remote execution on the cloud (see Figure 3). The execution mode is of no consequence to the user; the structure and content of the results package is presented in a consistent manner for both local and remote execution.

Tel (615) 343-7472 Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu
VANDERBILT UNIVERSITY

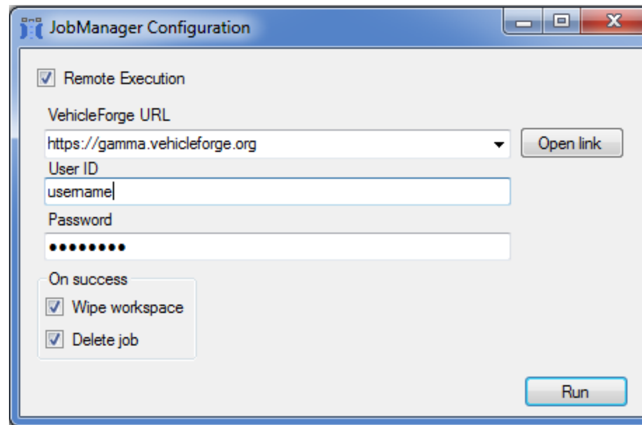

Figure 3: Screenshot of the Configuration - show remote setup with username and pass

In local execution mode the Job Manager employs the user's machine and its resources, including memory and CPU. The Job Manager queries the number of available CPU cores and runs execution jobs in parallel on as many CPU cores are available. This approach increases the throughput of Test Bench and Parametric Exploration model executions.

To enable the remote execution of jobs on servers, the Jenkins continuous integration system was chosen. Jenkins is an extensible open-source continuous integration server implemented in Java that supports scheduling, distribution of jobs, and the management of various slave workers, each of which may use different platforms (we have utilized Windows and Ubuntu workers for AVM). Jenkins provides a REST API for many job management tasks, including the ability to create, delete, and monitor jobs.

Jenkins has various native authentication methods, but these methods would have required all OpenMETA users to manually register on the Jenkins server. Instead we chose to use the VehicleFORGE collaboration platform to provide the authentication layer for all registered users on a specific deployment: beta, gamma, or FANG. Jenkins was configured to isolate different user's jobs and to provide access control to individual jobs.

Jenkins works based on a one-master-node-and-multiple-slave-nodes model, which by default results in the 'archived' job artifacts being transferred to the master machine (i.e. the user's machine) on job completion. Since we are running thousands of jobs with sometimes large execution artifacts, this default behavior does not scale in terms of hard drive usage. Thus, to support large archived artifacts, the Job Manager uploads the job execution source package to temporary cloud storage (i.e., Simple Storage Service [S3] bucket), uses the Jenkins REST API to create a job, and passes the cloud storage URL as a job parameter. The Jenkins master node accepts the job description and schedules the execution job on a slave. Each slave is configured to run a specific number of jobs concurrently. If the number of jobs exceeds the available slave capacity, the jobs will be

queued. Queued items are dequeued in a first-come first-served order, except if the user marks a job as “high priority.” Users can mark only a certain number of jobs as “high priority” per day. As part of the job distribution algorithm, the master node only considers slaves which have all of a job’s labels. The job labels specify the required tools, version, or operating system that the slave node must have in order to execute the job. Once the job description is sent to a slave, the slave downloads the source package from the cloud storage (S3 bucket) and performs the analysis. After an execution attempt, regardless of failure or success, the generated artifacts are packaged and stored in another temporary S3 bucket. If the ResultsZip attribute is specified for any job submitted to the Job Manager, then the ResultsZip Python script is used to perform the packaging/clean-up of the artifacts. This feature is critical when dealing with large redundant artifacts (e.g., CAD components from a CAD Assembly job), as it eliminates unnecessary network and storage overhead. The Job Manager is continuously monitoring all posted jobs to the remote Jenkins master node. If a job status has changed from “in progress” to “failed” or “succeeded,” the Job Manager downloads all archived artifacts, including the standard output log file (i.e., console log) to the client’s machine (i.e. the user’s machine). See Figure 4 for an example Jenkins monitoring dashboard.

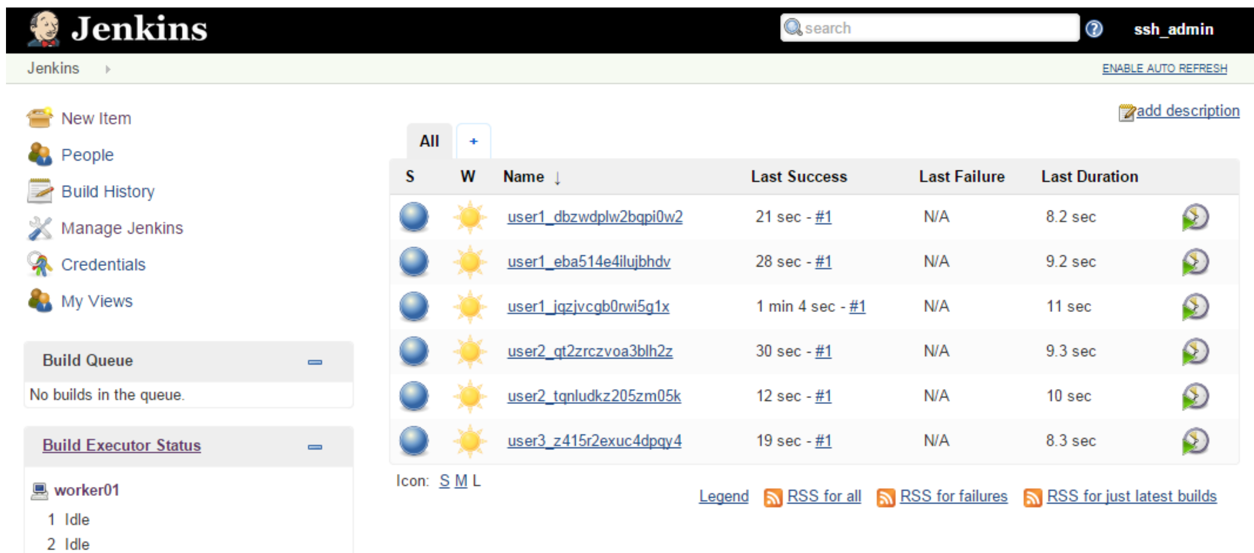

Figure 4: Showing jobs from multiple users

The Job Manager has network communication over HTTP and TCP/IP. Sometimes these connections are unreliable, and for jobs with extremely long execution times (e.g., several hours up to a few days) it is impossible to expect the user to keep the Job Manager application open and running. Therefore, the Job Manager was extended with various capabilities to handle network disconnectivity and recover any pending remote jobs on startup. For certain job types the generated artifact sizes were large, and in some cases a “redownload” capability was required if the user lost the network connection during download.

1.5. Future Enhancements

The current version of the Job Manager supports the VehicleFORGE API for authentication and remote job management. This API can be extended with other authentication services or with other cloud based job scheduling services. Thus, the jobs can be executed on public (e.g., Amazon) or private (e.g., OpenStack) clouds. It is extremely important to consider that all submitted data and results can have different restrictions on availability. Therefore, the connection between the cloud services and the Job Manager must be secure.

1.6. AVM Involvement

VehicleFORGE provided the collaboration platform for many alpha testers, in addition to all beta- and gamma-testers and FANG-1 participants. The Job Manager was used to perform design analysis in the cloud. In certain cases the end users did not have direct access to the analysis tools, since they were only deployed on the remote machines. Moreover, users mentioned above already had credentials for specific VehicleFORGE deployments, which made it easy to authenticate users that can have access to execution resources (e.g., CPU, memory, storage). Since all remote execution jobs went through VehicleFORGE, it was easy to monitor and adjust resources based on the users' needs.

In the gamma testing period, 2640 Test Benches were executed through the remote execution service (on the VehicleFORGE cloud). These Test Bench jobs used twenty-two different types of analysis tools, utilizing both Windows and Linux platforms.

1.7. Conclusions

The Job Manager application provides the flexibility to execute analyses (e.g., run simulations) locally or remotely. Users are able to use their multiple (CPU) core machines to effectively run analyses in parallel over an entire design space. If the size of the design space or the number of Test Benches grows beyond a single machine's computational power or the analysis/simulation takes hours (or even days), the remote executors can be used to distribute the work in the cloud. This approach significantly increased the execution and simulation working capacity and reduced the time required to get analysis results.

2.0 Project Analyzer/Dashboard

2.1 Summary

The OpenMETA toolchain allows creation of large design spaces along with any number of Test Bench models to evaluate design points against specific requirements. The *Project Analyzer*, or *Dashboard*, provides a concise user interface to easily understand and interpret all Test Bench results across user-selected point designs from the design space with respect to design requirements. The Project Analyzer is a web-based cross-platform application implemented in HTML, CSS, and JavaScript. Using various widgets, the Project Analyzer collects and visualizes all relevant design level results (e.g., metrics) in a web application.

The Project Analyzer was developed primarily by Georgia Tech (Aerospace Design Laboratory), with integration and design/debugging assistance from Vanderbilt University. A wide range of visualization techniques were implemented and integrated, including Parallel Axis Plots, design ranking, design point clustering, surrogate model response surfaces, etc. Each of these visualization techniques supports color coding of designs according to Multi-Attribute Utility Function (MAUF) score, ranking, and detection of component limit violations.

2.2 Objective

Complex designs in CyPhy can contain large design spaces, which must be evaluated across a large number of requirements and Test Benches. A visualization of the generated analysis results (e.g. simulation results) is necessary in order to efficiently compare designs to aid in making design choices and tradeoff studies. For this reason, metrics are defined on each Test Bench by users, and the metric values are extracted from the raw simulation data after every execution and stored in a standardized format. The AVM program uses Multi-Attribute Utility Functions to evaluate the “score” of a design point based on its calculated metric values. In addition to displaying the actual metric values for all the executed simulations in a succinct fashion, it would be useful for the results visualization tool to be able to calculate and even display the MAUF score for each design. These were the driving factors during the development of the Project Analyzer.

For the visualization of the execution results, it was logical to use a web-based approach[1]. A web-based implementation gives the flexibility to use any platform that has a web browser, and end users do not need to install any tools or keep them up-to-date. The Project Analyzer was implemented to work with or without an internet connection, including the following use-cases:

- on a server as a deployed web application
- as a visualizer plugin for VehicleFORGE
- locally from the user’s file system

[1] MS Excel was briefly considered, but it is neither open-source nor cross-platform, and was abandoned.

The choice of a web-based approach for the implementation of the Project Analyzer came with benefits and constraints. To get a responsive, usable web application, it was necessary to address all constraints including memory, computational, and scalability. Result sets can quickly become large and cumbersome, and it is important to be mindful of memory limitations and how the data is loaded, processed, stored, and cleaned up. Details are presented in the Project Analyzer report from Georgia Tech ASDL.

2.3 Architecture

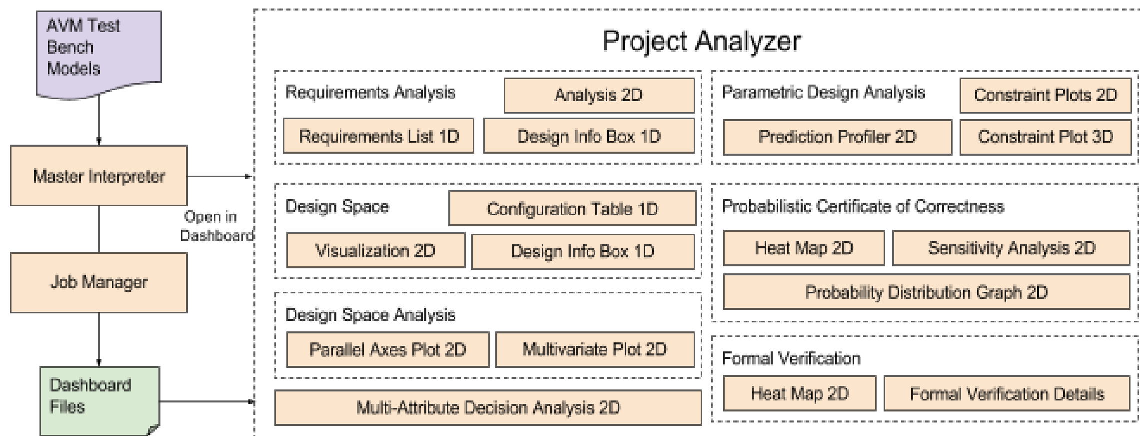

Figure 5: Project Analyzer architecture diagram

The Project Analyzer package is built up by the MasterInterpreter and Job Manager, and contains all the information to visualize and compare a set of results (see Figure 5):

- index.html - a static file; the main ‘entry-point’ for the web application
- dashboard directory - contains both static files (including the HTML page templates, and CSS files), and the JavaScript libraries required to parse the result packages and customize the presentation of the result data; this allows the dashboard to be used as a standalone application, without an internet connection
- manifest.project.json - contains links to all the files describing the results package; loaded by the dashboard application as a map to all results data
- designs directory - contains AVM Design Model (.adm) files describing the individual design configurations generated from the design space
- design-space directory - contains .adm files describing the design space
- requirements directory - contains the requirements (json) file describing the design requirements used for evaluating, scoring, and comparing results
- results directory - contains the results summary (json) file, listing all the result packages generated from the design space; each result package contains metric/limit violation plots and the testbench manifest, which summarizes the Test Bench execution results

- test-benches directory - contains Test Bench description files (Parameters, Metrics, etc.)

2.4 Data flow

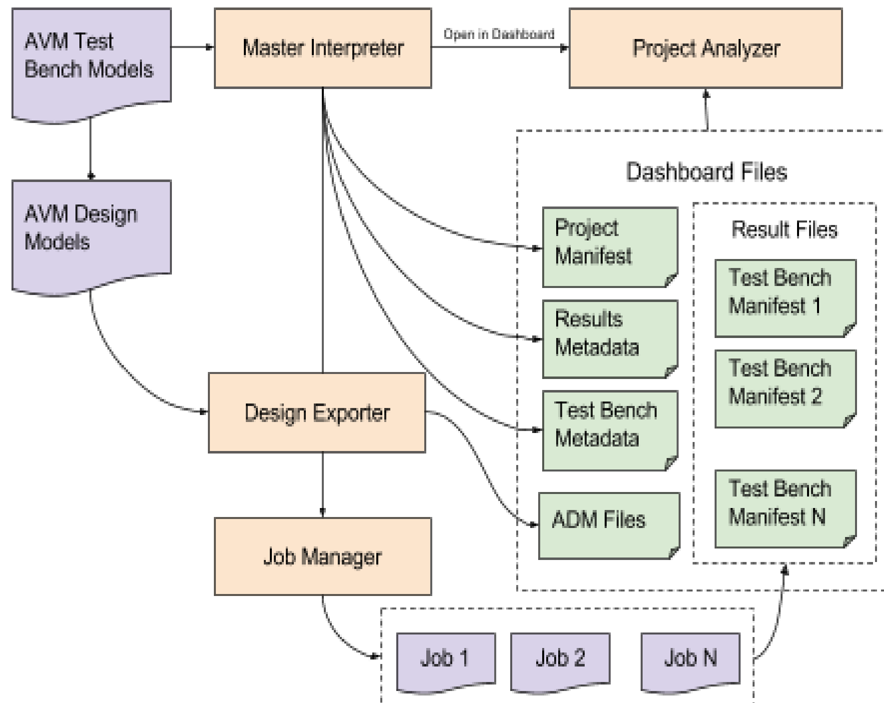

Figure 6: Project Analyzer data flow

Users generate design points using the OpenMETA tools' design space exploration functionality and execute CyPhy Test Benches to evaluate the system level requirements of their point-designs (see Figure 6). Once the Test Bench executions are finished, the Project Analyzer can load all generated analysis results and display them for the users in an interactive format. Then, users can more easily compare the performance of each design and make tradeoff decisions based on the presented analysis results. Based on the data analysis, further design constraints can be formulated and added to the original CyPhy design space. Through this iterative process the design can rapidly evolve in a constructive direction, allowing users to quickly get functional point-designs which satisfy all the defined system level requirements.

2.5 Detailed Description

The Test Bench Manifest file, which is the standard result format, allows the Project Analyzer to display metric values for multiple designs and Test Benches simultaneously; the user can also add a constraint from the visualizer, and immediately see which designs

fail to meet that constraint. The Project Analyzer provides a set of widgets to visualize all data and results generated by the OpenMETA toolchain. In addition to the visualization widgets, users can reorganize and change each page's layout as well as resize the widgets to get all the relevant details in a single page. The design points can be color-coded on several widgets based on limit violation, scoring, ranking, meeting the requirements, etc. The visualization framework ensures that the color coding is consistent across all widgets, which helps the user to understand and interpret the data. In case the Project Analyzer is deployed on VehicleFORGE, the scoring and component hyperlink functionality is enabled. Users can see the leaderboard, how their design performs compared to others, and can easily navigate to any component documentation from the widgets that list components.

The Project Analyzer provides one-, two- and three-dimensional visualization techniques for different purposes. The one-dimensional techniques are used for the Test Bench metric visualization, the physical limit violations on components, and the formal verification results with counter examples (if applicable). The two-dimensional techniques are the parallel axis plot, the multivariate plot, the prediction profiler, the Probabilistic Certificate of Correctness (PCC) distributions, and the design ranking. The parallel axis plot shows all the user-selected metrics on individual vertical axis and represents each design point as a line, which connects the vertical axis. The multivariate plot is a scatter plot, where the x-axis is one metric, the y-axis is another metric, and the points represent design points. Users configure which metrics to plot. The prediction profiler is available if a surrogate model was generated for a Test Bench. The prediction profiler is used to predict the Test Bench metrics (e.g., system performance characteristics) based on the input parameters of the Test Bench utilizing the surrogate equations. If PCC experiments are performed, the output distributions and sensitivity measures are plotted for each Test Bench. The design ranking widget provides visualization and configuration for weights to evaluate the Multi-Attribute Utility Functions (MAUFs), to compare design points, and to rank all design points according to the relative scores. This capability can be used to answer several design problem questions:

- Which design should we use for a set of requirements?
- Which design is the best for a specific set of weights?
- Which designs remain on the top of the list even if we change the weights?

Finally, the three-dimensional visualization technique is used to display response surfaces for Test Bench surrogate models. The 3D plots show two input parameters of the Test Bench in the xy-axes and one metric on the z-axis (see example in Figure 7). The projections of all 3D plots are shown in 2D constraint plots and the Test Bench input parameters can be set using sliders to change the constraint plots within the ranges for which the surrogate model is valid.

ISIS
Tel (615) 343-7472 Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu
VANDERBILT UNIVERSITY

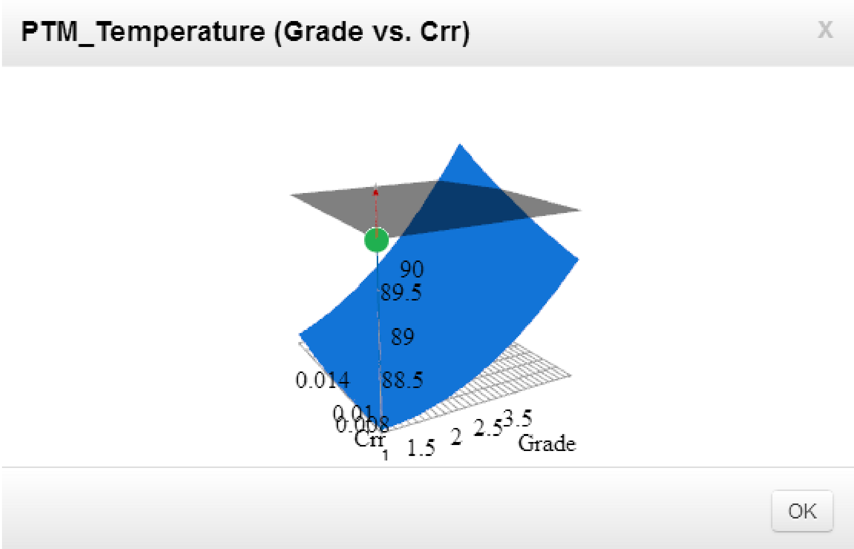

Figure 7: Response Surface for Power Take Off Module Temperature w.r.t. Grade and Coefficient of Rolling Resistance

The Project Analyzer report, prepared by Georgia Tech, contains in-depth details about the application and its widgets, features, accessibility matrix, extensibility, and how other applications can be built to leverage this framework.

The Project Analyzer aims to assist with data visualization across multiple and bigger data sets, hence it is not designed and has no functionality to visualize detailed simulation data (e.g., plots of variables over simulation time). For dynamics simulation result visualization a web-based application was developed called SimViz. SimViz is used to load dynamics simulation results generated by a Modelica simulation tool (e.g., Dymola, OpenModelica). This application allows users to plot a particular simulation variable (vs. time) for many design points, which helps comparing both design and architecture alternatives at the lowest level. For metrics and limit checks the time series of data are automatically saved in the results directory and referenced from the Test Bench manifest file. Thus, the Project Analyzer can load these plots; an example of such a plot is shown in Figure 8 below.

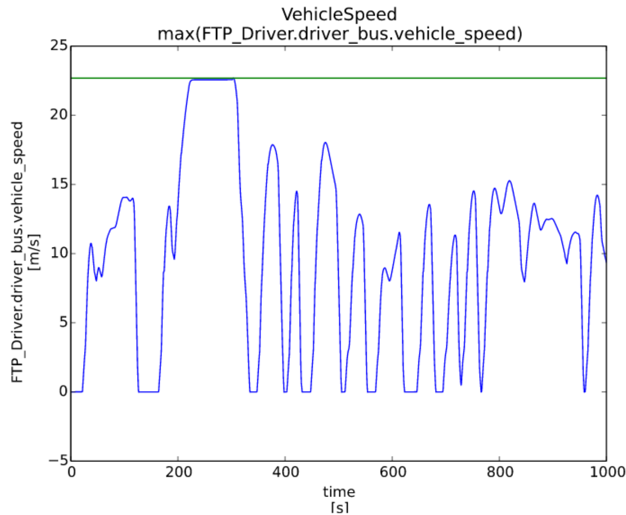

Figure 8: Vehicle speed vs. time

2.6 Future Enhancements

The Project Analyzer was implemented in a way that it is easy to visualize result data from any 3rd-party tool that can produce metric data in accordance with the JSON file format (i.e., the Test Bench manifest file). Also, it is possible to implement new JavaScript widgets to visualize new types of data.

2.7 AVM Involvement

The Project Analyzer was used extensively in alpha testing, beta testing, FANG-1 challenge, and in gamma testing. Each group showed interest in this tool and found it essential to speed up the interpretation of the results and the evaluation of the designs, and even proved useful for internal development (i.e., functional testing). Many beta and gamma testers pointed out features that would improve their current design processes. They also gave feedback on possible future enhancements. For instance, the 'Export as CSV' feature was added after the FANG-1 challenge, allowing users to export any tabulated data. All users from the FANG-1 challenge interacted with the Project Analyzer since it was deployed on the VehicleFORGE platform.

2.8 Conclusions

The Project Analyzer is one of the key components of the OpenMETA toolchain. It provides a rich, web-based, cross-platform, interactive visualization environment for designers. It is highly customizable based on the user's preference, and data can be exported to CSV format if more sophisticated analysis is desired. The Project Analyzer can read any input data conforming to the defined directory and file structure. In other words, if other tools can generate and structure analysis results in the same way as the OpenMETA tools, then the Project Analyzer can easily visualize that results data.

3.0 Parametric Tool Exploration

3.1 Summary

The Parametric Exploration Tool (PET) allows the evaluation of design models and Test Benches across a range of parameters to accomplish various types of design space exploration tasks. The CyPhy language defines a set of concepts which makes it possible for any user to compose a model specifying a design-of-experiment (DOE).

Each PET model consists of a reference to a CyPhy Test Bench and a driver object, which defines ranges and/or statistical distributions for the experiment inputs (i.e., Test Bench parameters), along with constraints on the experiment outputs (i.e., Test Bench metrics). There are three types of PET drivers: (a) Optimizer, (b) Parametric Study (Design of Experiment), and (c) Probabilistic Certificate of Correctness (PCC).

The Probabilistic Certificate of Correctness (PCC) is used to determine the robustness of a single design point. The design may perform well under specific conditions with certain parameters, but we cannot assume those ideal conditions will exist for all time in the future use of that design. PCC gives users the ability to vary inputs and parameters according to expected statistical distributions, perform multiple simulations on the same design, and determine how consistently the outputs behave across those input ranges.

The CyPhyPET tool takes a PET model and translates it into OpenMDAO modules and an OpenMDAO assembly, which defines the Parametric Exploration problem. After the problem is executed in the OpenMDAO framework, results are stored in a format that the Project Analyzer can parse and visualize for the user. The result formats include response surfaces, surrogate model equation sets, and distribution functions for input and output values of the CyPhy Test Bench. The Parametric Exploration Tool (PET) is part of the CyPhy Modeling Language and the model transformation tool (CyPhyPET) is implemented as a GME model interpreter in .NET (C#).

3.2 Objective

The design space exploration tool (DESERT) provides a very good method for solving the discrete design space exploration problem. The design space encodes possible point-designs using alternative and optional component and subsystem choices, along with constraints on those possible combinations. However, in certain phases in the design process, this component-wise discrete design comparison is too crude to be useful. In such cases, users need to analyze designs across continuous parameter ranges, resulting in two common use-cases: (1) they can use PET results to formulate additional constraints on the design space, in turn yielding a subset of design points which are more likely to fulfill design requirements; (2) they can assess the sensitivity and robustness of the finalized design candidates.

3.3 Architecture

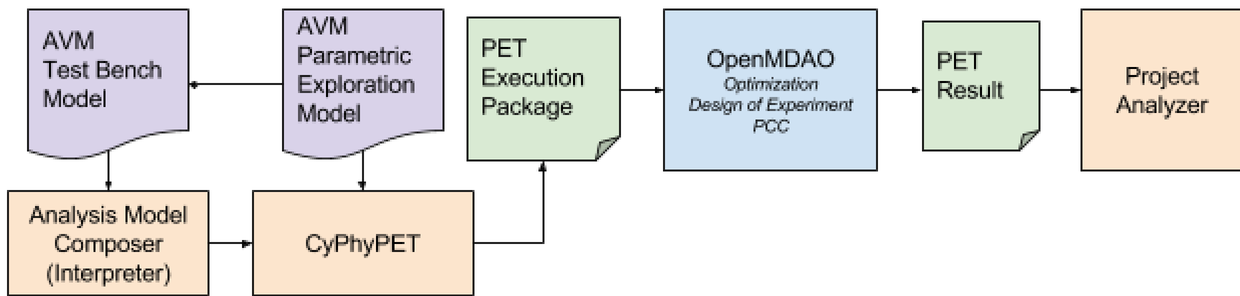

Figure 9: Parametric Exploration Tool – architecture

3.4 Data flow

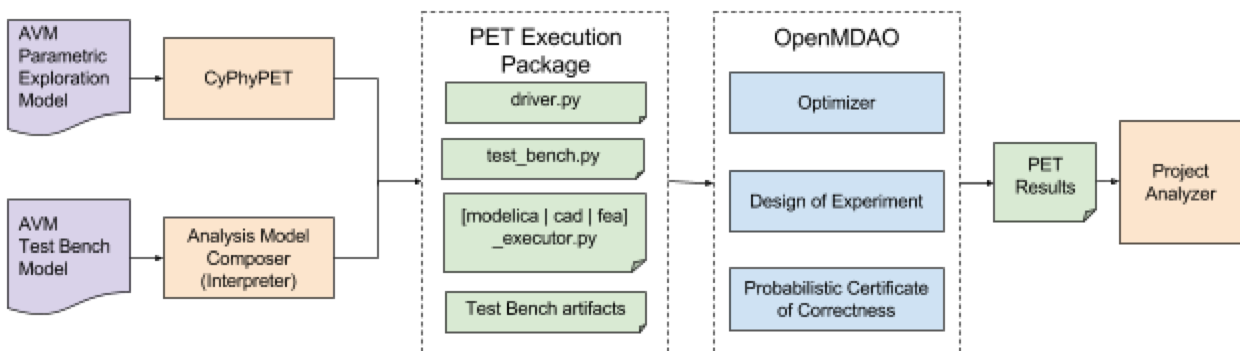

Figure 10: Parametric Exploration Tool - data flow

3.5 Detailed Description

To facilitate the definition and execution of Parametric Explorations problems in compliance with the project requirement to use an open-source tool, we have chosen the Open-source Multidisciplinary Design Analysis and Optimization (OpenMDAO v0.8.1) framework (see Figure 9). OpenMDAO is a cross-platform project that is implemented, maintained, and supported by NASA. It has a wide range of existing plugins, and boasts an active user community. The OpenMDAO framework is implemented in Python and utilizes “components,” which are Python-wrapped functions or executables, the logic or execution of which is provided by some external tool. Because CyPhy Test Bench models are executable versions of the user-defined requirements, it is a reasonable choice to wrap them as OpenMDAO components. In this way, the OpenMETA tools’ CyPhyPET component provides design-of-experiment capabilities, and its implementation is provided in such a way that it is easy to extend it with any new CyPhy Test Bench types, as well as any plugins and drivers from the OpenMDAO community (see Figure 10). A detailed description of the CyPhyPET model interpreter is presented below. An example of a driver extension is the Probabilistic Certificate of Correctness (PCC) robustness analysis, which is implemented as an OpenMDAO driver leveraging all OpenMDAO features and capabilities, such as parallel execution of experiments. In the scope of this project, users ran Parametric Exploration experiments on both Windows and Linux (Ubuntu) platforms.

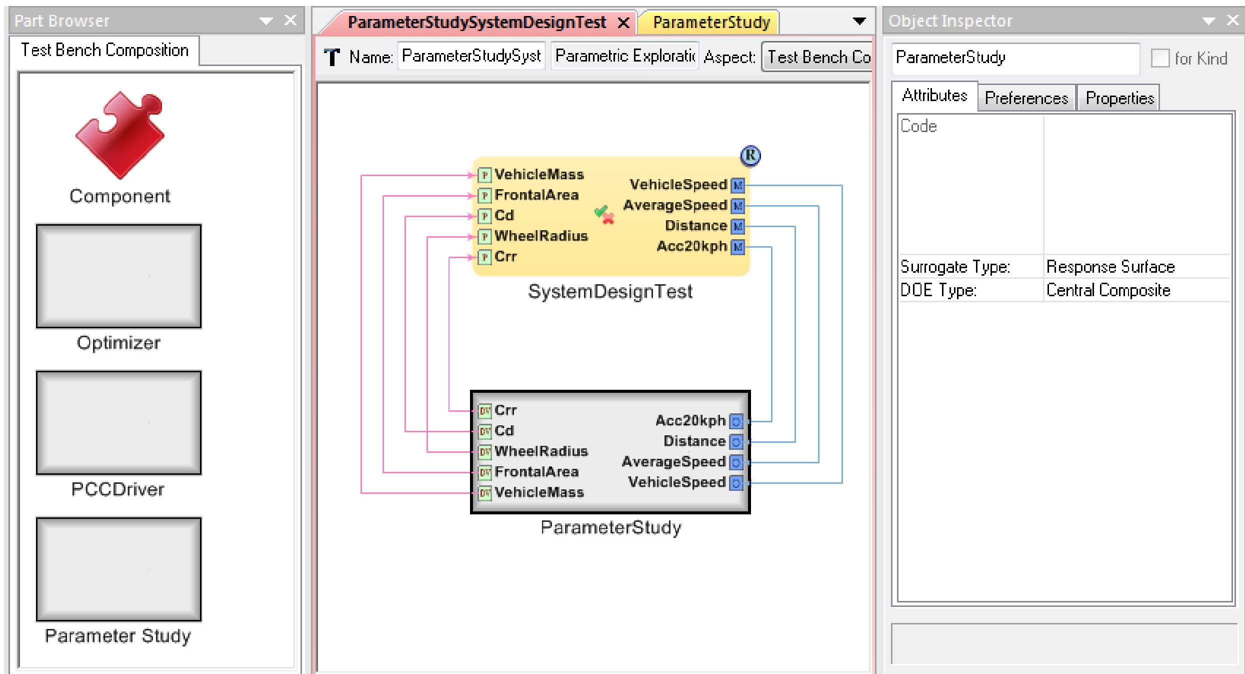

Figure 11: A CyPhy PET model with a DOE driver generating a Response Surface Surrogate Model

An example of a PET model is shown in Figure 11. This example shows the different PET driver types on the left-hand side and some attributes of a Parameter Study driver on the right-hand side. The middle window depicts a CyPhy Test Bench, a Parameter Study driver, and the experiment set up through the connections between the Test Bench and the driver. To implement PET models, the CyPhy Language had to be augmented with some new concepts:

- Test Bench Reference object - used to point to an existing Test Bench model
- PET Driver object
    - Optimizer
        - Optimizer methods
            - CONMIN: CONstrained function MINimization. Implements the Method of Feasible Directions to solve the NLP problem
            - NEWSUMT: NEWton's method Sequence of Unconstrained Minimizations
            - COBYLA: Constrained Optimization BY Linear Approximation of the objective and constraint functions via linear interpolation
        - Parameters/Outputs
            - Design Variables are connected to the parameters of the referred Test Bench. The 'range' attribute defines an inclusive real interval that is sampled.
            - Objectives are connected to metrics of the referred Test Bench and define which values have to be minimized.
            - Optimizer Constraints are custom mathematical expressions (using Python syntax) of Design Variables and Objectives.
    - Parameter Study (Design of Experiment)
        - Design of Experiment methods
            - Full Factorial generates a set of design points that fully span the range of the parameters at the requested resolution
            - Central Composite
            - Opt Latin Hypercube produces an optimal Latin hypercube based on an evolutionary optimization of its Morris-Mitchell sampling criterion
            - Uniform performs a uniform space-filling Design of Experiments
        - Surrogate Model Types

Tel (615) 343-7472 Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu

- Response Surface: Surrogate Model based on second order response surface equations
- Kriging Surrogate: Surrogate model based on the simple Kriging interpolation
- Logistic Regression: Surrogate Model based on a logistic regression model, with regularization to adjust for overfitting
- Neural Network: Feedforward Neural Net surrogate model

- Parameters/Outputs
  - Design Variables are connected to the parameters of the referred Test Bench. The 'range' attribute defines an inclusive real interval that is sampled. Alternatively, an array of values can be specified for each design variable, in which case the driver will sweep over the user-defined set of discrete input values.
  - Objectives are connected to metrics of the referred Test Bench, and define the output values to be stored for each Test Bench execution. These metric values, along with the design variables, are used for calculating the surrogate models.

- Probabilistic Certificate of Correctness (PCC)[2]
  - Uncertainty Propagation methods
    - Monte Carlo Simulation
    - Taylor Series Approximation
    - Most Probable Point Method
    - Full Factorial Numerical Integration
    - Univariate Dimension Reduction Method
    - Polynomial Chaos Expansion
  - Sensitivity Analysis methods
    - Sobol
    - FAST
    - EFAST
  - Parameters/Outputs:
    - Input distributions types: Beta, Log-Normal, Normal, and Uniform

[2] For in-depth implementation details on PCC, please refer to the OSU PCC report.

- PCC Output: defines a minimum and maximum value for metrics and the target joint PCC value.

The currently supported Test Bench types in a PET model are:

- Dynamics Test Benches using OpenModelica or Dymola
- CAD Test Benches using Creo
- FEA Test Benches using Abaqus

One merit of using OpenMDAO is the improved scaling provided with parallelism, since independent OpenMDAO jobs can be executed simultaneously, allowing faster turnaround time. In the same way that CyPhy Test Benches are translated into 'jobs' for execution, Parametric Exploration models are translated into a set of independent OpenMDAO execution tasks; this set of OpenMDAO tasks can be scheduled for execution by the Job Manager, and allocated to execution resources (both local and remote) as they become available. A further benefit is the utilization of surrogate models, which are approximations (e.g., polynomial) of the Test Bench models; this often reduces computational time compared to the original Test Bench execution, resulting in reduced design time. A user-specified surrogate model is automatically generated for Test Benches when a DOE is performed.

CyPhyPET is a GME interpreter that translates PET models into a set of OpenMDAO components in the form of Python scripts. CyPhyPET implements the *IMgaComponentEx* and *ICyPhyInterpreter* interfaces and works on a PET model, where the referred Test Bench contains a single point-design. Since the *ICyPhyInterpreter* interface is implemented by CyPhyPET, the Master Interpreter can be used to invoke CyPhyPET on the PET model template over an entire design space.

In order to wrap the composed CyPhy Test Bench and generate executable PET artifacts, CyPhyPET must first invoke a domain-specific model translation interpreter, defined by the Test Bench's workflow object. CyPhyPET evaluates 55 structural model checking rules in five contexts, and if the PET model is invalid, it generates error/warning messages and hints for the users to fix the errors. CyPhyPET also has to prompt the user (once) to get all necessary configurations (e.g., the set of design-points to analyze and the domain-specific translator's configuration). Once the user has entered the required configuration and the PET models have been elaborated over the design space, the PET model is processed by the CyPhyPET interpreter. The following files are generated in the defined temporary results directory:

- driver_runner.py
  - Entry point and error handling.
- test_bench.py
  - OpenMDAO component that wraps the Test Bench.
- modelica/cad/fea_executor.py

  - Domain specific implementation for running initial Test Bench execution, updating parameters and executing the Test Bench.
- driver.py
  - OpenMDAO Assembly defining the Optimizer, Parameter Study or Probabilistic Certificate of Correctness experiment.

Additionally, save_results.py, surrogate_model.py and SurrogateModelValidation.py are generated for Parameter Studies. The latter two are generated when a surrogate model is defined.
- surrogate_model.py
  - OpenMDAO Assembly defining the surrogate model (for the Test Bench).
- SurrogateModelValidation.py
  - Does an evaluation of the generated surrogate model.

After all files are successfully generated, the MasterInterpreter creates a new job entry in the Job Manager and requests the execution of each job (one job is generated per point-design). The results directory contains analysis artifacts after the automated execution of the OpenMDAO experiment. The generated artifacts are as follows:
- Optimization
  - No artifacts are generated.
- Parameter Study
  - meta_model_info.p - pickle file (serialized object) containing the surrogate model (when such is chosen in the experiment definition). It is loaded during the surrogate model evaluation and can be used in further experiments.
  - output.csv - list of inputs and outputs for each iteration.
  - model_perf.json - summary of the surrogate model evaluation.
- Probabilistic Certificate of Correctness
  - parameters.csv - list of inputs and outputs for each iteration.

Since the results are stored in a unified way within the Test Bench Manifest file, the Project Analyzer can parse the outputs of the OpenMDAO experiment and can visualize all analysis results. Parametric Study results are visualized as 3D surface plots, surrogate models, and an interactive prediction profiler. Probabilistic Certificate of Correctness results are visualized as distribution functions and heat maps across the design space and across all the different Test Benches. See the Project Analyzer section for further visualization details.

3.6 Validation

The OpenMDAO environment includes several examples, for instance the optimization problem of a paraboloid. As we developed the CyPhyPET model transformation tool, we used some of their examples, including the optimization problem of a paraboloid, to validate our Python source code generator. The execution results of the generated source code were continuously compared to the results that the built-in examples provided. This validation process ensured that the OpenMDAO components are accurately generated from the CyPhy models.

3.7 Future Enhancements

The current version of the OpenMETA toolchain (14.12) uses the 0.8.1 version of OpenMDAO. To enable the latest OpenMDAO drivers and features, the OpenMETA tools should be updated to use OpenMDAO version 0.10.3.2. Additional drivers and plugins can be installed or implemented to provide more flexibility in terms of analysis capabilities. Support for unit conversion (between the driver and the Test Bench) and non-linear value flow expressions (between parameters) would be a valuable improvement.

3.8 User Guides

The end user documentation is published on VehicleFORGE and comes with the OpenMETA toolchain installer. Once the tools are installed the documents are available in the Public Documents directory.

3.9 AVM Involvement and Conclusion

The Parametric Exploration Tool was used extensively in the AVM program, both as a design tool for FANG users and in the C2M2L curation project as a component model verification tool. CyPhyPET supports and was used in both dynamics and 3D geometric analyses, including automated parametric analysis of Lumped Parameter Dynamics (i.e., Modelica) models and Finite Element Analysis experiments using Creo and Abaqus. In the FANG competition and in beta/gamma testing, CyPhyPET provided the users with sensitivity analysis, optimization, and design of experiment capabilities, all of which contribute to a high level of confidence in the final design selection. It provides the ability to compose and compile a single system model, and to rapidly execute many simulations under similar conditions with minimal user interaction. Users may select from different drivers, and for PCC, they may choose from different analysis methods. In the case of PCC, the estimated number of executions (based on the method and number of inputs/outputs) will be shown on the GME console, giving users an idea of how much time the analysis will take to complete.

In the context of the C2M2L component curation process, CyPhyPET was used on unit Test Benches for component model verification. Parametric Exploration, PCC in particular, provided the ability to quantify the robustness of parametric component (Modelica) models from the C2M2L Modelica library, and to verify that the models were indeed parametric and were functional across the full parameter ranges specified by the model author. A component model is of limited value (and is therefore considered of lower quality) if its behavior is fragile in relation to the choice of parameter values. Thus, it is important for designers to know the limitations of any component models used in the system design, since fragile component models will contribute to fragile system models. The parametric exploration and analysis capabilities provided by CyPhyPET are invaluable in the design process, and gives users an easy way to configure such experiments, leveraging the OpenMDAO framework.

4.0 External Analysis Tool Integration

In the OpenMETA toolchain, all system-level design requirements are captured in Test Benches as metrics. Test Benches must be executed to produce the metric values and evaluate all requirements. Each Test Bench execution may require different tools to perform a specific analysis. The OpenMETA framework provides a flexible approach to the integration of third-party analysis tools.

4.1 Objective

The standard OpenMETA tools already provide the means to generate execution artifacts for typical cyber-physical systems, including dynamics behavior models and CAD (i.e., geometric) system assemblies, but specialty users may need to build upon these capabilities to extract metrics for their specific domains. For this reason, we want to provide a simple way to mesh new custom analysis tools with OpenMETA. Thus, all analysis tools are implemented as self-contained directories and the OpenMETA tools provide hooks to inject user-defined scripts (i.e., external analysis tools or modules) into the Test Bench execution process. This approach has the several merits, which benefit both the robustness/extensibility of the OpenMETA toolchain and any 3rd-party tool developer:

- Extension modules can be deployed to add new analysis capability without recompiling the OpenMETA source code
- No restriction on the programming language for the external analysis tools
- External analysis tools can leverage any model transformation component provided by OpenMETA

- Each analysis tool can be executed over an entire design space including different architecture alternatives, which yields significant design time reduction compared to traditional methods
- Analysis tools are executed by the Job Manager, which provides a parallel execution framework over multiple Test Benches and design points
- Results (i.e., metrics) of the analysis are presented by the Project Analyzer at no extra implementation cost. This helps end users to visualize all results over a design space and make decisions based on the design rankings.

4.2 Architecture

New analysis tools can be easily added to the OpenMETA toolchain without the need to recompile the OpenMETA source code or build a new OpenMETA installer from scratch. Each analysis tool has to be put into a special directory (i.e., *%METADIR%\analysis_tools*), along with a tool descriptor file (*analysis_tool.manifest.json*) file and all the execution files including the entry point of the tool. Figure 12 depicts the logical grouping of this directory structure as well as examples for a manifest file and a workflow configuration. After the analysis tool is created in the directory, a registration process is executed that registers all analysis tools. All AVM/CyPhy Test Bench models include a workflow object that lists all registered analysis tools, which are associated with the selected CyPhy interpreter (i.e., model transformation tool). This architecture makes the analysis tool integration testing easier, since the integration can be done on any machine where the OpenMETA toolchain is installed.

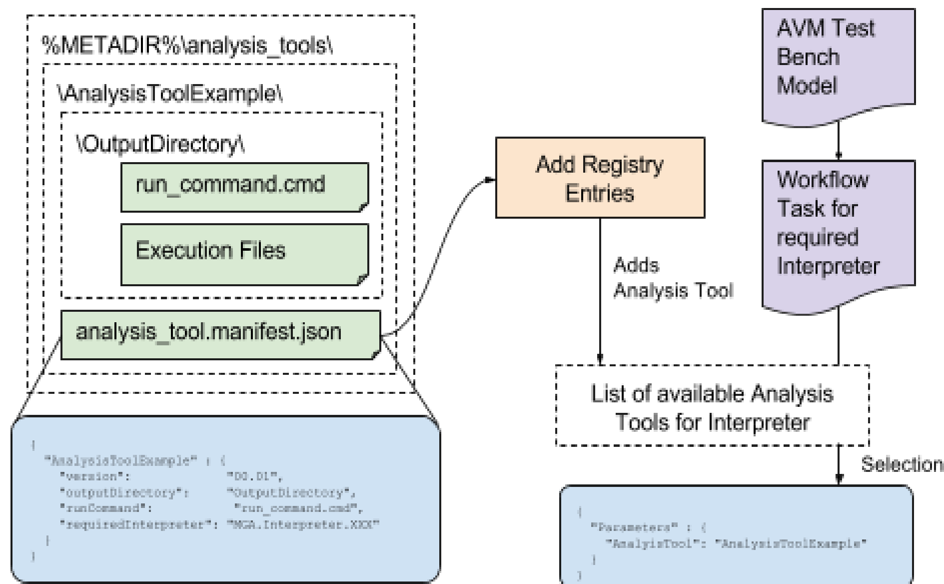

Figure 12: Analysis tool architecture

4.3 Data Flow

Each AVM/CyPhy Test Bench model contains a workflow reference that defines a *Task*. The *Task* in the workflow object specifies a CyPhy *Analysis Interpreter*, which creates/exports some artifacts from the CyPhy model. Users are prompted to select the analysis tool from a list; when the *Default* option is selected the CyPhy *Analysis Interpreter* generates all default execution artifacts and an entry point for the default analysis. If the analysis tool is specified, for example *AnalysisToolExample*, then the execution files are copied into a temporary output (i.e., result) directory by the *Analysis Interpreter*. The generated result package is posted to the Job Manager; and the Job Manager executes the analysis job and saves the analysis results. See Figure 13 for an depiction of the analysis tool interpretation flow.

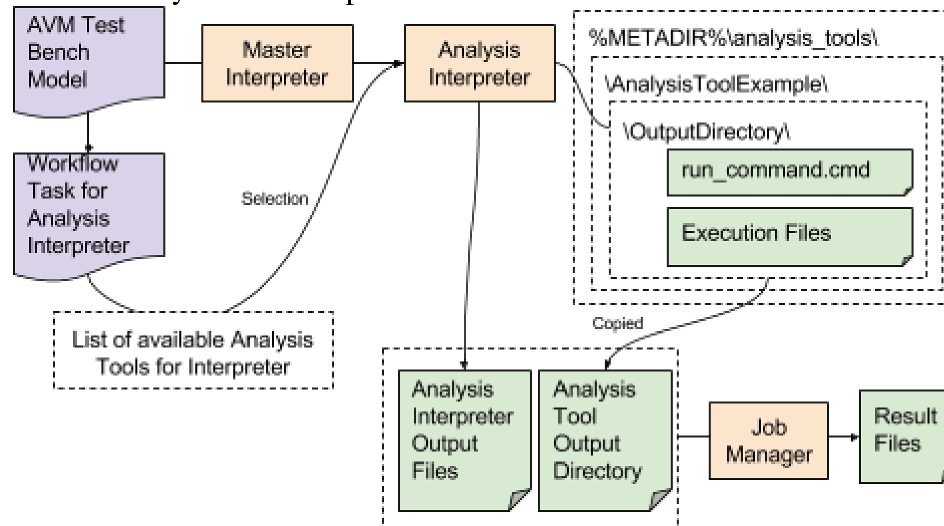

Figure 13: Analysis tool data flow

4.4 Detailed Description

Many analysis tools are integrated with the OpenMETA toolchain as shown in Table 1. The analysis tools can be implemented in any programming language as long as they have an entry point that can be called from command line (on Windows or Linux). If the analyses are executed remotely, then the remote slave computers must have the specific analysis tools and all of their dependencies installed separately if they are not part of the OpenMETA installer.

If an analysis tool is integrated into the OpenMETA toolchain, then the new tool can leverage **all** OpenMETA capabilities, such as discrete design space exploration and parallel execution of the analysis over a design space both locally and remotely (using the Job Manager).

Analysis tool name	Implemented by	Used model	Programming

		transformation (Analysis Interpreter)	**language**
CFD	Vanderbilt University	CyPhy2CAD_CSharp	Python
Closures	Ricardo Inc.	CyPhyCADAnalysis	Python
Completeness	PSU (iFAB)	CyPhyCADAnalysis	Python
Conceptual manufacturing	PSU (iFAB)	CyPhyCADAnalysis	Python
Corrosion	Ricardo Inc.	CyPhy2CAD_CSharp	Python
Detailed manufacturing	PSU (iFAB)	CyPhyCADAnalysis	Python
Ergonomics	Ricardo Inc.	CyPhyCADAnalysis	Python
Example Count Components	Vanderbilt University	CyPhyDesignExporter	Python
Example Generate Bill Of Materials	Vanderbilt University	CyPhyDesignExporter	Python
FAME Critical Fault Count	PARC	CyPhy2Modelica_v2	Python/Java
FAME Fault Count	PARC	CyPhy2Modelica_v2	Python/Java
Field of fire	Ricardo Inc.	CyPhyCADAnalysis	Python
Field of view	Ricardo Inc.	CyPhyCADAnalysis	Python
Freed Linkage Assembler	Vanderbilt University	CyPhy2CAD_CSharp	Python/Java
HybridSal (Formal Verification)	SRI	CyPhy2Modelica_v2	Python/SAL
Ingress and egress	Ricardo Inc.	CyPhyCADAnalysis	Python
Qualitative Reasoning (QR) Module (Formal Verification)	PARC	CyPhy2Modelica_v2	Python/Lisp
RAMD	PSU (iFAB)	CyPhyCADAnalysis	Python

Transportability	Ricardo Inc.	CyPhyCADAnalysis	Python

Table 1: Integrated analysis tools

4.5 Future Enhancements

Any tool can be integrated with the OpenMETA toolchain as long as the new analysis tool can take artifacts generated from the OpenMETA toolchain as inputs. Several model transformations are provided to generate artifacts in the form of *Analysis Interpreters*; for instance, they can generate composed geometric models (i.e., 3D CAD models) and composed dynamics models. If additional information has to be extracted from the CyPhy models, then a new *Analysis Interpreter* can be implemented and added to the toolchain, which may require developers to recompile the tools and build a new installer.

4.6 AVM Involvement

Several analysis tools were used during beta testing, the FANG challenge, gamma testing, and other AVM performers. One beta testing team has implemented an example analysis tool that uses the Java programming language. Some system level requirements (e.g., transportability, field of view, etc. developed by Ricardo) were evaluated by the analysis tools listed above during the FANG competition and gamma testing.

4.7 Conclusion

The OpenMETA tools provide several useful model transformation interpreters (i.e., Analysis Interpreters), which are deployed as part of the installer. These interpreters generate executable artifacts from the multi-domain CyPhy models, including dynamics behavior models and geometric analysis techniques. These general built-in capabilities provide a good start on the path of designing cyber-physical systems, but it is common that domain-specific design teams may need to implement specialty downstream tools for post-processing of the standard OpenMETA execution artifacts. These downstream tools are typically used to evaluate specific system level requirements pertaining to that team's expertise. The OpenMETA developers have provided an easy interface for 3rd party developers to add their custom analysis tools, automate their execution using the Job Manager, and immediately see the impact of the new tools using the Project Analyzer.

5.0 Performance Optimization and User Interaction Enhancement

5.1 Summary

This section describes all major performance optimizations and user interaction enhancements for the OpenMETA toolchain. The most frequently used (critical) software

components were identified and analyzed in order to provide better usability and scalability of the META toolchain. The critical software components are used by many model interpreters, model management tools, and CyPhy Test Benches such as the CyPhyElaboratorCS, the MasterInterpreter and the Job Manager. All three software components are implemented in .NET (C#).

- The CyPhyElaboratorCS is a library (.dll) that unfolds component and component assembly references into component instances and component assemblies, respectively. This tool helped to address scalability (memory) issues, as the models were growing bigger and bigger.
- The MasterInterpreter is a GME model interpreter implemented in .NET (C#). It provides automation to translate any CyPhy Test Benches, Parametric Exploration (PET) models, and Suite of Test Benches (SoTs) over a full design space, meaning that it supports elaboration of Test Bench templates and automatically calls the associated analysis tools defined by the workflow object. After the individual model transformations are done, the generated artifacts are posted to the Job Manager in the form of an execution job.
- The Job Manager is a Windows Forms application implemented in .NET (C#). It makes analysis job execution possible both locally and on remote servers. The jobs are executed in parallel based on the available resources.

In addition to the three critical software components mentioned above, the optimization addressed the performance and usability issues of a few domain-specific programs, such as CyPhy2Modelica and ParameterEditor. For instance, the CyPhy2Modelica model interpreter now runs an order of magnitude faster and contains an extensive structural model checker to prevent downstream execution issues in advance.

5.2 Objective

The AVM Program has provided many opportunities for testing the OpenMETA toolchain. All tools were alpha tested by Vanderbilt engineers and student interns during the development process. There was also an AVM-sanctioned beta testing group which incorporated test users from several industry design teams, including Ricardo, JPL, and others. During the FANG competitions, competitors were encouraged to submit Help Desk tickets whenever they encountered unexpected behavior from the tools. In many cases, the questionable behavior provided an opportunity to improve the User's Guide and tool documentation; a few of the Help Desk tickets turned into OpenMETA bug reports for the Vanderbilt team. The remaining feedback from users became a "suggestion box," the industry designer's wish-list for features that would improve usability of the tools.

There were several recurring themes within the set of Help Desk tickets: many cited a need for performance improvements relating to scalability. Some of these shortcomings

became apparent as new analysis tools were integrated with OpenMETA, and could not have been foreseen at the earliest stages of the tool development process. For instance, structural and thermal finite element analysis (FEA) were added after the execution infrastructure was somewhat mature. Some of these FEA Test Bench execution times required several hours to complete for a single design point. Invariably, the growing size of the design space coupled with the need to execute many time-consuming analyses on each design point resulted in a bottleneck when executing the analyses. The overhead involved in configuring complicated analyses and long execution times may be acceptable on the scale of a few analyses per design iteration (i.e., the typical engineering design flow); however, the OpenMETA design paradigm is was a fundamentally different process, due largely to its utilization of full-fledged execution for a large set of potential design candidates.

The first step was to identify the software components that were central to the bottlenecks forming in the OpenMETA design flow. Next, we tried to quantify the performance and generated overhead of these critical software components by pressing the limits of their usage. This included constructing worst-case scenario models intended to cause problems or failures and profiling the code to narrow the field of focus to the slowest and most-used portions of the code base. These tests revealed a list of inefficient paths that could aid developers in determining which paths deserved highest priority for performance improvements.

Some of the inefficiency was present in 3rd-party analysis tools, and it was impossible to address these issues, short of finding and integrating a new and more efficient tool to fill each role. Some inefficiency was directly correlated with the number of objects (e.g., components and design points) in the CyPhy model, and the resulting improvements were primarily user-interface related, such as adding progress bars to keep the user engaged, using multi-threading to maintain some degree of functionality during execution, and adding the option to cancel a long-running process prior to completion. We also determined that three software components were candidates for a complete overhaul, allowing us to re-implement the full functionality (with improvements) while including optimization as a primary goal: the CyPhyElaboratorCS, the CyPhy2Modelica model translator and the Master Interpreter. The Job Manager received several usability enhancements driven by the longer execution runtimes introduced later in the program. Additionally, the Modelica Importer and Parameter Editor features were created to improve productivity.

There is always room for optimization and usability improvements; if time were not an issue, it may have been appropriate to re-write other sections of the OpenMETA code base, targeting optimization throughout the process. However, this was not the case, and will probably never be the case in the scope of a large-scale cutting-edge research project like AVM, where goals and deliverables are constantly in flux based on changing needs and trial-and-error. New tools and concepts are incorporated while they continue to

mature, and time constraints force improvements to be considered as a secondary priority, during the short periods when integration has reached a state of temporary stability.

5.3 Detailed Description

If an application is user-friendly, it is much more likely to be adopted by design teams. Part of usability is related to software efficiency, and OpenMETA developers made as much effort as possible to analyze and streamline the code base to minimize runtime overhead. Another big part of usability is ergonomics and familiarity; new users should be able to pick up the OpenMETA tools and immediately be productive, using minimal input to achieve maximum productivity. Vanderbilt put significant effort into accommodating requests from external testers, with the goal of making the adoption and integration process as seamless as possible, both for users and 3rd-party tool developers.

The OpenMETA design flow is predicated on the pre-existence of domain models (e.g., dynamics, 3D geometric, etc.), which can then be wrapped as multi-domain CyPhy *Component* models, enabling the user to leverage the OpenMETA capabilities. Parametric Modelica models are used within the component models to capture the dynamics behavior of components, which made it necessary to extract interface information from the Modelica models and wrap them in CyPhy, a process called *curation*. For the AVM FANG competition, the C2M2L[3] model libraries were used to create AVM Component Models (ACMs), which are AVM-specific CyPhy *Component* models. The complexity of the C2M2L Modelica package and the idiosyncrasies of the Modelica language itself made the manual curation process time-consuming and error-prone. To minimize the manual user interaction, the Modelica Importer (an OpenMETA tool) and the py_modelica_exporter (a custom Python module) were created. These reduced the time required to curate components by an order of magnitude, and standardized the resulting ACMs. These curation tools help future users of the tools to effectively import their existing Modelica libraries into the CyPhy environment as ACMs.

Once users had a working library of parametric ACMs, they needed to create component *instances*[4], which required copious amounts of tedious manual data entry. The Parameter Editor tool was created to facilitate this process and to enhance usability. The Parameter Editor presents all a Component's *Property* and *Parameter* objects in an alphabetically sorted table format, allowing the user to quickly find and edit the appropriate values.

After the user has created a system (or system design space) by composing a group of ACMs (either parametric components or instances), it is time to run analyses. For the

---

[3] C2M2L was another project within the AVM program with the goal of creating a library of multi-domain AVM Component Models, including Modelica, 3D geometric, and manufacturing models.

[4] Component instances are copies of the parametric models that have actual component data from manufacturers' datasheets

dynamics domain, the CyPhy2Modelica model translator is used to produce a composed Modelica system model, ready for execution. This was the first domain model translator to reach maturity; consequently, it was used more thoroughly, giving the developers the opportunity to quantify its shortcomings and re-implement it in a more robust and efficient manner. Improvements were made as follows:

- A structural model diagnostic utility is provided in the form of a structural model checker including 110 rules; prior to the model translation, the structure of the CyPhy model is evaluated against the checker rules. These rules include violations of Modelica syntax, in addition to semantic errors such as incompatible connections, self-connections, invalid value ranges, loops and type mismatches in value flows, etc. The checker results are summarized in the GME Console for the user based on the severity of the rule violations, along with hyperlinks to the culprit objects.
- As an additional option for pre-execution model diagnostics, generated Test Bench models can be checked for correctness using an external Modelica compiler, and the results are mapped back to the GME Console, with hyperlinks to the original source models.
- Templates are used when generating (Python) execution scripts for better maintainability, where string building was used in the first version
- Generated Modelica models are simplified, using minimal overhead code (hierarchical ‘wrappers’ were used in the first version; the new version leverages the object-oriented nature of the Modelica language using the ‘extend’ keyword). This makes the generated models more readable and easier to debug.
- Generated models comply with Modelica Language Specification 3.2
- A full Modelica package is generated, allowing the user to view all the associated models (including components, subsystems, and the full system), and sub-packages from a single entry-point.
- When CyPhy2Modelica is invoked as a standalone interpreter, it exports all component alternatives as part of the generated Modelica package.

The Master Interpreter is used in the setup and execution of almost all OpenMETA analyses. It has the ability to call one or more domain interpreters repeatedly, over an entire design space, minimizing the necessity for user interaction. It also can call cross-domain (i.e., domain independent) analysis interpreters, such as the Parametric Exploration Tool (PET) over a design space (or a user-defined set of design candidates). Due to its ubiquity in the OpenMETA design flow, and to its maturity, it was also selected for a complete overhaul. Improvements were made as follows:

- The *ICyPhyInterpreter* Interface was created, and all interpreters which should be called by the Master Interpreter were modified to implement this interface. *ICyPhyInterpreter* defines a common set of functions to interact with individual

interpreters such as show interpreter configurations to the user, save and retrieve the user entered configurations, define the output directory where artifacts have to be generated, and invoke the model transformation repeatedly in silent mode. This will also aid when integrating new analysis tools that require new model translator interpreters.

- A structural model checker was added, allowing the Master Interpreter to catch general modeling errors when it is called. The advantage of using such a checker becomes apparent with large design spaces: if there is a problem in the 'master' model template, it will propagate to all the models in the design space. Rather than generating hundreds or thousands of invalid models, the Master Interpreter will fail up front, assess the problems, and provide feedback to the user on the GME console, with hyperlinks to the problem models.
- The Master Interpreter can be invoked on large sets of models, calling other interpreters repeatedly. Since each interpreter (including the Master) has a non-zero runtime, this process can easily run into the 10s or 100s of seconds. For the re-write, a progress bar was added, giving the user a way to estimate the time to completion. Also, we added the option to cancel an invocation before it is completed, allowing users more control over their machine's resources.

When building system models from ACMs, designers have the option to use *ComponentReferences* in their system models rather than *ComponentInstances*. This reduces the memory and hard drive footprint for GME/CyPhy models. The design space exploration tool (DESERT) also utilizes these reference objects when generating design configurations from the discrete design space, for the same reason. Before domain-specific analysis interpreters run on a system design, the references must be replaced with component instances; the CyPhyElaboratorCS tool performs this task automatically. Since it is invoked on every single design configuration, it is crucial that it runs as quickly as possible. Also, its task is well-known and very straightforward, and it too was completely re-implemented with the primary goal of optimization.

The Job Manager is used to execute analysis packages after they have been generated by other interpreters. It can access the user's machine to execute jobs on idle CPU cores, and it can execute jobs on remote machines, which may be on a local area network or on another continent. Sometime after the Job Manager was implemented and had reached maturity, we discovered that some analysis jobs can take hours or even days to complete (in particular, finite element analysis). If such jobs are running remotely, it is likely that the network connection will be lost during execution (e.g., a power outage, the Job Manager application is closed, etc.). To address these limitations, the Job Manager was augmented with the following capabilities:

- Users can prioritize jobs. If a job is expected to consume CPU cycles for an extended period of time, users can prioritize jobs with the goal of maximizing resources
- When remote jobs are completed, the result artifact must be downloaded to the user's machine. Some results artifacts can be large, and take time to download. In the event that connectivity is lost during download, the Job Manager will attempt to resume/retry the download when connectivity is restored
- If the Job Manager application is closed while remote jobs are running (or the user's machine sleeps, or hibernates, or is shut down), it will sync long-running jobs from the server on restart/reboot.

5.4 Validation

To validate the improvement detailed above, we timed their performance on identical models. In the case of re-implementation, versions 1.0 and 2.0 existed side-by-side. We invoked each version on identical models, clocked the execution runtimes, and checked to ensure functionality was preserved or improved. In the case of added features, new functionality was thoroughly tested, both internally and externally, to ensure the improvement both addressed the relevant issue and did not introduce any new problems.

5.5 AVM Involvement

The OpenMETA tools have seen use by many different groups of users, including Vanderbilt University student interns, other AVM performers[5], AVM program beta/gamma testers, and FANG competitors. These groups provided useful feedback to the development team regarding bottlenecks in the design flow, enabling developers to target those critical components for optimization. Furthermore, as new tools were integrated with the toolchain and the size and scale of projects grew larger, unforeseen effects came to light.

5.6 Conclusions

Widespread usage allowed Vanderbilt developers to better assess the shortcomings in the OpenMETA toolchain. The feedback from users helped to prioritize the list of shortcomings. Items with the highest rates of usage received the highest priority, and significant time was allocated to categorize the required behavior and to re-implement these software components with optimization as a primary goal, in addition to preserving the full functionality. Finally, the lower priority items received the remainder of time, and were improved to the fullest extent possible. These changes improved the quality of the tools and increased the productivity of both developers and end users.

---

[5] Palo Alto Research Center (PARC), Stanford Research Institute (SRI), and Ricardo all developed analysis tools which were integrated into the OpenMETA toolchain.

6.0 WebGME Development

6.1 Summary

The baseline OpenMETA tools are based upon the Generic Modeling Environment (GME). GME is a meta-programmable, Windows-based desktop modeling environment that has been developed and used at Vanderbilt starting in 2000. While its flexibility and programmability is attractive for the language definition and evolution of the CyPhy language, a new editing environment will accelerate widespread deployment. The following features were specifically requested by users from the FANG competition and gamma testers:

- Collaboration: Desktop GME is single user. Complex systems involve multiple designers. Sharing a single model adds overhead to the modeling process.
- Multiplatform: Desktop GME is Windows-only. Support for Linux/Unix and tablet devices is desired.
- Usability and visualization: Desktop GME implements a rigid "boxes-and-lines" model representation. Improved editing and navigation of large models are needed, along with integrated visualization of simulation results.

The web-based version of the Generic Modeling Environment (WebGME[6]) is being developed in ongoing ISIS internal R&D efforts to address these needs. WebGME is implemented in JavaScript on the server and client side, and is cross-platform as it requires only a web browser. WebGME provides a web-based meta-modeling environment for any domain specific language and meta-model authoring. The meta-model and the associated instance models are stored within the same project. This approach helps users with accessing and authoring the meta- and instance models within the same environment and at the same time. WebGME allows multiple users to edit the models through real-time collaboration. All models are under version control and are stored in a persistent database using MongoDB.

In addition to the generic user interface, WebGME provides a set of Application Programming Interfaces (APIs) for the following workflows:

- implementing custom decorators and visualizers
- accessing (i.e. reading and writing) the data model
- implementing model transformations as plugins
- creating domain-specific user interfaces

[6] http://webgme.org/WebGMEWhitePaper.pdf

All APIs are provided in JavaScript; if developers would like only to read models, any programming language is supported through a RESTful API.

This effort led to the implementation of WebCyPhy, which includes a domain specific meta-model and a custom visualizer implemented using WebGME, coupled with the plugins needed to interpret and execute model analyses. This provides users with the ability to collaborate on model design, and maintains the OpenMETA Test Bench analysis functionality as a back end.

6.2 Objective

The OpenMETA toolchain saw extensive use and matured significantly throughout the AVM program. As the tools were exercised, it became apparent that there are limitations which cannot be addressed by simply improving existing features or adding new ones. One limitation is scalability: the functional size of a CyPhy project is limited by the fact that the mature OpenMETA tools are using the 32-bit version of GME. A second major limitation involves model portability and versioning: CyPhy projects (i.e., GME projects using the CyPhy Modeling Language or CyPhyML) are stored as binary files (.mga), so version tracking with Subversion or Git is impractical. Another significant limitation is the lack of efficient multi-user support. The simplest workflow for collaborative model editing was (1) each team member would get a copy of the model, (2) team members would work in parallel on separate machines, and (3) periodically, one team member would manually merge the models. For a variety of reasons, this process of parallel editing and periodic merging is tedious, time-consuming, and error-prone. These inherent shortcomings revealed the need to pursue drastically different avenues in order to find an effective resolution (as a temporary solution, a SubTreeMerge tool was implemented to address the model merging aspect of that issue).

The first step was to quantify, as best we could, the limitations. We accomplished this in a few different ways. For example, to determine the maximum size for a CyPhy project, we used a script to repeatedly create new objects within a CyPhy project until the 32-bit GME application ran out of memory and crashed. This revealed that for a typical developer's machine, ~10k AVM Component Models could be stored[7]. Throughout the AVM program, OpenMETA developers have created CyPhy models for testing purposes, some of them based on users' models (e.g., for bug fixes) and others built internally in the process of tool development and testing. OpenMETA developers are intimately familiar with the tedium involved with merging models, and in many cases were resigned to having a single person getting a 'lock' on the master model (using Subversion), making the necessary changes, committing the changes to the master copy, and then releasing the lock. The OpenMETA development team experienced much frustration with

[7] This limitation could become problematic when using component-based design techniques, where a typical system design could incorporate libraries with thousands of off-the-shelf component alternatives.

this limitation, despite the fact that building/editing models was required during only a fraction of the work week; this limitation on collaborative model development would have a much stronger effect on actual design teams who are end-users of the OpenMETA tools. Indeed, this frustration was echoed by beta and gamma testers, who strongly expressed the need for improvements in collaborative abilities.

As mentioned above, it is impossible to address these concerns by augmenting or adding to the existing GME or OpenMETA tools. Due in part to the recent ubiquity of internet access and the resulting improvements in web development frameworks/libraries, we decided to pursue a web-based approach. In work unrelated to AVM, ISIS Vanderbilt has been developing WebGME; it is functionally similar to GME, in that it supports custom meta-model definition for domain-specific applications, domain-specific instance model creation and editing, along with the ability to integrate existing external modeling techniques and analysis tools. However, it also provides solutions to the limitations listed above. It is cross-platform, in that it requires only a network connection and a web browser to create and edit models. Models are stored in a persistent Mongo database, which can be hosted locally (i.e., on the same machine as the WebGME server) or remotely. Multiple databases can be associated with a single WebGME server, and multiple WebGME servers can utilize a single database. A single Mongo database is capable of storing and managing terabytes of model content, which exceeds the desktop GME limitations by several orders of magnitude.

A single WebGME instance can serve multiple clients, allowing several users to access and edit the same project simultaneously, and WebGME tracks of all changes in the database. In terms of model analysis, we leverage the entire OpenMETA toolchain capabilities on the server side, since it is a mature and well-tested application.

6.3 Architecture

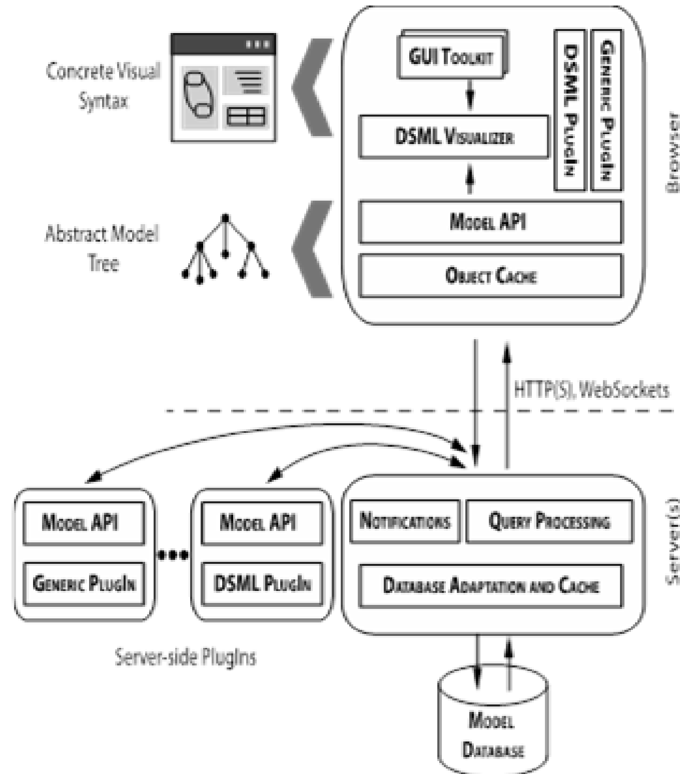

Figure 14: WebGME high-level architecture diagram (http://webgme.org/WebGMEWhitePaper.pdf)

WebGME is a server-client application, which consists of a web server and clients (e.g., web browsers) connecting to the web server. Using this server-client architecture makes cross-platform support easier, because modern web browsers are available on all major platforms, from desktops to mobile. Hence, using a web-based user interface (UI) can reduce the development time, can make the source code more portable, and can present a consistent look and feel for the entire application.

The web server provides access to the models using HTTP(S) and WebSocket (Secure) WS(S) connections through the model API, query processing, and notification services. The server is also capable of running plugins, and may have better performance profiles than the clients. Thus, transferring the models to the clients is unnecessary, decreasing the execution time in cases where the plugin's algorithm is complex or when it must process a significant amount of model content. The web server is implemented in JavaScript using a fast, easily scalable, non-blocking, event-driven platform NodeJS. The web server accesses the model database using TCP/IP connection. The models are stored as JSON documents in a NoSQL database MongoDB.

Programmatically traversing models is done through plugins written in JavaScript. These are the counterpart of interpreters in GME, with the exception that a plugin cannot receive input from the user beyond an initial configuration menu. Plugins are typically configured and invoked from the browser but can also be invoked within a standalone NodeJS process without the WebGME application running. When invoked from the browser any plugin can be configured to run either directly in the browser or on the server. Accessing nodes (i.e., models) is done through the model API granting access to the model storage database. Based on the context of the invocation, the root-node of the current project and branch are automatically loaded. From the root-node, the meta nodes, active node, and active selection are preloaded. A typical plugin execution loads, modifies, and adds nodes from this root-node as the model is traversed. The default 'save' method pushes all accumulated changes into one commit, and when possible, sets the head pointer of the branch to this commit.

Plugins have access to the BLOB storage through the BLOB Client and can load metadata and content based on hashes given from the plugin configuration or from *asset*-type attributes of model objects. Conversely, new files can be added using the BLOB Client and the generated hashes can be used to update the model or be returned to the user as a downloadable artifact in the result object. The option of running a plugin in either the browser or on the server puts certain constraints on the implementation of the plugins. For instance, template files need to be transformed into JavaScript files that can be loaded into the browser, and JavaScript code cannot rely on global objects (i.e., accessible from the browser only).

6.4 Data flow

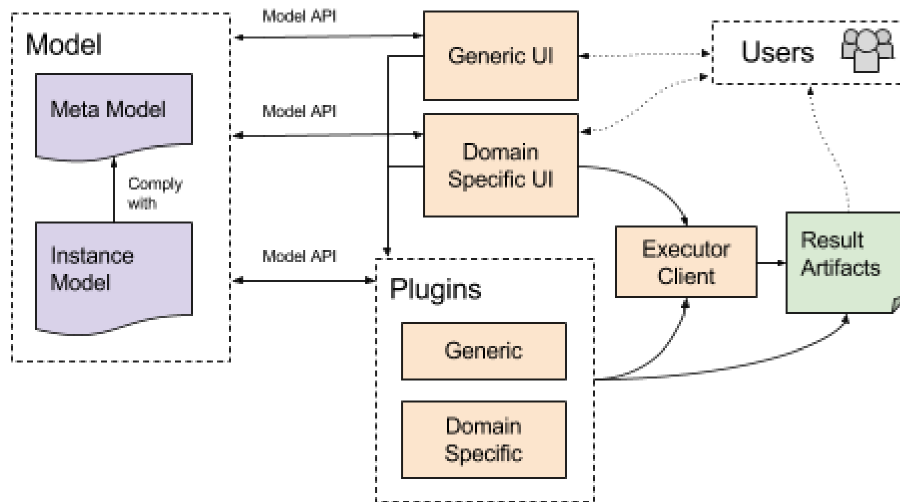

Figure 15: WebGME Data Flow

Figure 15 shows how users interact with WebGME models. All models, including meta-models and instance models, are version-controlled and stored in a persistent database. Users can access any model through a generic user interface provided by WebGME or a domain specific user interface provided by WebCyPhy. Both user interfaces are using the "Model API" to read and write the data models in the database. Users can invoke plugins (i.e., model transformations) on their models using both user interfaces. The plugins can be divided into two main categories: (a) generic plugins and (b) domain specific plugins. Generic plugins are implemented in such a way to work with any domain, whereas domain specific plugins are aware of many meta-model concepts and rules for a specific domain. Plugins may use the *Executor Client* API to analyze or run the generated code and transformed models. Generated code, models, and results are stored as user-accessible result artifacts in the WebGME environment.

6.5 Detailed Technical Description

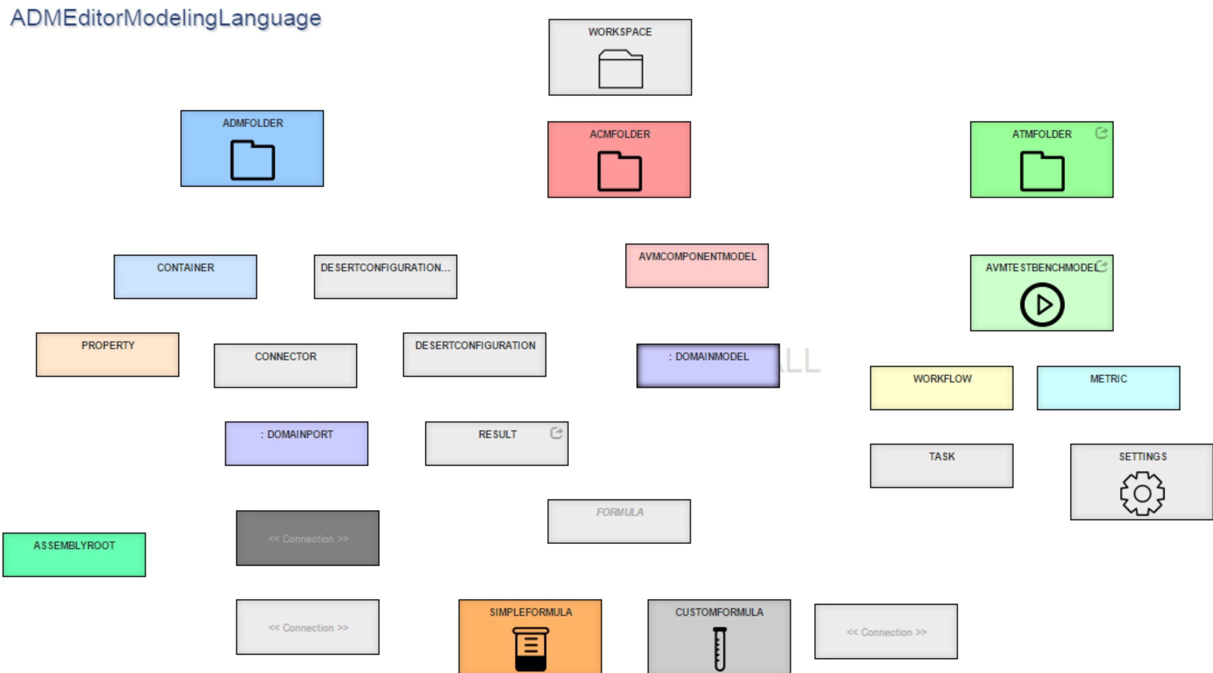

Figure 16: ADMEditor language concepts

The CyPhy language used by OpenMETA evolved significantly throughout the AVM program as new domains and their analysis tools were integrated into the toolchain. The end result is a complex meta-model, spanning multiple distinct domains and supporting a wide range of design/analysis techniques, both domain-specific (e.g. dynamics models targeting Modelica, and geometric models) and generic, i.e., domain independent (e.g., DESERT, PET). Because of this complexity, resource limitations prohibited a comprehensive re-implementation of the CyPhy meta-model in WebGME. Furthermore, the OpenMETA toolchain is a mature and well-tested software application which supports the desired design/analysis techniques and a large degree of automation, along with well-defined APIs. Consequently, we chose to implement a minimal meta-model for collaborative design editing in WebGME, targeting the AVM interchange formats (e.g., ACM[8], ADM[9]), and leveraging the desktop OpenMETA tools as an execution framework. Essentially, users can upload component and system design models to WebGME, create/edit designs (multiple users working in the same project, with version

[8] AVM Component Model, an XML file representation of a discrete component, describing its domain models and interfaces

[9] AVM Design Model, an XML file representation of a system design, describing internal design containers (e.g., Alternative or Optional), component instances, and interfaces

tracking in a network-accessible database), explore their design space using DESERT, set up Test Benches, and execute analyses. After execution, results are stored in the WebGME design models, allowing all team members to view existing results, with the goal of preventing redundant analysis execution.

This vision for WebCyPhy "1.0" as a collaborative design editor was the primary driver for the meta-model development process, resulting in the ADMEditor language. ADMEditor supports ACM import, ADM import/export, the ability to create new and edit existing ADMs, and execution of pre-existing CyPhy Test Benches (see Figure 16). Additionally, a domain specific user interface (WebCyPhy), was developed in order to incorporate context-specific visualization techniques. The following four sections describe the WebCyPhy implementation of AVM Component Models, AVM Design Models, AVM Test Bench Models and the domain specific user interface.

6.5.1 AVM Component Model

Components span multiple domains (e.g., structural, dynamics, cyber, manufacturing, etc.) and include the necessary model assets from each domain. Compositions of components form AVM Design Models. Components can be constructed in the CyPhy modeling environment by manually wrapping domain models targeting different analysis tools and extracting common parameters and interfaces. To facilitate the component library building process, a range of component authoring tools are provided by the OpenMETA toolchain (an example Component model is shown in Figure 17).

The Component Exporter tool exports a CyPhy component model into a single ACM zip package. All resource files associated with the component are included within the ACM zip-file. During the export process, the exporter tool ensures that violations of the format are either automatically fixed (e.g., blank ID attributes are filled) or reported to the user as warnings or errors (e.g., duplicate IDs, missing resource files).

Conversely, ACMs (i.e., zip packages) are imported to CyPhy using the Component Importer tool. The OpenMETA toolchain provides a command line utility to automate the ACM import process (e.g., for an entire component library). Within the context of WebCyPhy, these data flows and model transformations are used to programmatically build up the CyPhy models in the OpenMETA toolchain desktop environment during Test Bench executions.

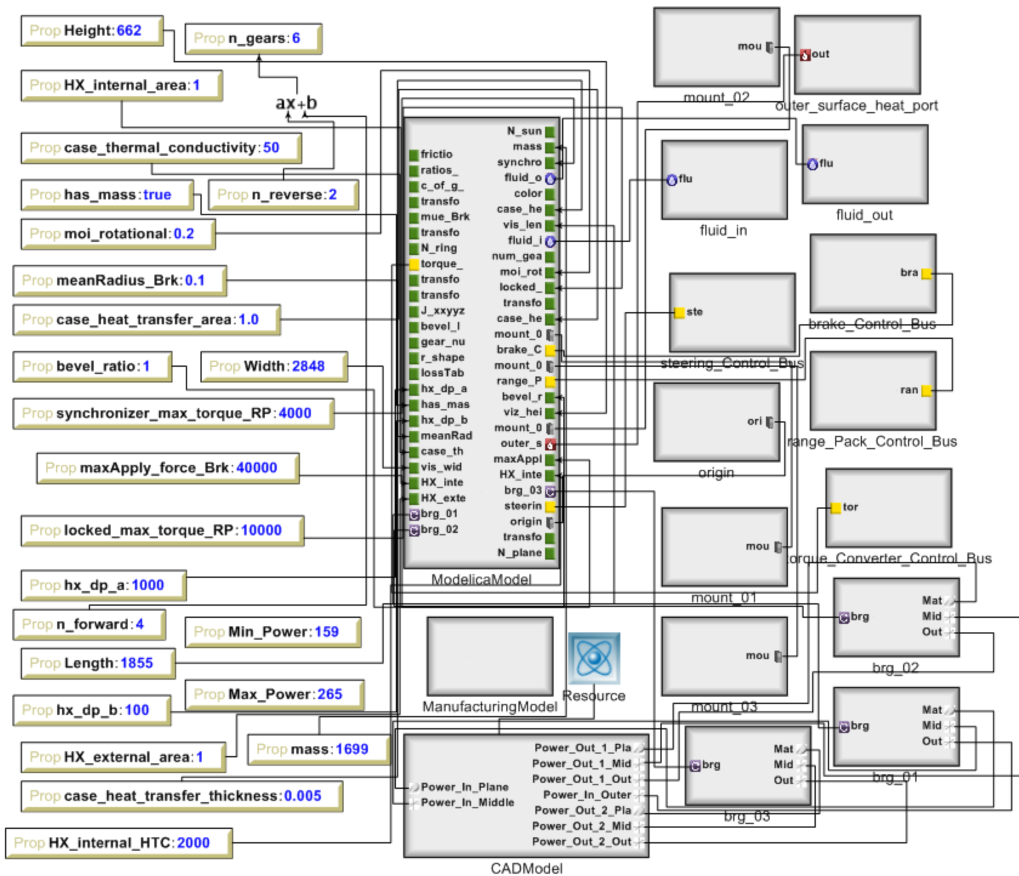

Figure 17: A Component representing a Cross Drive in CyPhyML in GME

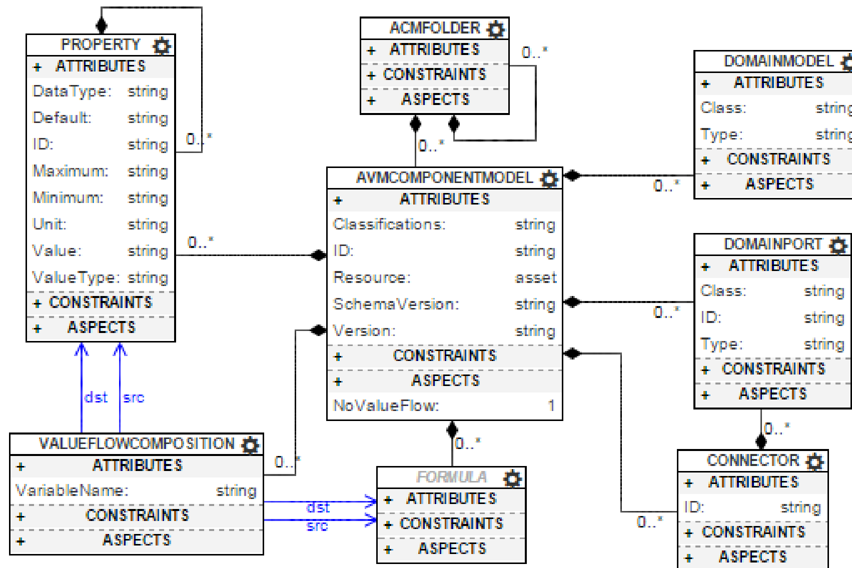

Figure 18: META model of AVM Component Model in ADMEditor in WebGME

In the current implementation of the ADMEditor language, ACMs are read-only models and the full content is not preserved directly in the WebCyPhy component model. Only the content necessary for composition and instantiation are stored as objects (e.g., properties, connectors, ports and domain-models). All other parts (e.g., resources and domain-model content) are stored as an asset in the WebGME artifact storage containing the original ACM zip package. This artifact is referenced from the component through the *Resource* asset attribute. ValueflowCompositions between properties are preserved in the model in order to distinguish between public and derived (i.e. internal) properties (see Figure 18). Importing components into WebCyPhy is done using the *ACM Importer* plugin. The configuration of the plugin takes one or more uploaded ACM zip-files. There is no plugin for exporting components from WebCyPhy. However, since the ACM is fully described by the artifact in the ACM's *Resource* attribute, the user can export the ACM by downloading the *Resource*. An example Component model in WebGME is shown in Figure 19).

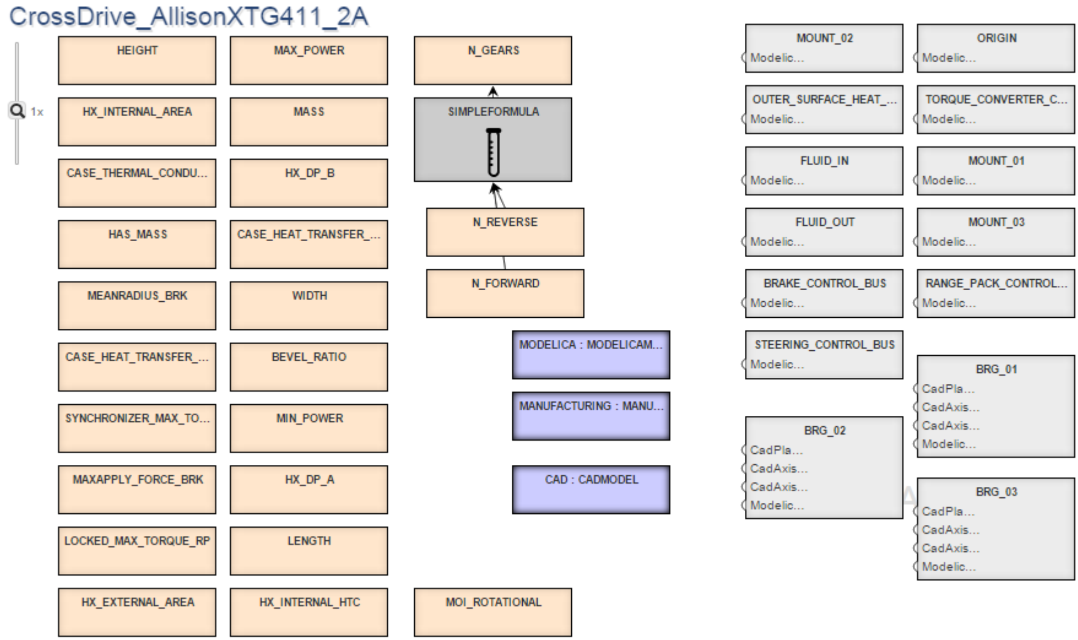

Figure 19: AVM Component Model in ADMEditor in WebGME

6.5.2 AVM Design Model

Designs are composed of AVM Component Models and can contain defined *Properties*, *Formulas*, *Connectors* and a hierarchy of *Containers*, in addition to value-flows and connections between components and subsystems. A Container in itself can be viewed as an AVM Design Model for the subsystem it describes. An ADM can describe either a single design point or a whole design space (see Figure 20). A single design point only contains *Compound Containers*, whereas a design space also contains *Alternative Containers.* In both cases, the description of an AVM Design Model is only complete when all of its referred ACMs exist in the same Project or *Workspace*, since the ADM contains only the ACM IDs and not the full ACM descriptions. The ADM Meta-model is shown in Figure 21.

An AVM Design Model in CyPhy can be exported as an ADM using the Design Exporter interpreter. During export, the interpreter ensures that violations of the format are either automatically fixed (e.g. blank ID attributes are filled) or reported to the user as errors or warnings. Conversely, importing AVM Design Models is done using the Design Importer. For a successful import, all referenced AVM Component Models must be available in the project.

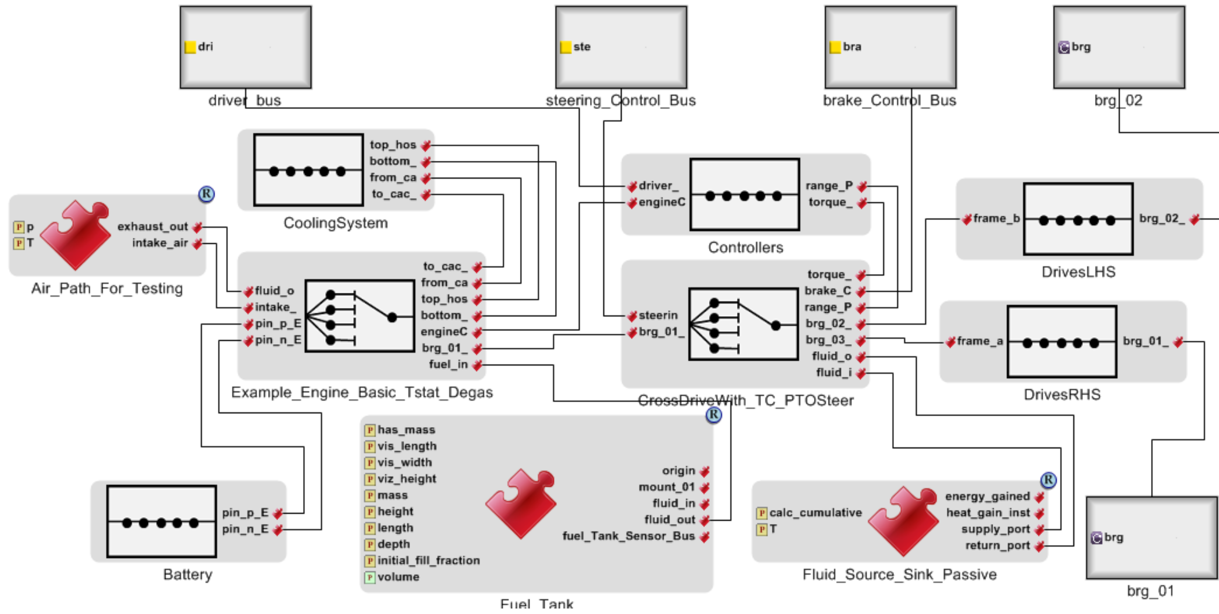

Figure 20: A Hierarchical Design-Space in CyPhyML. Shown are Connectors, Components, Compound and Alternative Containers.

In WebCyPhy the full content of an AVM Design Model is stored in the model and any changes are persistently saved in the database. The generic WebGME User Interface (UI) allows model editing in a similar fashion to that of GME. Invoking the ADM Exporter plugin on an AVM Design Model will create a new ADM (which can be imported back to CyPhy). Thus, in regard to design modeling, multiple users can develop their system design with all the collaborative benefits that WebGME offers (e.g., version control, platform independency, simultaneous editing, etc.). At the same time, the AVM interchange format allows such system designs to be evaluated by the analysis tools integrated within the OpenMETA suite.

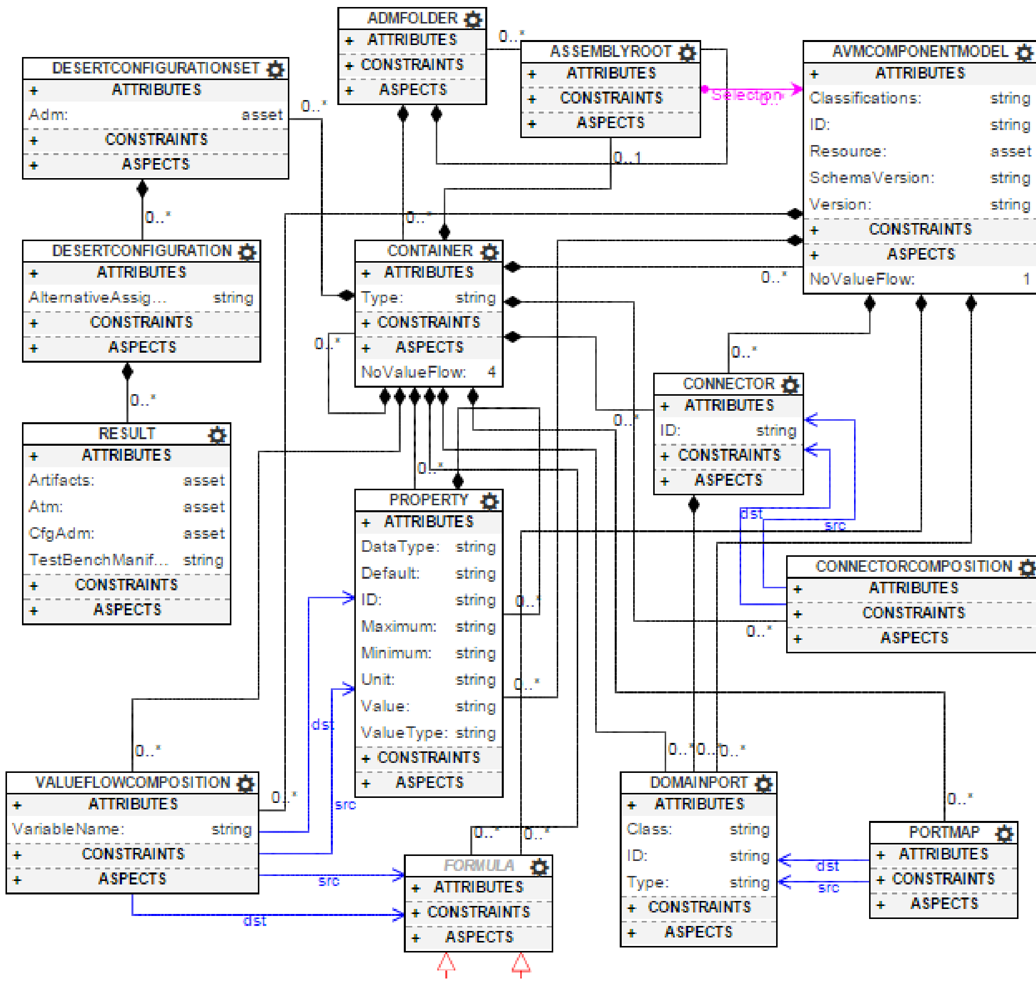

Figure 21: META model of AVM Design Model in ADMEditor in WebGME.

The *ADM Importer* plugin is used to import an ADM file into WebCyPhy. The configuration of the plugin takes an uploaded ADM file as input. To successfully import an AVM Design Model, all referenced components must already be present in the *Workspace*. The ADM Importer gathers the IDs of the components referenced in the ADM and checks if those components exist in the *Workspace*; if not, the missing components are reported as errors. Conversely, the *ADM Exporter* plugin exports an AVM Design Model from WebCyPhy to an ADM file. To ease the exchange process there is an option to package all the referred components together with the ADM file in a single artifact.

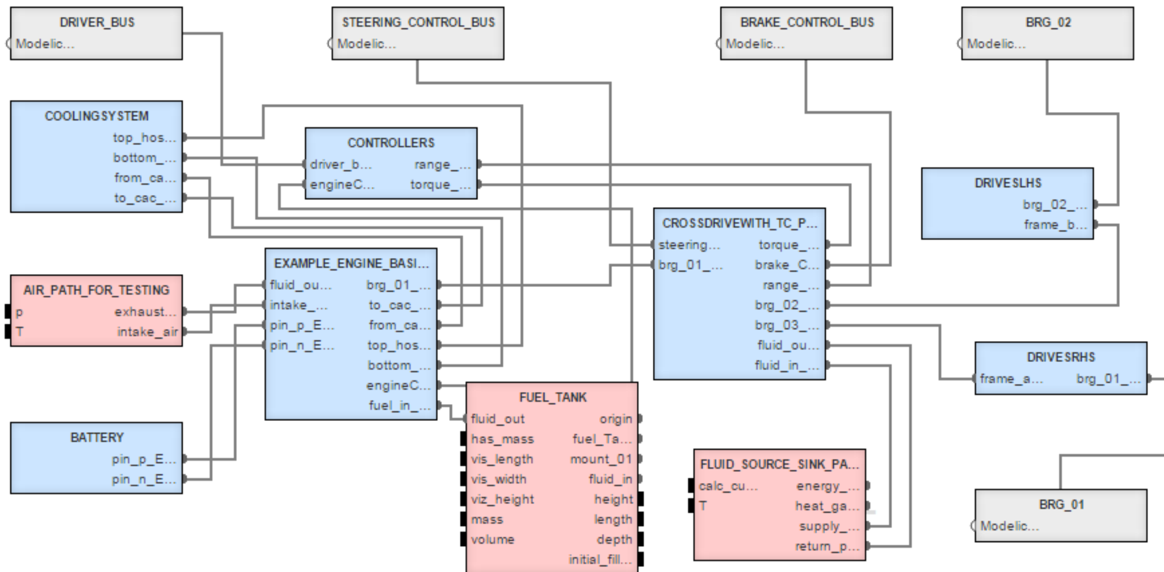

Figure 22: A Hierarchical Design-Space in ADMEditor. Shown are Connectors, Components and Containers.

The ADMEditor language currently supports unconstrained design space exploration, which relies on the DESERT tool. This aspect of the modeling process is implemented in a domain specific WebCyPhy UI (explained more in detail in the last section). When the user requests the configurations to be calculated for a design space, the design space is traversed and an input XML for the DESERT tool is created and saved to the BLOB storage together with execution instructions as an artifact. This artifact is used to create a job in the *Executor Client* (1. in Figure 23 below); the plugin periodically queries the *Executor Client* about the job's status (2. in Figure 23 below) and once the job has finished (3. in Figure 23 below) the DESERT output can be accessed and visualized in the browser (see Figure 24).

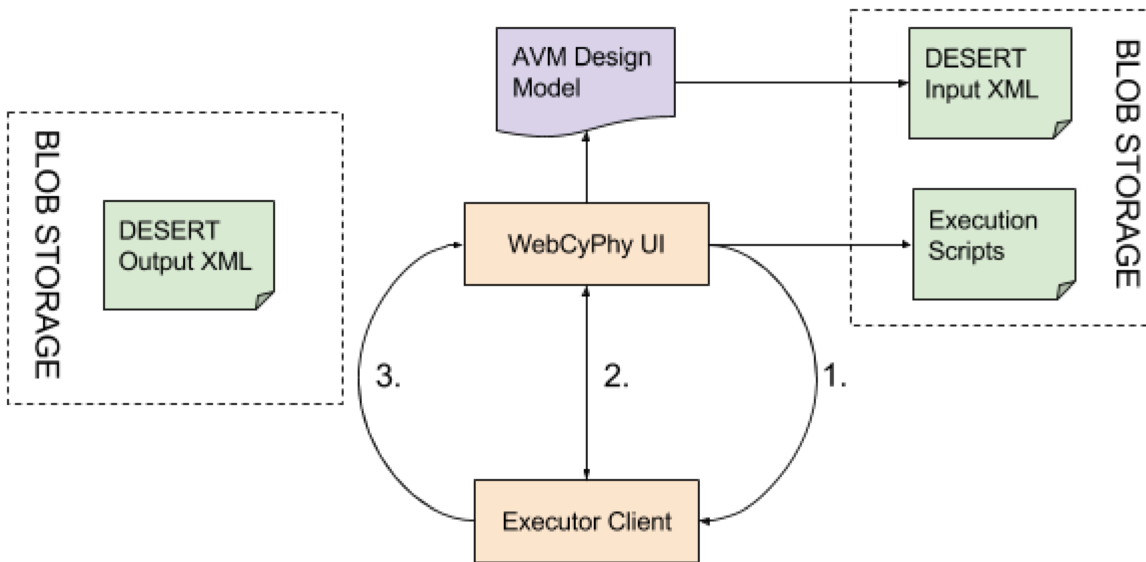

Figure 23: Data flow during Design Space Exploration from the Domain Specific WebCyPhy UI.

Generated configurations can be saved in the model as light-weight objects containing references to alternative selections. These configuration objects also serve as containers for *Result* objects, which store the artifacts generated from Test Bench executions.

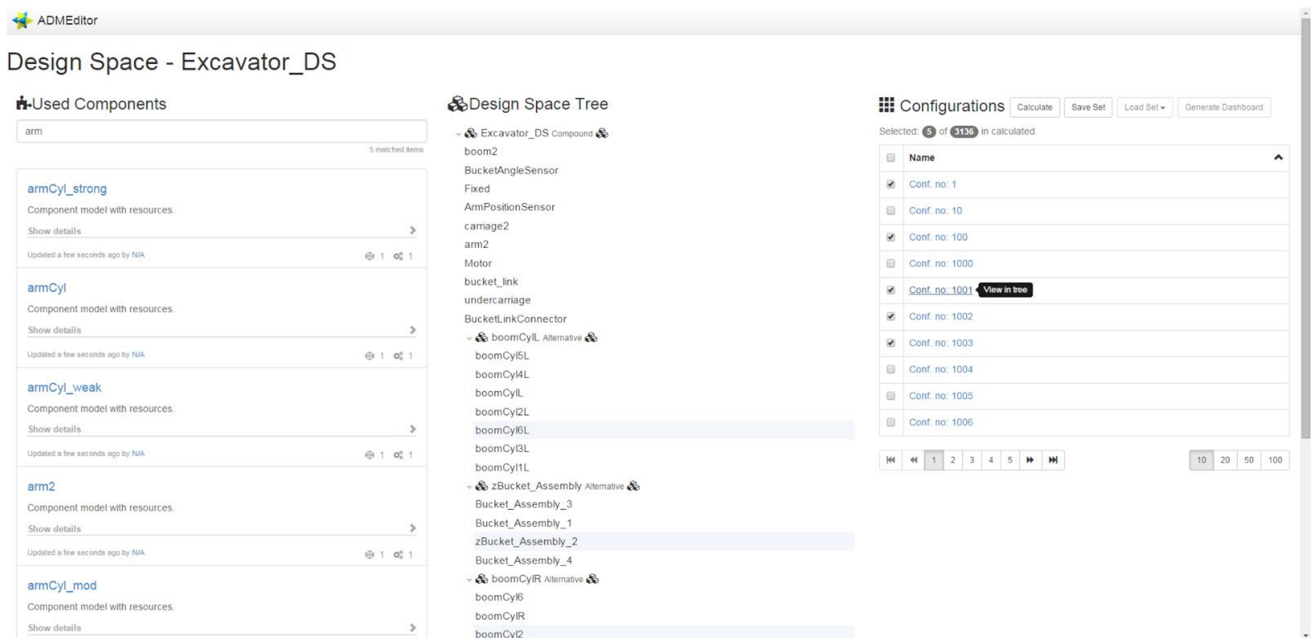

Figure 24: Design Space with generated Configurations visualized in the Domain Specific WebCyPhy UI

ISIS
Tel (615) 343-7472 Fax (615) 343-7440
1025 16th Avenue South Nashville, TN 37212
www.isis.vanderbilt.edu
VANDERBILT UNIVERSITY

The *Generate Dashboard* plugin was implemented to generate visualization packages for the Project Analyzer based on a set of design configurations. By default, the plugin searches for all result objects associated with the ADM, or the user can specify which result objects to visualize. From that set of result objects, the plugin generates all the necessary input data for the Project Analyzer. These input data files, together with the entire dashboard application are saved into a single artifact and exposed to the user.

6.5.3 AVM Test Bench Model

In the OpenMETA tools, Test Benches are the executable versions of the requirements and are used to evaluate system designs against specific requirements.

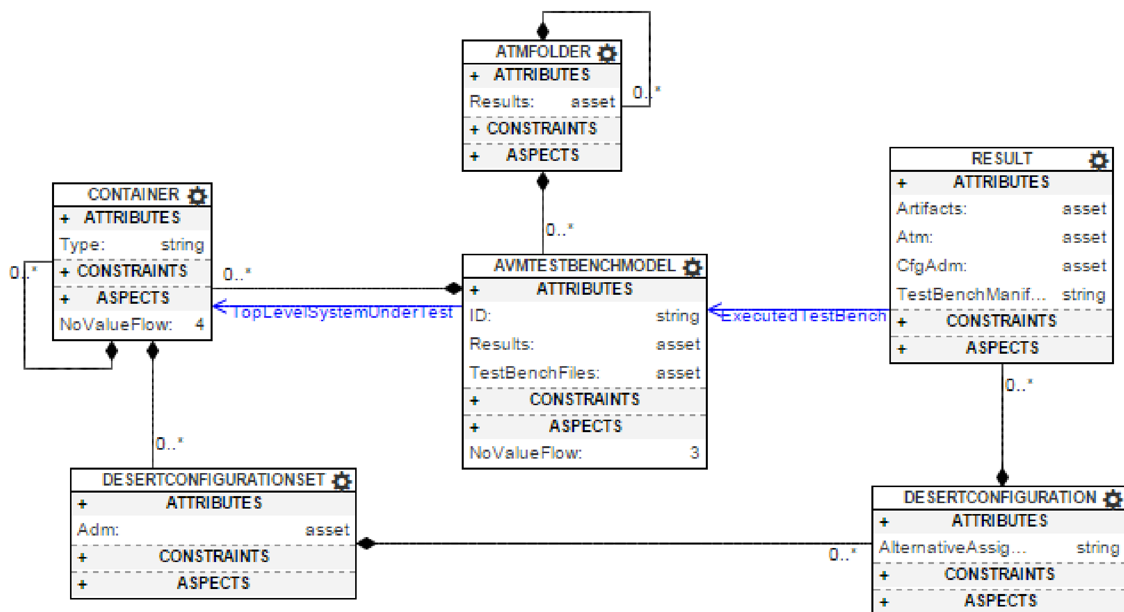

Figure 25: META model of AVM Test Bench Model in ADMEditor

Test Bench creation and editing is not fully supported in WebCyPhy. For the general case, users must export the AVM Design Model to an ADM file and import it into CyPhy, connect it in a Test Bench and execute it using the OpenMETA tools. However, in order to illustrate and exercise automatic Test Bench execution from WebCyPhy, a simple model of a Test Bench has been defined in the ADMEditor language. This approach is based on wrapping existing CyPhy Test Benches by attaching the xme (i.e., the GME/CyPhy project containing the Test Bench) file and related assets (e.g., post-processing scripts, configuration files, Modelica Libraries) to the *TestBenchFiles* asset attribute. A CyPhy Test Bench within the xme is referenced by providing its path in the WebCyPhy Test Bench's *ID* attribute. The xme file does not need to contain any AVM Component Models or detailed AVM Design Models, apart from a single 'dummy'

design defining the interfaces required to connect it in the Test Bench as the Top Level System Under Test (TLSUT). The dummy TLSUT, specified in the Test Bench of the xme file, will be replaced[10] with the actual AVM Design Model used as the Top Level System Under Test in WebCyPhy (see Figure 26 below).

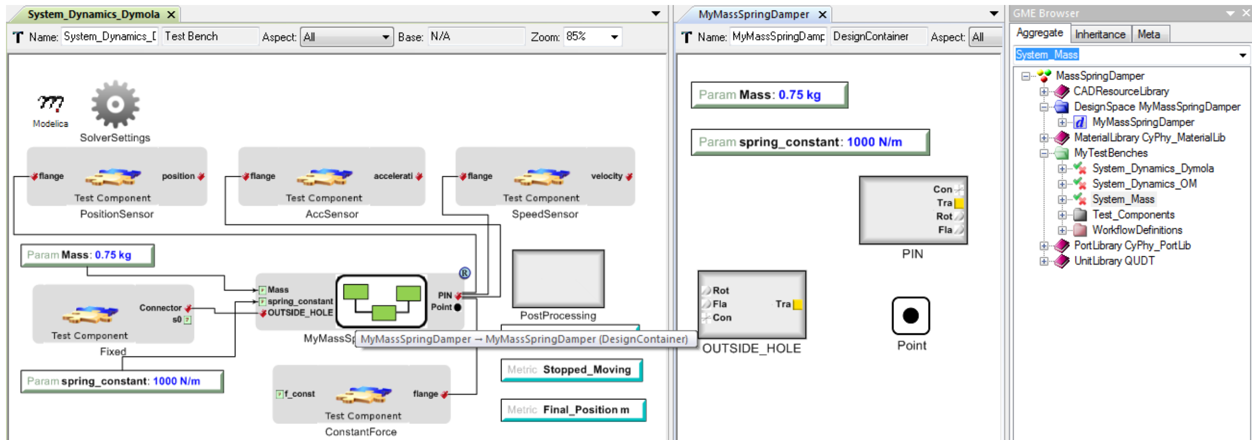

Figure 26: META model of AVM Test Bench Model with a Top Level System Under Test containing only the interfaces of an AVM Design Model.

When the user opens a Test Bench in the domain specific WebCyPhy UI, a list of all interface compatible AVM Design Models in the current workspace is displayed. Users can set the Top Level System Under Test and load any previously generated configuration sets for the chosen Design Space. If *Executor Workers* are available, selected configurations of the Design Space can be executed using the *Test Bench Runner* plugin (see Figure 27 and Figure 28 below showing the evolution of the Test Bench as the user makes selections).

[10] Switching the reference is done by the *Reference Switcher* interpreter, which rewires connections based on type and names. Implicitly this defines the interfaces an AVM Design Model must have in order to be evaluated against the Test Bench. To preserve this interface definition in the Test Bench of WebCyPhy, a Container containing the required interfaces can be placed within the Test Bench.

Figure 27: Selecting an AVM Design Model as Top Level System Under Test of a Test Bench (before).

Figure 28: Selecting an AVM Design Model as Top Level System Under Test of a Test Bench (after).

Invoking the *Test Bench Runner* will create an artifact on the BLOB storage containing the execution scripts and configurations for the job; the artifact from the *TestBenchFiles* attribute, an ADM[11] file for the AVM Design Model that is set as the Top Level System Under Test, and the ACM zip packages of the Components referenced by the ADM. This generated artifact (0. in Figure 29) contains all the necessary artifacts to run the analysis. It can be accessed as a downloadable zip package from the plugin result or it can be used to automatically create a job for the *Executor Client* directly from the *Test Bench Runner* plugin, (1. in Figure 29). The plugin checks the status of the job at specified intervals (2. in Figure 29) and once the status indicates that the job has completed, the plugin can access the job's result artifacts (3. in Figure 29). Each job is associated with a single design configuration, contained within the design space node. The job's result artifacts are saved in a new result object within the selected configuration node. The execution of a Test Bench can take several hours, and the users may continue editing models during execution. After the job is completed, the plugin checks the database revision history and stores the results to the latest version of the model. Running the *Test Bench Runner* on

[11] By specifying a configuration the ADM Exporter only adds the alternatives that are part of the selection defined by the configuration.

the server permits the browser which initiated the execution to be closed; the result will always be stored in the model as explained above.

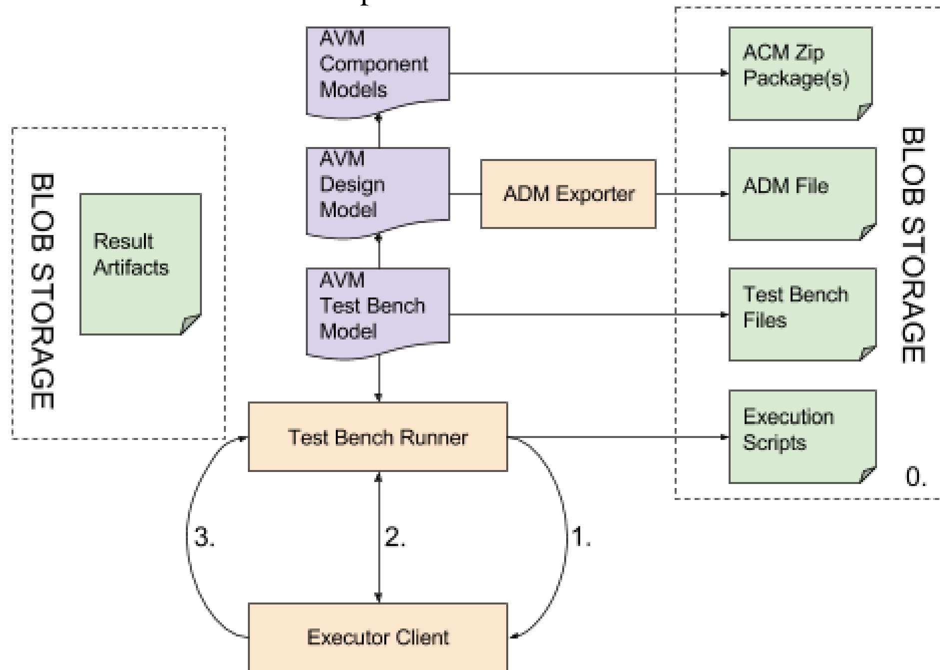

Figure 29: Input Artifacts generated by the Test Bench Runner.

6.5.4 Domain Specific User Interface

The generic WebGME UI allows model editing in a similar fashion to that of GME. A user can create hierarchical models, edit attributes, define relations (pointers and sets), etc. either from the view of a canvas ("boxes-and-lines") or a containment tree structure (*Object Browser*). While the above is a powerful approach for modeling hierarchical cyber physical systems, some aspects of the entire design process might be visualized more naturally using other techniques.

- As projects grow bigger and more users have access to them, it is desirable to present the project and *Workspaces* in a concise and meaningful way.
- Although manual editing and plugin invocations cover many uses cases, none is well suited for programmatic changes based on user input and model content. In WebCyPhy, examples of such cases are design space exploration and Test Bench execution over a design space.

To enable such customizations a web application was implemented using AngularJS[12]. It acts as a parallel (and complementary) view to the generic WebGME UI and uses the same model API for accessing and writing to the model. To ease the development of domain specific user interfaces, a distribution build process was added to WebGME. The required JavaScript classes, the *Core Client* for accessing nodes, the *BLOB Client* for BLOB access, and the *Plugin Manager* for executing plugins are bundled into a single JavaScript file. Once this JS file is included in a new project, it exposes all these classes from a global JavaScript object in browser. From the domain specific application this global object can be extended with the domain specific plugins. Since the WebCyPhy UI is an AngularJS application all these classes are implemented as AngularJS services (singletons containing functionalities accessible across the web application), making use of the built-in capabilities of AngularJS, particularly the use of promises for asynchronously loading nodes using the *Core Client*. The AngularJS services, specifically the *Node Service* implementing the *Core Client*, were also extended with additional utility functions, such as determining the meta type of a node, loading the direct children of a node, and attaching event handlers directly to a node.

The design space exploration and Test Bench execution are presented in previous sections; below, the WebCyPhy *Workspace* management and visualization capabilities are presented (see Figure 30). Visiting the start (or landing) page of the application displays a listing of all *Workspaces* of the ADMEditor project. Live statistics are also displayed to the user, including number of Components, Design Spaces and Test Benches within each *Workspace*, as well as actions for each *Workspace* (e.g., rename, delete, etc.).

[12] https://angularjs.org/

Figure 30:Listing of all available workspaces

When creating a new *Workspace*, a single button click will invoke all required importers in the right order after the resources have been added to the form (in Figure 31, the user has added ACM and ADM files; clicking 'Create' will run the ACM Importer and then the ADM Importer).

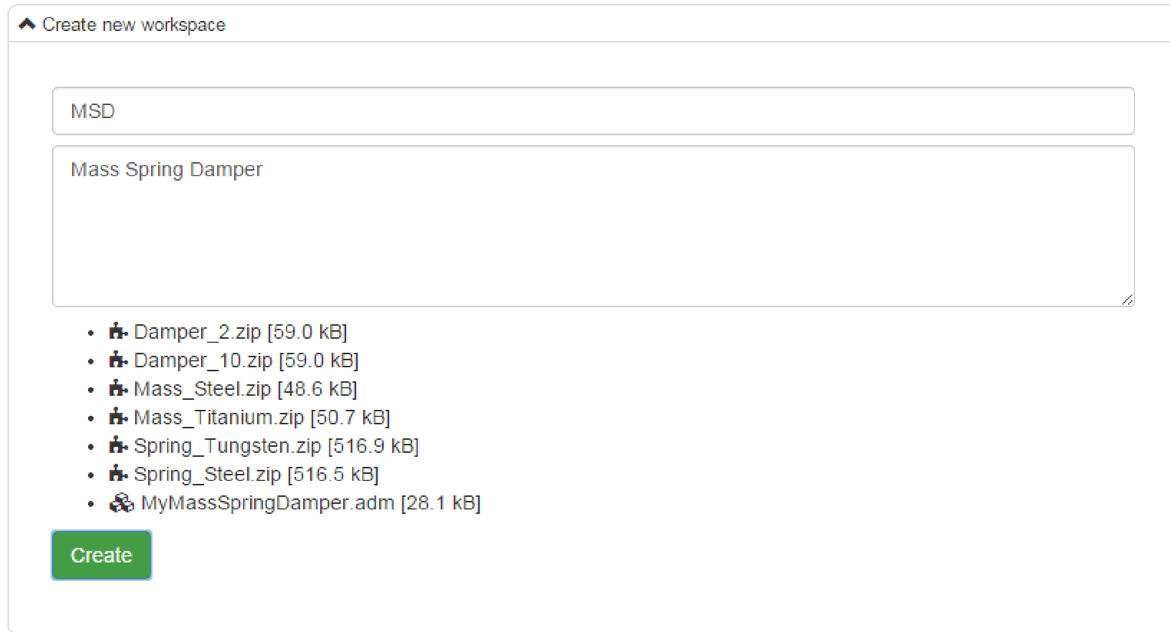

Figure 31: Creating a workspace

Navigating inside of a *Workspace* will bring up details about its content in terms of Component, Design Spaces and Test Benches (see Figure 32) . Each list provides a filter for quick navigation and the list items all have actions available through a drop down menu in the top right corner. These actions include options such as opening up the object in the WebGME UI, editing attributes, exporting to the ACM or ADM, deleting, etc. based on the object type. Statistics about domain models, number of configurations and results are also available where applicable.

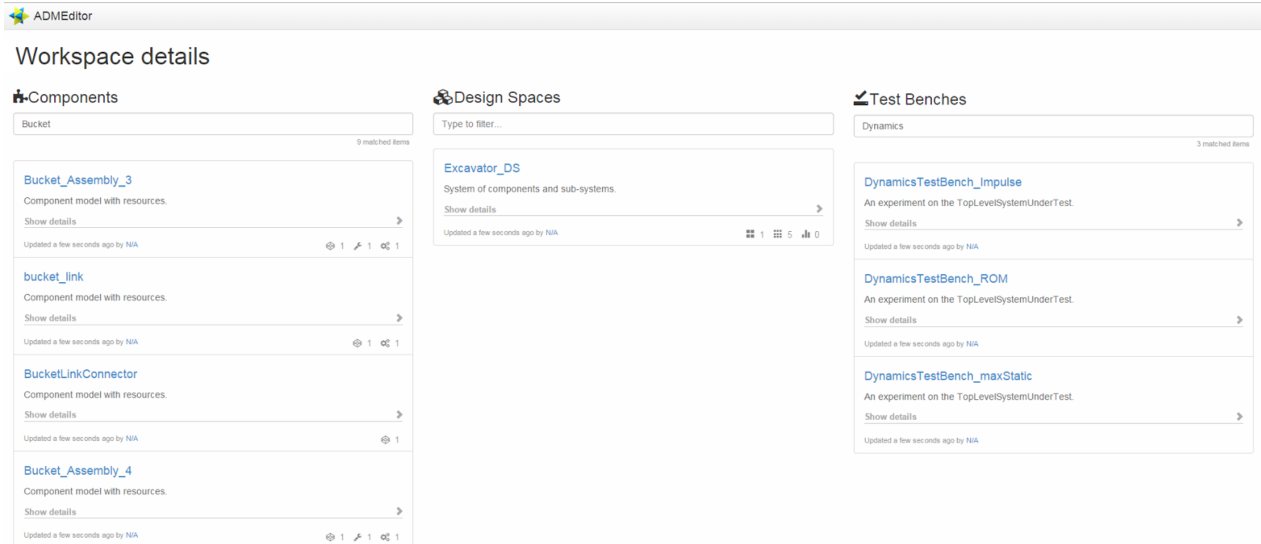

Figure 32: Workspace details showing available Components, Design Spaces and Test Benches

6.6 Future Enhancements

The current implementation can be extended with Component authoring and Test Bench authoring capabilities. Both authoring capabilities are required to reduce the direct dependency on the desktop tools. All models are version controlled and WebGME supports branching of the model, but the WebCyPhy user interface currently displays only the master branch. The WebCyPhy user interface can be extended with the support of selecting branches, merging, and visualizing diffs or conflicts during merges.

The binary large object storage (BLOB) should be fully implemented and tested with S3 Amazon or private cloud storage backends. There are limitations in the maximum working size of BLOB complex objects (i.e., hierarchical objects such as a zip file with a folder structure). The size limitation (1GB) has not yet been problematic, but we should test the limitations to make future development easier and more robust.

6.7 AVM Involvement

The collaborative and version control capabilities of WebGME were tested by alpha and beta testers. WebGME and WebCyPhy were demonstrated to gamma testers who had pointed out the limitations of the OpenMETA desktop tools; these design teams found useful alternative workflows in WebCyPhy. It is possible that some workflows and some part of the web-based tools will be adopted by the gamma testers in the future.

6.8 Conclusions

The Windows implementation of the CyPhy Modeling Language coupled with the OpenMETA toolchain is a robust and valuable modeling paradigm. It was tested thoroughly during the 4-year AVM program, and while its value was noted, its limitations revealed the need to explore new avenues for future development. WebCyPhy addresses critical limitations, and provides the ability to leverage the desktop versions of the tools, allowing users to build upon previous work in contrast with starting from scratch.

WebCyPhy is an extensible and flexible application, and improves on CyPhy/OpenMETA capabilities in several ways. It gives users the ability to collaborate on model editing in real-time, along with a central model storage scheme, minimizing the headache of merging parallel work. Utilizing a central database betters the desktop model storage capabilities by several orders of magnitude. WebCyPhy provides a domain-specific visualization platform, which is invaluable in cases where a boxes-and-lines schematic viewer fails to communicate the model content in a meaningful way. It is platform independent, and users can access their models over a network connection and remotely configure/execute analyses from a variety of internet-capable devices, including touch screen tablets.