RT 117: Development and Application of the Framework for Assessing Cost and Technology (FACT) to Support USMC Ground Vehicle Design Analysis

Technical Report No. SERC-2015-TR-017-117

June 12, 2015

Investigators
Principal Investigator: Daniel C. Browne, Georgia Institute of Technology

Research Team
Santiago Balestrini-Robinson, PhD
Melissa Rost
Thomas Mark
Todd Shayler
Bryan Ake
Joshua Eppinette
Jordan Carroll
Dane Freeman, PhD
Devraj Mehta
Robert Kempf
Blaine Laughlin
Cusuh Ham
Giuseppe Scoppino
Timothy VanHeest

Sponsor: Marine Corps Systems Command – FACT PM
Copyright © 2015 Stevens Institute of Technology, Systems Engineering Research Center

The Systems Engineering Research Center (SERC) is a federally funded University Affiliated Research Center managed by Stevens Institute of Technology.

This material is based upon work supported, in whole or in part, by the U.S. Department of Defense through the Office of the Assistant Secretary of Defense for Research and Engineering (ASD(R&E)) under Contract H98230-08-D-0171 (Task Order 0017).

Any views, opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense nor ASD(R&E).

No Warranty. 
This Stevens Institute of Technology and Systems Engineering Research Center material is furnished on an “as-is” basis. Stevens Institute of Technology makes no warranties of any kind, either expressed or implied, as to any matter including, but not limited to, warranty of fitness for purpose or merchantability, exclusivity, or results obtained from use of the material. Stevens Institute of Technology does not make any warranty of any kind with respect to freedom from patent, trademark, or copyright infringement.

This material has been approved for public release and unlimited distribution.
Table of Contents

Table of Contents ........................................................................................................................................... iii
Table of Figures ............................................................................................................................................... v
Table of Tables .............................................................................................................................................. vi
Background and Objectives .......................................................................................................................... 2
Overview .......................................................................................................................................................... 4
   TASK 1: Explore Modeling and Simulation Capabilities Available for Evaluating Solutions Against the
   Requirement Set .................................................................................................................................................. 4
   TASK 2: Support Technical Exchange Meetings .................................................................................... 4
   TASK 3: Update Data/Models in FACT to support Tool Use .................................................................... 4
   TASK 4: FACT Toolset Improvements .................................................................................................... 4
   TASK 5: Explore and Implement Visualization Techniques ...................................................................... 4
   TASK 6: FACT Hosting ................................................................................................................................ 5
Synergy with Engineered Resilient Systems and RT-120 ............................................................................ 6
FACT Architecture Redesign .......................................................................................................................... 8
   Overview ....................................................................................................................................................... 8
      How did we get here? ................................................................................................................................. 8
      Architecture Redesign Goals ................................................................................................................ 8
      Architecture Redesign Goals Status ...................................................................................................... 9
Data Redesign and REST API ....................................................................................................................... 11
   API Requirements ..................................................................................................................................... 11
   API Status .................................................................................................................................................... 12
OpenMDAO and OpenModelica ................................................................................................................... 15
FACT 2.0 Point Solution Capability .............................................................................................................. 17
   What is it? .................................................................................................................................................... 17
   Inspiration: Apply lessons learned from FACT 1.x ................................................................................. 17
Capability Overview ........................................................................................................................................ 18
      System Tree ............................................................................................................................................... 18
      Attribute Modification ............................................................................................................................. 19
      Requirements Satisfaction ....................................................................................................................... 19
      Requirements Treemap .......................................................................................................................... 19
      Requirement Bar Chart .......................................................................................................................... 20
      Diverging Bar Chart ............................................................................................................................... 21
      Requirement Progress Bar ..................................................................................................................... 21
Supporting Libraries ..................................................................................................................................... 22
Data Analytics Pipeline: Tradespace Field of Dreams ................................................................................. 23
   Inspiration: Why was TFOD built with FACT 2.0? .................................................................................. 23
TFOD Data Model .......................................................................................................................................... 25
   Block Operation Types .............................................................................................................................. 26
The TFOD Interface ......................................................................................................................................... 27
   A User Story ............................................................................................................................................... 27
      Building a Pipeline .................................................................................................................................. 28
TFOD Next Steps .......................................................................................................................................... 32
TABLE OF FIGURES

Figure 1 The Systems Engineering workflow. ................................................................. 6
Figure 2 FACT 2.0 and ERS TRADESPACE, though separate capabilities, shares some common components and architecture, enabling sharing of capabilities across the projects. ........................................ 7
Figure 3 Applications of the modular architecture ..................................................................... 12
Figure 4 The FACT 1.x Point Solution was designed with the user in mind to support the primary use case of understanding how well a point design meets or fails to meet a requirement set and furthermore, how it compares to specific baseline..................................................................................................................... 17
Figure 5 FACT 1.0 Point Solution requirements analysis ................................................................. 18
Figure 6 The tree responsible for navigating the baseline systems components......................... 18
Figure 7 Attribute modification pane corresponding to the selection in Figure 3 ................................. 19
Figure 8 High-level requirements overview displaying many requirement categories. The colors displayed by the requirement categories are a combination of the colors displayed by the children of that requirement category. ............................................................................................................. 20
Figure 9 Drilled-down view of one requirement category with four quantified requirements .................. 20
Figure 10 Baseline requirement numbers .......................................................................................... 21
Figure 11 Bar chart after generating point solution. Baseline values are represented by thicker opaque bar .. 21
Figure 12 Diverging bar chart showing percent change from baseline to point solution. ......................... 21
Figure 13: The “ferry range” progress bar appears after clicking the "ferry range" rectangle in Figure 6. ...... 22
Figure 14 Example of the interactive analysis tool used to support decision-making. The left provides a cost vs. mass margin scatterplot while the right shows histograms of the selected parts for each subsystem. In this example image, the last part in the 2.01.01.01.02 WBS element has been clicked and selected, filter out (displaying them in red) all of the options that did not utilize that specific part......................................................... 24
Figure 15 Data process used by ACV Feasibility Study in 2013 to create variants within a tradespace. .......... 25
Figure 16 Architecture of the web-based SysML analysis framework .................................................. 34
Table of Tables

Table 1. OpenMDAO components developed for OpenModelica using OMPython ........................................... 15
Table 2. Hierarchy of block types implemented within TFOD. ............................................................................. 27
Table 3. TFOD offers six different states to define the status of a specific block in a pipeline. ......................... 30
This page intentionally left blank
The Marine Corps Systems Command (MARCORSYSCOM) has been leading the development of the Framework for Assessing Cost and Technology (FACT) since the spring of 2011. FACT is designed to be a comprehensive Model-Based Systems Engineering (MBSE) environment to support DoD acquisition. FACT offers a means to capture a solution architecture, define design variables and performance parameters of interest, integrate models and/or simulations, execute trade studies, and analyze the results. All of these capabilities are offered in a collaborative web interface. The Georgia Tech Research Institute (GTRI) has been the primary developer of FACT.

A specific project of note, which utilized FACT, was the Amphibious Combat Vehicle (ACV) Feasibility Study conducted in 2013. Through using FACT, the collaborative Government, Industry, and Academic study team (to include GTRI) was able to explore nearly one million different variants to understand the implications of design choices on cost.

Over the course of 3 years, the complete FACT feature set was pieced together through supporting various efforts. Each effort built upon the existing capability new functions and features to meet a specific problem’s needs. Throughout this report, this version of FACT will be referred to as “FACT 1.x”. As this occurred, it was realized that FACT’s roots were a monolithic software structure as opposed to a modular, extensible architecture.

After the success of FACT, and from the lessons learned over the three years of development, the effort summarized in this report was initiated to re-architect FACT into a more modular and extensible architecture, offering an industry standard REST API. Re-factoring FACT’s capabilities into this architecture would allow Government and Contractor developers to build upon the FACT base with a greatly reduced learning curve, easing adoption of the framework. Throughout this report, this newly architected FACT will be referred to as “FACT 2.0”.

To this end, the following specific tasks were initially laid out to guide the effort. The tasks were evaluated and updated as needed throughout the execution to meet the needs of the FACT 2.0 development.

1. **Explore Modeling and Simulation Capabilities Available for Evaluating Solutions Against the Requirement Set.** First, GTRI will examine the defined requirement set and determine what modeling and simulation capabilities, which could be leveraged to evaluate a design against the requirement. GTRI will collaborate with Government Subject Matter Experts (SMEs) on tool availability and applicability to each requirement. This set of tools informs the packages which will need to be integrated into FACT. A specific program was not selected during the execution of this research topic; therefore the work related to this task was conducted in a general sense based on past experience.

2. **Support Technical Exchange Meetings.** GTRI will be prepared support regular (approximately weekly) Technical Exchange Meetings with the Government analysis team. These meetings will allow GTRI to provide status updates to Government stakeholders and ensure the GTRI team is providing the appropriate support.

3. **Update Data/Models in FACT to support Tool Use.** GTRI will provide support in updating the Program-specific instance of FACT with appropriate data and models. Task 1, to ensure the appropriate modeling, informs this task and simulation tools are integrated within the framework. Configuration management of data within the Framework will be conducted to allow the Government to understand the provenance of design decisions. This task also includes GTRI documenting and (semi-) automating the process on bringing new applications into FACT. Completing this work is necessary to ensure the Government fully understands how to utilize FACT for future work. Additionally, GTRI will develop and execute training
curriculum and materials to enable Government engineers and analysts to utilize the systems engineering toolset directly.

4. **FACT Toolset Improvements.** GTRI will improve the FACT toolset to support execution of a design process using sound model-based systems engineering principles. GTRI will coordinate with MARCORSYSCOM on specific updates. Improvements include (1) expanding the tradespace capability with visualized data pipelines and automated parallel execution; (2) strengthening the representation of requirements and implementing a means to relate requirements to higher-level desired capabilities, including definition of utility functions; and (3) expanding the means to represent part/component interoperability to include insight into groups of potential component sets via graph analysis of the interoperability graph.

5. **Explore and Implement Visualization Techniques.** GTRI will explore different means by which to visualize large amounts of design data. GTRI will implement various visualization techniques and iterate with MARCORSYSCOM decision makers in order to evaluate the effectiveness of each technique. This task will deliver a presentation or report explaining the techniques and providing examples. Some of the techniques will be implemented in the delivered toolset.

6. **FACT Hosting.** GTRI will either support MARCORSYSCOM through direct hosting of FACT instances or work with Government hosting facilities to ensure the proper computational resources are available. SERC will conduct analysis on the expected loads and performance/time requirements from the Government team to size computational resource requirements properly.
**Overview**

Details on the work completed for each of the above six tasks are provided throughout this report. To begin, a synopsis of the work completed under each task is offered.

**TASK 1: Explore Modeling and Simulation Capabilities Available for Evaluating Solutions Against the Requirement Set**

FACT is itself a Framework, but in order for a specific Program to leverage the full set of capabilities offered by FACT, a suite of models needs to be identified and integrated. In this task, researchers completed two separate efforts. First, modeling and simulation integration and execution frameworks were examined to replace the “home grown” capability built in FACT 1.x. The FACT 1.x capability offered the basic needed functions, but was limited in its feature set compared to some open source tools identified by the researchers since FACT’s inception. The second effort was to examine some specific modeling tools of interest and determine strategies for integrating these to make them reusable for multiple Programs.

**TASK 2: Support Technical Exchange Meetings**

In order to ensure that the research team adequately understood Government priorities and needs and progress were relayed to the Government, technical exchange meetings were held at the Government’s discretion. The research team offered updates on technology selection for the updated architecture stack and progress on the FACT 2.0 build.

**TASK 3: Update Data/Models in FACT to Support Tool Use**

Originally, the research team planned to support a specific Program while FACT 2.0 was being built, to allow feedback from the Government users into the development process. A specific Program was not identified during execution of this task; therefore to meet the intent of this task, new methods for getting data and models in FACT 2.0 were developed and documented. Specifically, a “super user” interface leveraging IPython notebook is included in FACT 2.0 with example notebooks of creating a problem in FACT and populating data. Additionally, FACT 2.0 is leveraging NASA’s Open-source Multi-disciplinary Design Analysis and Optimization (OpenMDAO) framework for model integration and execution. OpenMDAO offers a more comprehensive execution capability than FACT 1.x and has well-defined data structures and documentation that FACT 2.0 developers can now leverage.

**TASK 4: FACT Toolset Improvements**

Each of the front-end capabilities offered by FACT 1.x were rethought, starting with the initial Use Case. Not all of the FACT 1.x capabilities could be rebuilt in this research effort, so specific capabilities were redesigned and implemented. Of specific note is the new Point Solution capability. In FACT 1.x, users were able to build a single variant for a defined architecture and compare its cost, performance, etc. to defined requirements. FACT 2.0 extends this to allow a user to additionally compare the defined point to a selected baseline in order to understand how the design is improved or deficient compared to the selected baseline. Additionally, the interface has been redesigned making it easier for users to build a palette of variables of interest and conduct a series of what-if scenarios without needing to jump around the work breakdown structure tree.

**TASK 5: Explore and Implement Visualization Techniques**

With each Program supported by FACT, a custom front-end capability and/or visualization was required in order to best support decision making. Some new visualizations were implemented in the overall FACT Toolset Improvements. Since a specific Program was not selected to be supported by FACT 2.0 under this research effort, the researchers explored the development of a customizable data-pipelining tool. When working with the ACV...
Study Team in 2013, GTRI developed a data pipeline in order to execute all of the trade studies of interest. However, many parts of the data pipeline were manual and the automation came through scripts executed from a command line or SSH shell. To enable FACT 2.0 users the ability to build their own data pipelines, a modular and extensible capability with a web-based graphical user interface (GUI) needed to be developed. To this end, researchers developed the “Tradespace Field of Dreams” or “TFOD” capability as standalone capability that is also being integrated into FACT 2.0. Reference the later Data Analytics Pipeline: Tradespace Field of Dreams section on details of this effort.

**TASK 6: FACT HOSTING**

In this effort, the SERC (through GTRI) was prepared to offer a hosting of FACT 2.0 for a specific Program. Though a specific Program was not selected to be supported under this effort, the researchers improved the technologies utilized to manage the development and hosting environment, easing spool-up time and effort for FACT 2.0 developers. FACT 2.0 leverages industry capabilities for managing the software/operating environment and managing third-party packages within FACT 2.0; further, modular capabilities developed under this effort have been packaged accordingly to industry standards and integrated into FACT 2.0 in this way to exemplify the modular and extensible approach utilized to build FACT 2.0.
SYNERGY WITH ENGINEERED RESILIENT SYSTEMS AND RT-120

During the execution of RT-117 for the development of FACT 2.0, GTRI was supporting the Engineered Resilient Systems (ERS) effort through SERC RT-120 in the development of the ERS TRADESPACE toolset. Each RT had separate goals, tasks, and ambitions. However, there were some core capability needs shared between the two. At a high level, both had the goal of creating a toolset to support the systems engineering workflow. One view of the systems engineering workflow developed at GTRI is offered in Figure 1. In mapping this workflow into toolsets, the various stages have been distilled into three main pillars: Define, Execute, and Analyze.

Figure 1 The Systems Engineering workflow.

RT-117 and RT-120 focus on different parts of this process, but both required some of the basic capabilities such as defining a system and representing relationships between systems, attributes, and performance parameters. Rather than creating two separate capabilities, GTRI strove to co-develop the core of FACT 2.0 and the first phase of the ERS TRADESPACE tool. Doing so resulted in a stronger backend and offered more time to the researchers for building custom front-end capabilities.

Figure 2 offers a graphical depiction of the shared and separate components between FACT 2.0 and ERS TRADESPACE. The core backend capability, named “Cortex”, is the primary shared component between the two. This offers the SysML-based ontology and datastore, base REST API, and execution engine leveraging OpenMDAO. Each toolset, however, offers separate front-end capabilities and graphical user interface. Based on the use cases each toolset wants to support, different front-end applications have been developed. However, though the front-end applications have been built for separate toolsets, since the same development process was utilized and for each application a goal of code reuse was in place, a front-end app can quickly be “re-skinned” and transitioned into the other toolset. Similar, any custom backend support for a front-end capability can be installed as a Django¹ app for an instance of FACT 2.0 or the ERS TRADESPACE tool and reused.

This synergy and collaboration between RT-117 and RT-120 has allowed for the delivery of a strong systems engineering framework base to the Department of Defense and encourages collaboration between the services in building tools to tackle the acquisition challenge.

<table>
<thead>
<tr>
<th>Frontend (Browser)</th>
<th>Backend (Server)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td>Python Django</td>
</tr>
</tbody>
</table>

**Unique components**

**Page-Views:**
- Point Solution
- Sensitivity
- Instance Manager

**Bower—**
- Manages/captures dependency between Javascript frontend components.
- Angular JS modules are wrapped as Bower components: Directives (view), Services (communications), Factories (data model).
- Share some logic for how we communicate with server, hold data at front end, etc.

**FACT 2.0 data pipeline—“TFOD”**

Pulls one table row at a time. Creates waterfall/task queue approach to function evaluation. Highly scalable to large data sets but performance not as fast or efficient.

**Cortex**

- **DJANGO apps**
- Functional components:
  - Cerebral – data structure
  - Motor – Execute/orchestrate analyses using OpenMDAO

**ERS TRADESPACE data pipeline—“Frontal Cortex”**

Pulls data table into memory at once and uses matrix-based functional operations. Results in simple and fast performance but scalability may be limited for massive data sets. Approach more suited for functions that require column of data instead of row (design instance).

Includes modular, scalable, composable analytical blocks.

---

**Figure 2** FACT 2.0 and ERS TRADESPACE, though separate capabilities, shares some common components and architecture, enabling sharing of capabilities across the projects.
FACT ARCHITECTURE REDESIGN

OVERVIEW

HOW DID WE GET HERE?

GTRI’s development of FACT 1.x was a piece meal collaboration over the course of three years, led by MARCORSYSCOM. Different customers required different capabilities, which were built into the application. This process resulted in a large feature set which existed in a single monolithic application. In order to setup and use FACT, many pieces needed to be in place. After the success of using FACT to support the 2013 ACV Feasibility Study, this larger effort was dedicated to the re-architecting of FACT for a 2.0 release.

ARCHITECTURE REDESIGN GOALS

Modular Architecture
The feature set offered by FACT 1.x needs to be broken down into modular, reusable pieces. Development of a modular architecture also provides examples of how to create new modules to fit into the framework.

Standardized API for Data Access
The only way to get data into FACT 1.x was to (1) use the capabilities offered by the GUI or (2) use the management shell/command line for entering data or creating and running custom input/update migration scripts. The former offered limited capabilities or often a cumbersome approach for large data entry while the latter required familiarity with shell scripting or a programming language such as Python.

Separation of Front-End and Back-End
Separating the front-end and backend is a first step in realizing the first two goals. FACT 1.x had tight integration and dependencies between the front-end and backend, making it difficult to reuse front-end components without a copy-paste and increased the learning curve for new developers.

Extensible Framework
More specifically, FACT 2.0 needs to be architected to lower the barrier of entry for FACT developers. FACT developers should be able to build new capabilities that leverage the existing base by only understanding the API to those base components and not inspecting or modifying that cost itself.

Improved Documentation
In order to enable the preceding goal, well-documented code with examples of use is required.

Automated Testing
As the developer base for FACT grows, an automated and standardize means to ensure regression defects are not introduced is required.
ARCHITECTURE REDESIGN GOALS STATUS

Modular Architecture

FACT has been broken down into separate Django and Angular\(^2\) apps. Apps can be installed or not installed based upon the needs of the customer. Apps can be replaced by other capabilities, which offer the same features in a different way. (e.g. MotorCortex, described later in this document, is the current model execution engine behind FACT which utilizes NASA’s OpenMDAO. This execution engine can be replaced by a different app, which offers a means to integrate and execute models or simulations.) One remaining step is to break the Django apps into separate Git repositories for configuration management. Currently, all Django apps live in one repo for CM. Front-end Angular components are already CM’ed in separate repos as individual components.

Standardized API for Data Access

FACT 2.0 offers a standard REST\(^3\) API (Representational State Transfer). Using Swagger\(^4\) FACT 2.0 offers interactive documentation, enabling discoverability. Swagger interactive documentation is driven by the code itself, therefore if there are updates to the API, the documentation is updated automatically. Developers can see example HTTP requests to understand how to communicate with the FACT 2.0 backend from a front-end application consuming the API.

Separation of Front-End and Back-End

The FACT backend is now completely isolated from the front-end capabilities. This will allow developers to focus on creating front-end capabilities using Angular that consume the API and leverage the backend capabilities without needing to be fluent in backend development. Likewise, developers can create new backend functionality offer new endpoints via the API without being fluent in front-end development. As an example, both FACT and the ERS Tradespace (RT-120) tool leverage the same backend with custom front-end applications. Some components can be shared between these two tools, as could other Angular components.

Extensible Framework

Achieving the previous goals has resulted in FACT 2.0 being the extensible framework sought. The use of industry standard frameworks and package managers allows users to leverage existing training materials and documentation for spooling up FACT developers. Google’s Angular framework is the backbone for front-end development. Bower offers a standard process for packing up front-end components, enabling reuse. Python’s pip process supports packing up backend components, which will allow custom FACT instances to be stood, where only the required backend capabilities and support libraries are installed. FACT usually requires customization when supporting a specific Program, and the realized extensibility of FACT 2.0 enables that customization.

Improved Documentation

Code is thoroughly commented. Searchable documentation is automatically generated from code with links directly to source code. With a running development environment, navigate your browser to http://localhost:27380/static_assets/sphinx/html/documentation.html. Reiterating, using Swagger, FACT 2.0 offers interactive documentation for the API enabling discoverability. API documentation is automatically updated as the API changes ensuring that the code, functionality, and documentation are always in sync.

\(^2\) https://angularjs.org/. AngularJS is JavaScript framework developed by Google for creating dynamic, interactive web applications.

\(^3\) http://en.wikipedia.org/wiki/REST

\(^4\) http://swagger.io/
Automated Testing

FACT 2.0 does not offer 100% automated test coverage, but there are processes and tools in place, which have resulted in far, more automated testing than FACT 1.x. With a running development environment, navigate your browser to http://localhost:27380/static_assets/coverage/index.html in order to see how well individual code modules are covered by automated tests. Automated tests will allow FACT developers to be sure that existing functionality remains in place while new capabilities are added.
DATA REDESIGN AND REST API

FACT 1.x’s monolithic structure meant that the front-end and backend capabilities were tightly coupled. This increased the complexity of adding new capabilities but more important steepened the learning curve for new developers. Therefore, a requirement of FACT 2.0 was to offer a REST application-programming interface (API) for the FACT back-end, to and refactor FACT to be more modular in order to increase its extensibility and flexibility.

API REQUIREMENTS

GTRI has derived some high-level requirements for this API, which will be detailed in the following. The requirements fall into three main categories: (1) requirements on the refactoring of the current FACT endpoints and back-end logic, (2) requirements on refactoring of the existing FACT database and data model, and finally (3) refactoring of the front-end.

First, the existing FACT endpoints needed to be re-factored to provide a comprehensive REST API for FACT 2.0. At present, the endpoints provide data as needed for specific views in FACT and were added as needed to support new views. This is not ideal, and rather the API should be designed to provide data in well-known formats that any front-end could consume as they see fit. Front-ends should then compose the data they need from one or more API calls. In order to accomplish this, the existing back-end will need to be refactored into more modular components (“apps” in Django parlance). The API should also provide read/write access to all documents and fields (accommodating read/write access to be based on user permissions) in the database. In general, the API must also provide access to data in a single query wherever possible for performance reasons.

The final two requirements on the existing endpoints and back-end logic have to do with managing the API itself. First, an existing framework for developing REST APIs should be used and it should work well with the data model abstraction that will be discussed in the next section. In addition, the API must be well documented, and whenever possible be self-documenting. GTRI is examining tools for generating API documentation and it is expected that a web-browsable API can be easily generated for use by other developers.

Next, the database and data model on top of it would need to be re-factored to support the REST API. First, this will require utilizing an Object Document Mapping (ODM) or Object Relational Mapping\(^5\) (ORM) to manage data access on the back-end rather than the query as needed Django views that currently exist. In FACT 1.x, Django offered the web framework but the built-in ORM designed to sit on top of a Relational Database Management System\(^6\) (RDBMS) was not utilized since Mongo was the primary data store. PyMongo\(^7\), an open source Python library, offered data access to the database through a querying API similar to that offered by the standard Mongo shell. These queries were one of the primary challenges to new developers coming into FACT development and also made the codebase hard to debug and risk-prone to upgrades for the supporting libraries and database itself. For FACT 2.0, the mongo-engine library offers an ODM to manage the object database schema and handle all data access, with an API similar to that offered by Django’s built-in ORM.

Though the ODM offers a familiar API for data access, the same permissions capability is not built into the mongo-engine library. GTRI will need to add user permissions to the Mongo database, which is presently managed in a MySQL database. As a long-term requirement, GTRI is also looking into the ability to version data in the database so a previous state can be reloaded for comparing tradespace results. The final requirement for database and data model is documentation of the format for use by future developers.

---


\(^7\) [http://api.mongodb.org/python/current/](http://api.mongodb.org/python/current/)
Finally, the front-end needs to be refactored to use the new API. In this phase, as many FACT 2.0 modules as possible/practical should be refactored to utilize the new REST API for data access. The component that can best serve as an example to future developers on how to create a component that consumes this API should be well documented as such. Additionally, user permissions controlling read/write/admin permissions at the collection- and object-levels, would be needed as mentioned previously.

**API Status**

GTRI has been working extensively on the re-factor of the FACT back-end to support this new REST API. This new back-end, named *Cortex*, provides a more modular approach to the problem of integrating collaborative web-based systems engineering tools, as given in Figure 3. The new API will be used not only by existing (re-factored) FACT components, but also by third parties wishing to interact with the new systems engineering framework. The work started with an analysis of the current data model. During this analysis, it was determined that much of the existing database content was in a hierarchical format across many collections that made the data difficult to access. In addition, the team prepared a set of initial derived requirements based on technical exchange meetings and guidance from the stakeholders. Some of these requirements include:

- the ability to version data within the database
- the ability to support modularization of the back-end components
- the ability to assign permissions to the data at various levels
- performance requirements

Figure 3 Applications of the modular architecture

The team developed several proposals to mitigate the difficulty in accessing data and satisfy the aforementioned requirements. The selected proposal utilizes a singletree implementation for the new data schema, backed by an
implementation of the schema using *MongoEngine*\(^8\). FACT 1.x utilizes MongoDB for data persistence; however there is no ORM enforcing consistency across data objects. The introduction of MongoEngine will allow for the enforcement of schema consistency and be available as a self-documented representation of the schema. It is important to mention that MongoEngine is technically an ODM as it is not handling relational data, but documents. The application that contains these MongoEngine models and assigns methods to them was named *CerebralCortex*, as that is the portion of the brain that plays a key role in language, memory and thought, three of the main functions that must be performed during the 'Define' stage, i.e., the first of the three pillars of the base systems engineering framework. In addition, this application is able to layer linked data (using JSON-LD) to allow for reasoning about the relationships by querying a persistent Resource Description Framework (RDF), implemented using BerkeleyDB\(^9\). CerebralCortex strives to store and manipulate Systems Modeling Language\(^10\) (SysML) data, and as such, the objects specified under it are, whenever possible named and structured after SysML constructs. In more specific terms, this application defines models for the Nodes (analogous to Packageable Elements), Blocks, Constraints, etc., automatically maintains the RDF graph, etc., and contains the basic operations required to do unit consistency, create instances of concepts, etc. It is a standalone application in the sense that it does not rely on other applications in order to operate.

The second pillar, 'Execute' requires the framework to integrate analyses and execute them. In order to accomplish this, GTRI is replacing FACT's current home-grown execution engine with NASA's OpenMDAO\(^11\). By implementing custom interfaces between SysML constructs and Multi-Disciplinary Analysis and Optimization concepts, Cortex allows for the relationships of constraints to other objects (e.g., other constraints or nodes in the model tree) to determine the dependencies between executable components and use this to automatically generate OpenMDAO assemblies which can be then tied in to the rest of the OpenMDAO machinery, including Design of Experiment (DOE) execution engines, optimizers, surrogate generators, etc. The application handling the execution of analysis was named *MotorCortex* as that is the portion of the brain that coordinates movement and action. MotorCortex relies on CerebralCortex to perform its functions.

In order to maintain a record of the evolution of the models and artifacts, the team developed a custom archiving application, termed *Archicortex*, which (as one of its many functions) translates between short term and long term memory. All documents in this framework are stored in the MongoDB database, which stores Java Script Object Notation (JSON)\(^12\) documents. JSON documents can be versioned using JSON-Patch, a W3C standard for defining operations to transform JSON objects. Archicortex is a Django application that integrates with the other Cortex applications and keeps an archive of versions of any class of object required. It ties into MongoEngine's *save()* method, to automatically generate a JSON-Patch between the current and last versions of the document. It also adds methods to these objects to be able to retrieve its state at a given version number or date. If indicated, it will also store the user that made the change. This is new functionality that was not previously present in FACT. The four cortices described above are represented in Figure 3, including their purpose and underlying technology. The figure depicts how they incrementally build on each other to provide a modular and extensible architecture. As described previously, CerebralCortex is the underlying foundation application on which the other cortices build. Additionally, the communications application, *Broca*, is included for reference and to indicate that additional applications with atomic functionality can be added to the suite. Broca will support communication between users, in a similar manner to how current FACT users can chat, but build on the concept to allow for user to have private conversations, and implement a ‘collaboration’ model that synchronizes the state of the other applications between users that are collaborating on a view.

\(^8\) http://mangone.org/
\(^10\) http://sysml.org/
\(^11\) http://openmdao.org/
\(^12\) http://www.json.org/
Permissions are handled using a flexible permissions architecture that can enforce different cascading rules for different projects and collections of objects. Two new applications have been developed to handle authentication and authorization. Current authentication is utilizing usernames and passwords but can be transitioned to utilize CAC or some other authentication system offered by the Government.

Initial specifications for the API were being developed in a language called RAML, the RESTful API Modeling Language\(^{13}\). After iterating on the content provided by these specifications, as well as the format of the API endpoints, these specifications were implemented using the Django REST Framework and its extensions to realize the endpoints. Though easiest to generate automatically, there is not always a one to one correspondence between entity types in the database (implemented as MongoEngine classes) and API endpoints, although whenever possible, the team has tried to do that to simplify the collaboration between the backend and front-end teams. The Django REST Framework automatically creates an explorable API interface, which has been augmented through the use of Swagger to facilitate users interested in understanding the different endpoints available.

GTRI has revamped its SysML authoring tool to facilitate the user experience. Primary goals were to integrate the tool with the new backend and reduce the effort required to define and modify models in it. Future work includes extending the interface to fully exploit the new back-end data model, to include specialization relationships, integration with requirements, etc.

\(^{13}\)http://raml.org/
FACT 2.0 has the ability to run model simulations using the open-source Multidisciplinary Design Analysis and Optimization (OpenMDAO\textsuperscript{14}) framework. OpenModelica\textsuperscript{15} is an open-source Modelica-based modeling and simulation environment used in academia and industry to model dynamic systems of the type FACT 2.0 is intended to be used for. Customers who see themselves using FACT 2.0 as an integrated modeling, simulation, and tradespace exploration environment have expressed the need to be able to run their pre-existing Modelica models within the framework and use the data generated from the simulations to feed the FACT 2.0 tradespace exploration environment. GTRI researchers working on FACT 2.0 sought to find a solution to this request by using the OpenModelica Python interface, OMPython, to bring Modelica simulation capabilities to FACT 2.0 via OpenMDAO components.

The OpenMDAO framework allows users to integrate disparate executable models so long as they possess a Python API. Its Component, Assembly, Driver, and Workflow constructs make OpenMDAO an effective and flexible tool for handling complex design problems. The dataflow within a system model is specified by linking Components together inside an Assembly. Optimizers, solvers, and design of experiments can then be brought into a Workflow using Drivers, providing numerous possibilities for solving problems and populating a design space.

Modelica is a non-proprietary, object-oriented, equation based language for modeling complex physical systems. The language itself is not Python based, so it cannot be integrated into the OpenMDAO framework directly. However, the OpenModelica environment includes an open-source Python interface for Modelica modeling, compilation, and simulation that can be used to build new models or run existing ones. FACT 2.0 developers sought to create OpenMDAO components, using the OMPython API, to configure, execute, and capture output from existing Modelica models within the FACT 2.0 framework. These components could then be integrated into Workflows consisting of drivers that could optimize and explore Modelica models, easing the transition from Modelica-based system models to an integrated MBSE framework such as FACT 2.0.

Developers used pre-existing examples taken from the open-source SystemDynamics OpenModelica library in order to test OMPython commands and OpenMDAO components. The OMPython API included commands that allowed for the creation of the OpenMDAO components listed in Table 1.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoadModel</td>
<td>Loads model or package from the OPENMODELICALIBRARY environment variable path</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
classname: Model, package, or library                                  | success: Boolean indication of success or failure                  |
| LoadFile      | Loads Modelica file (.mo) and merges it with loaded AST                      | 
classname: Modelica file name                                          | success: Boolean indication of success or failure                  |
| GetCompModVal | Retrieves a component modifier value                                         | 
classname: Component modifier: Component modifier                      | value: Component modifier value                                   |
| SetCompModVal | Sets a component modifier value                                              | 
classname: Component modifier: Component modifier                      | success: Boolean indication of success or failure                  |

\textsuperscript{14} http://openmdao.org/
\textsuperscript{15} https://openmodelica.org/
The LoadModel and LoadFile components allow users to load models, packages, or libraries into the OpenModelica simulation environment in FACT 2.0 as part of an OpenMDAO workflow. The values of simulation inputs can be retrieved and set using the GetCompModVal and SetCompModVal components, which could be used in drivers that optimize Modelica models or create design of experiments. Information about the model can be retrieved using the CheckModel component, and the simulation can be run using the SimulateModel component, which also returns the output. The final component, QuitOM, is necessary to end OpenModelica processes from an OpenMDAO workflow.

FACT 2.0 developers determined that the OMPython library is sufficient to enable Modelica model simulations within the OpenMDAO framework and modify them as needed for the purposes of design space exploration, Monte Carlo runs, and design of experiments. Plotting of the simulation results can also be done through the OMPython API, but the visualization capabilities of FACT 2.0 are superior in their appearance and usability to those provided by OMPython and OpenModelica, so plotting components were not created. Simulation results can also be returned in the form of a Python object, rather than as an output file, if the Modelica model is configured properly. OpenMDAO components could be developed to parse simulation output if the format is known. The OMPython library is not comprehensive enough to allow for full, top to bottom model development, although limited capabilities exist to modify existing model classes.
**FACT 2.0 POINT SOLUTION CAPABILITY**

Figure 4 The FACT 1.x Point Solution was designed with the user in mind to support the primary use case of understanding how well a point design meets or fails to meet a requirement set and furthermore, how it compares to specific baseline.

**WHAT IS IT?**

The FACT 2.0 Point Solution page has multiple purposes. Situated at the end of the FACT workflow, its main goal is to provide analysis toward a before and after comparison between a baseline and a generated design based off that baseline. Additionally, it allows for a requirements analysis at both a high level, looking at groupings of requirements, and a low level, looking at exact numbers for specific requirements. It also allows a user to create a modified baseline system and quickly determine how the modified system meets the same requirements the baseline system is judged upon.

A typical use case is when a baseline is created based off a tradespace study and the user wants to see if they can push the envelope of the broad design space explored in that study. Applying domain knowledge, the user can perform targeted exploration of discrete input parameters in a fraction of the time a full-factorial tradespace study would take. Another use case is when a new scenario dictates a different requirement set. The Point Solution page can be used to quickly create a virtual, modified baseline system to determine what changes need to be made in order to adapt to the new requirements (for instance, a greater payload requirement driving an increase in an existing systems cargo capacity).

**INSPIRATION: APPLY LESSONS LEARNED FROM FACT 1.x**

The FACT Point Solution page presented in Figure 4 is not the first version of the page. The first version of this page was the very first page in FACT 1.x. Many of the ideas and core purpose of the 2.0 version derive directly from the 1.x version. The 1.x version allowed for a requirements analysis of a vehicle (now called a system). This analysis was based on a three-point requirement scale. The vehicles could exceed a requirement objective, just
meet a minimum acceptable value (threshold), or fail to meet the threshold. Each requirement had a bar associated with it that displayed the various colors and had a point that illustrated where the vehicle lay on that requirement spectrum. The green part of the line corresponded to meeting the objective, yellow corresponded to meeting just the threshold, and red corresponded to failing to meet the threshold (see Figure 5).

**Figure 5 FACT 1.0 Point Solution requirements analysis**

A few things were learned from the creation and use of the FACT 1.x Point Solution page. First, do not try to do too many things on one page. Focus on the core responsibility of that page. FACT 1.x had a few extra features that were not part of the main requirements analysis goal. The FACT 2.0 version focuses exclusively on generating a point solution and aiding in requirements analysis. Second, as the complexity of the vehicles increased, it was often difficult to understand the results at a higher level as well as find the requirements that were of greater importance. The new version provides an overall requirements view in the form of a treemap to quickly observe larger trends and a convenient way to drill down to the more important requirements. Third, context is important. With FACT 1.x there was no direct requirements comparison between a baseline vehicle and the generated point solution. It is important to quickly see where there is an improvement in a design and where there is a regression. The FACT 2.0 version provides before and after comparisons with several interactive visualizations so the user can quickly, at both a broad and detailed level, determine improvements and regressions.

**CAPABILITY OVERVIEW**

**SYSTEM TREE**

The system tree is the leftmost tool on the page (illustrated in Figure 6). It allows a user to navigate the chosen baseline systems components (called instances). This tree provides a detailed, drill-down, view of the entire baseline. The dark green background illustrates the clicked-on ancestors of the tree. In this example the Tandem Helicopter Project bar is the root node of the tree. The Helicopter bar is a previously clicked-on child node of the Tandem Helicopter Project node. What is illustrated below the Helicopter bar are the children of Helicopter. Engine and Rotor are children of Helicopter that have children of their own (indicated by the lighter green background color). The other bars (k, cd, sfp, ...etc) are children of Helicopter and leaves of the tree with no children of their own. They are direct properties associated with Helicopter. This navigation tree is useful for picking input variables the user would like to modify for a point solution as well as viewing the baseline systems values. Clicking on any of these bars modifies what is shown in the Attribute Modification pane discussed in the next section.

**Figure 6 The tree responsible for navigating the baseline systems components**
**Attribute Modification**

After navigating through the system tree to pick an instance (with an associated list of attributes) or specific attribute to modify, the user will see the attribute(s) appear in the attribute modification pane (Figure 7) to the right of the system tree. This is where input variables can be changed from their baseline values and a point solution can be run. Sliderbars show the approved range of values the input variable can take. A text box is also provided for more precision. If units are provided, the unit is displayed below the text box and can be changed via a dropdown menu. After all desired changes have been made, the run point solution button at the bottom can be clicked to generate a point solution.

**Requirements Satisfaction**

The Requirements Satisfaction pane lies on the far right of the page. This pane provides all requirements analysis for both the baseline case (before any point solutions have been generated) and a point solution. Each time a new point solution is generated, the visualizations update to display information about the new point solution while illustrating some baseline results. The requirements analysis is encapsulated by multiple visualizations.

**Requirements Treemap**

The most notable visualization on the page is the treemap. Located in the upper right portion of the page within the Requirements Satisfaction pane, it is responsible for displaying a high level view of the requirement analysis. The interactive visualization has multiple parallel levels that can be walked down by clicking on their respective rectangle. Each rectangle represents either a requirement category or quantified requirement. Requirement categories allow for sorting of quantified requirements into logical groups. The upper levels correspond to requirement categories (Figure 8) while the bottom level represents all quantified requirements (Figure 9). The size of the rectangle can be indicative of the relative importance of the requirement or group of requirements within a requirement category. The color of each rectangle corresponds to how well the baseline or system met the requirement or requirement category. Green represents meeting the objective, yellow represents meeting just the threshold, and red represents a failure to meet the requirement. At first the visualization displays the results for the baseline, but after a point solution is generated it will update to display the point solution results.
Figure 8 High-level requirements overview displaying many requirement categories. The colors displayed by the requirement categories are a combination of the colors displayed by the children of that requirement category.

Figure 9 Drilled-down view of one requirement category with four quantified requirements

To navigate back up the tree, clicking the bar directly above the treemap traverses the tree backwards.

**Requirement Bar Chart**

The requirement bar chart displayed directly below the requirement treemap illustrates the number of quantified requirements that exceeded the objective, met the threshold, and failed to meet the threshold. Originally it displays just the results of the baseline system (Figure 10) but after a point solution is generated, it will display the point solution results in an overlapping, thinner bar within each category (Figure 11).
DIVERGING BAR CHART

The diverging bar chart is shown when a point solution is generated. Currently it illustrates the percent change for each quantified requirement from the baseline to the point solution (Figure 12). The chart can be sorted alphabetically, by largest gain, and by largest loss. Hovering over a bar will display the exact percent difference. A red bar indicates the point solution performed worse for the associated quantified requirement than the baseline. A green bar indicates the point solution performed better than the baseline for the associated quantified requirement.

Figure 12 Diverging bar chart showing percent change from baseline to point solution.

REQUIREMENT PROGRESS BAR

The progress bar is very similar to the requirement “stoplight” bars from the FACT 1.0 Point Solution page (shown in Figure 5). It appears after clicking on a quantified requirement rectangle in the treemap, replacing the two previously mentioned bar charts in the space below the treemap (Figure 13). This is where the user can see the exact numbers and how close they are to the key points on the requirement scale. Before a point solution is run, there is just a single number and pointer located on top, indicating where the baseline lies for that quantified requirement. After a point solution is run, the baseline pointer and value is relegated to the bottom of the bar and
made opaque similar to how the bar chart is represented in Figure 11. The point solution value appears prominently above the bar.

![Ferry Range Progress Bar](image)

**Figure 13:** The “ferry range” progress bar appears after clicking the “ferry range” rectangle in Figure 6.

---

**Supporting Libraries**

A few different open source software libraries were used in the creation of the visualizations and tools shown in the above sections. Angular.js wraps all of these components and provides the framework for the entire website. D3.js is a visualization library used to create each visualization, the system tree, and the sliderbars. Bootstrap is a library that helps style web pages to give them a nice look and feel. Each component discussed in the sections above is a component created by GTRI that is highly modular and extendable. Developers can easily pull out, easily replace, or customize the different panes or components to their liking with minimal effort.
DATA ANALYTICS PIPELINE: TRADESPACE FIELD OF DREAMS

The Tradespace Field of Dreams (TFOD) is a data analysis pipeline tool for taking a large selection of items of unknown quality and creating a chain of operations to analyze those items and reduce them to a desired set. It is an extension that builds upon FACT’s strengths as a web-based tradespace analysis tool and provides another avenue for injecting GTRI’s systems engineering expertise to guide customers in the system architecting lifestyle.

TFOD helps the data analyst in many ways. Each operation in a configured pipeline runs automatically once the appropriate upstream operations have completed, freeing them from having to manually start the next step in the process. Each pipeline is named and many can be defined which helps them to save common pipeline operations but still differentiate between them. Whole pipelines or individual operations can be run and rerun, which allows the analyst to quickly switch out the data that is being used but run the same analysis, easily change the analysis, or change a later portion of the analysis without having to repeat a potentially time consuming previous step that has already been performed. Each operation block allows the analyst to see the items before and after the operation, making the whole analysis transparent and enabling quick debugging of the data pipeline.

INSPIRATION: WHY WAS TFOD BUILT WITH FACT 2.0?

FACT 2.0 was the incarnation of lessons learned from the development and use of FACT 1.x. The use of FACT 1.x to support the MARCORSYS COM ACV Feasibility Study Team in 2013 was a significant milestone for the adoption of FACT in the acquisition process. In order to properly support the ACV Team, new FACT capabilities and modified workflow has to be developed. In short, these capabilities included:

(1) A means to define relationships between parts such as requires, is incompatible with, and required by. This capability was needed so that when creating variants, if there was existing knowledge about the compatibility or incompatibility of engines and transmissions, for example, then it was important to ensure variants created by the data pipeline met those understood relationships.

(2) Improve the tradespace execution capability to handling tens or hundreds of thousands of runs. The existing tradespace execution capability had not been stress tested and therefore needing improvement to handling the ACV problem.

(3) Enhance the tradespace execution capability for handling trading by part rather than by parameter. The initial FACT 1.x capability focused the tradespace exploration on a per parameter basis, meaning, trade on horsepower or torque as opposed to the whole engine component itself. Through this study the researchers learned that present day acquisition often leans toward selecting existing off-the-shelf components and assembling them to create the needed capability. Therefore, the tradespace is a discrete set of options rather than a continuous parameter space.

(4) Create a custom interactive visualization for exploring the tradespace execution results. Figure 14 provides an example view of this interactive capability.

(5) Development of a new metric for understanding risk associating with specific design decisions. This metric was proposed by a Government lead and iterated on by the research team to enhance the scatterplot seen in Figure 14 with a contour of coloring based on risk.

It was for these reasons, though (2)-(4) in particular, that drove the creation of TFOD. In order to execute the set of trade studies for the ACV team, the researchers created shell scripts, Python scripts, and some specific FACT backend logic not exposed to the existing front-end in order to meet the needs of the ACV Team. In short, they created various pieces to a data pipeline. Manual inspection was required to monitor the execution of tradespace runs and ensure they were succeeding. Additionally, though minimal work was involved, data was often manually moved from one stage of the pipeline to the next, specifically from the creation of the tradespace to the creation of the static artifacts required by the ACV Team for analysis and decision making.
One particular part of the data pipeline process is displayed in Figure 15. The core of capability FACT 1.x required the definition of a physical decomposition of the solution architecture, usually referred to as the WBS or work breakdown structure. For the ACV study, each element in the WBS was a system or subsystem, and for each leaf subsystem 1 to \( n \) parts were defined which were options to satisfy that subsystem need; for some subsystems, void parts were allowed. In order to create a complete variant, a part which satisfies each subsystem is required. Therefore, the process outlined above in Figure 15 describes the data process, which results in the creation of a single variant. In order to create 10,000 variants for a specific study, this process would be run 10,000 times.

![Figure 14 Example of the interactive analysis tool used to support decision-making. The left provides a cost vs. mass margin scatterplot while the right shows histograms of the selected parts for each subsystem. In this example image, the last part in the 2.01.01.01.02 WBS element has been clicked and selected, filter out (displaying them in red) all of the options that did not utilize that specific part.](image)

To begin the process seen in Figure 15, the WBS was randomly shuffled in order to select which subsystem to select first. This was a precautionary measure to increase randomness (or at least maintain randomness) in the likelihood of a selected part given the dependencies between parts. This random order was used to select the next WBS element (unless already filled based on a preceding requires or other relationship). With a current WBS element, the valid options for the specific trades studying were identified. One goal of the trades study was to minimize mass. Therefore, some percentage likelihood was set at the start of each study run as to how strongly to prefer the minimum mass. For example, if the percentage was 50% then for each WBS element, 50% of the time the lightest valid part was selected. Depending on the set percentage and random number for the iteration, the parts were either randomly sorted or sorted ascending by mass. Each option was reviewed in turn to determine if it was valid and could be selected based on the requires and incompatible relationships. Once a valid part was selected the process continued. If no part was validated, the process terminated and began from the beginning. If all subsystems were filled, then a feasible candidate variant has been built and was saved for analysis.
After the completion of a tradespace, static artifacts (including the interactive decision support tool seen in Figure 14) were created from the generated data. This process continued dozens if not hundreds of times over a nine-month period, resulting in the creation of over one million candidate variants. Parts of it were automated, but the management of the pipeline was more manual than not.

Figure 15 Data process used by ACV Feasibility Study in 2013 to create variants within a tradespace.

It is this past experience that drove the need for the creation of a data-pipelining tool for FACT 2.0. Had the researchers in 2013 had TFOD, custom blocks could have been created for the process outlined in Figure 15, the execution of valid vehicle sets using the model graph defined in FACT, and the creation of the static artifacts for analysis. Further, TFOD automatically handles the parallel execution of tasks utilizing available computing hardware. In 2013, researchers manually sized and began jobs to utilize the available hardware; with TFOD the manual management of computing resources would have been replaced by the automated queueing system. Lastly, the visual representation of the workflow, provided by the TFOD interface, would have been a powerful visualization itself to show decision makers the process by which the data being presented to them was created.

TFOD DATA MODEL

The pipeline is the whole collection of data analysis operations. Each individual operation is a block and each block performs only one type of operation. The blocks operate on individual items that flow through the pipeline from one block to the next as appropriate. The flow of items is defined by connections between blocks. In TFOD, each block operation is defined to be one of two basic types: streaming or bulk. A streaming block is one that operates on one item at a time and passes that item downstream as soon as it is processed. In contrast, a bulk block is one that operates on all items simultaneously and passes all the items downstream once all are processed. Certain operations are more suited for one type over the other. For example, a filter operation is streaming because
whether one item meets a certain criteria has no effect or whether a previous or later item will while a scatterplot of all items will require all the items before it can be rendered, making it bulk.

The underlying task queueing system in TFOD is greatly affected by the choice of streaming or bulk. For streaming tasks, one task is added to the streaming queue per item. For bulk tasks, one task is added to the bulk queue per block. For blocks with many items, many tasks are created and the queueing system must handle each of them, which takes time, even for very simple tasks such as filtering. Due to this large amount of overhead, performing the same operation with the same number of items in bulk is generally much faster than in streaming due to only having one task. For this reason, many streaming operations have also been implemented in bulk. However, streaming blocks are still useful for any operations that can act on a single item that are known to be time consuming. If they are processed in streaming, the preliminary results could be viewed while the remaining items are still processing which is much more informative than having to wait until they are all done as in a bulk process.

As will be discussed further on, near the end of this effort the researchers determined that the best approach was to replace the streaming and bulk blocks with a single batch block, where the batch size could vary from 1 to all where 1 is the same as streaming and all is the same as bulk. The advantage of the sliding scale would be to allow optimization of the passing of data for non-bulk blocks to minimize overhead percentage. For each operation, some overhead is involved in the management of that task. For a very small filter operation, the overhead far exceeds the single conditional execution. Therefore, batching filters into groups of a few thousand or even more could prove to greatly reduce execution time. Furthermore, a feedback loop could be applied to automatically optimize the batch size during runtime.

**BLOCK OPERATION TYPES**

One of the goals of TFOD was to implement many basic types of data analysis operations, but simultaneously make it very easy for a data analyst to add their own customized block. For this reason, there is a hierarchy of block types that defines shared attributes and functions at the highest level possible so that new blocks can be added with a minimal amount of work once a proper parent is identified. That hierarchy is shown in Table 2 below. Block types shown in blue are those that can be added directly to a pipeline.
Table 2. Hierarchy of block types implemented within TFOD.

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming</td>
<td>Parent class for all operations that process one item at a time</td>
</tr>
<tr>
<td>FilterStreaming</td>
<td>Determine whether items meet a given criteria</td>
</tr>
<tr>
<td>TransformStreaming</td>
<td>Parent class for modifying items by adding a new attribute</td>
</tr>
<tr>
<td>NewFieldStreaming</td>
<td>Add a new attribute that is a product, sum, difference, fraction, or square of existing attributes</td>
</tr>
<tr>
<td>SplitStreaming</td>
<td>Parent class for sending items to multiple downstream blocks</td>
</tr>
<tr>
<td>CloneStreaming</td>
<td>Send all items to all downstream blocks</td>
</tr>
<tr>
<td>BinStreaming</td>
<td>Send items to downstream blocks based on mutually exclusive bins</td>
</tr>
<tr>
<td>Bulk</td>
<td>Parent class for all operations that process all items together</td>
</tr>
<tr>
<td>Artifact</td>
<td>Parent class for generating a file from the items</td>
</tr>
<tr>
<td>Scatterplot</td>
<td>Generate a two dimensional static scatterplot</td>
</tr>
<tr>
<td>CSVArtifact</td>
<td>Generate a CSV file with all of the items and their attributes</td>
</tr>
<tr>
<td>Dataset</td>
<td>Parent class for getting data into a pipeline</td>
</tr>
<tr>
<td>CSVData</td>
<td>Bring data in by reading from a CSV file where each row represents an item</td>
</tr>
<tr>
<td>JSONData</td>
<td>Bring data in by reading from a JSON file where each dictionary represents an item</td>
</tr>
<tr>
<td>Merge</td>
<td>Parent class for combining items from multiple downstream blocks based on an exact match of all attributes</td>
</tr>
<tr>
<td>Union</td>
<td>Combine items by taking all unique items that were on any downstream block</td>
</tr>
<tr>
<td>Intersection</td>
<td>Combine items by taking only items that were on all downstream blocks</td>
</tr>
<tr>
<td>FilterBulk</td>
<td>Determine whether items meet a given criteria</td>
</tr>
<tr>
<td>TransformBulk</td>
<td>Parent class for modifying items by adding a new attribute</td>
</tr>
<tr>
<td>NewFieldBulk</td>
<td>Add a new attribute that is a product, sum, difference, fraction, or square of existing attributes</td>
</tr>
<tr>
<td>SplitBulk</td>
<td>Parent class for sending items to multiple downstream blocks</td>
</tr>
<tr>
<td>CloneBulk</td>
<td>Send all items to all downstream blocks</td>
</tr>
<tr>
<td>BinBulk</td>
<td>Send items to downstream blocks based on mutually exclusive bins</td>
</tr>
</tbody>
</table>

There are only a few simple rules regarding the structure of blocks on a pipeline.
1. *SplitStreaming* and *SplitBulk* blocks are the only types that may have more than one downstream block
2. *Merge* blocks are the only type that may have more than one upstream block
3. *CSVData* and *JSONData* blocks are the only types that may have no upstream blocks

### The TFOD Interface

The TFOD Interface provides the data analyst with a simple way to create and interact with pipelines, blocks, connections, and items. This is where the pipeline is built, data added, and the results analyzed.

### A User Story

Rich is trying to produce the next great sports car for Ford. Using a statistical tool set, he has produced 10,000 variants of potential candidates. He wants to browse through those variants to help him make decisions about
what components the next car should use, how parts should be sized, what tuning parameters should be used, and more. The end goal of his analysis is to end with a set of no more than 5 candidate vehicles that he can present to the rest of the engineering team, along with documentation to defend his reasoning behind why those particular vehicles are the best.

**BUILDING A PIPELINE**

Rich can use TFOD to accomplish this goal. From the TFOD home page, he can create a new blank pipeline by clicking the ‘+’ button from the pipeline selection dropdown in the top left corner, as shown below, and giving the pipeline a name in the modal window that follows.

Next, he needs to begin adding blocks by clicking and dragging from the Add Block menu shown below. This menu is organized similarly to the block hierarchy shown above.

Generally, a good first step is to add a Dataset block. Assuming the data is in a CSV format, Rich can click and drag the CSVData block onto the pipeline and drop it on the ‘Add to pipeline unconnected’ hover target. For all blocks, selecting the block exposes all of the options for interacting with that block. The options for the CSVData block are shown below.
Each selected block has two buttons in the top right corner of the block. The first one from the left is the play button for running that individual block. When it is clicked, that block processes all of its items and then runs the pipeline from that point forward so that all downstream blocks are run while upstream blocks are not. The next button is the ‘x’ button for deleting that individual block along with any connections to or from it.

The ‘x’ button just outside the block is for adding connections from that block. Based on the connection rules given above, this button may not always be available. When it is available, clicking and dragging to a click target on another block creates a connection between those two blocks.

Selecting a block opens the middle panel with a set of tabs for interacting with a block. The ‘Configuration’ tab is first and will differ for every block type depending on what the analyst needs to provide for that specific block type. Except for a child of the Dataset block such as this one, the next tab will be an ‘Input Items’ tab that shows the items coming into that block from the downstream block(s). The ‘Output Items’ tab is next and shows the items leaving that block and going downstream. The last tab is the ‘Previous Runs’ tab, which is still under development.

More blocks can be added to the pipeline by clicking and dragging from the Add Block menu again. Though there was only one drop target for the first block, more will become available, as shown below.
To add a block to the pipeline without any connections, choose the ‘Add to pipeline unconnected’ target. To add a block downstream from a given block, choose the ‘Add downstream’ target for a specific block and the new block is added to the pipeline and a connection drawn from the given block to the new block. To add a block between two blocks, choose the ‘+’ target on an existing connection and that connection is deleted and replaced with two new connections with the new block in the middle of the original two blocks.

As the pipeline is built and blocks are configured, the block states will change accordingly and error or warning messages will appear in the blocks as needed to guide this process, as shown below.

There are six states that the blocks could assume. These are given in Table 3 below along with their meaning.

### Table 3. TFOD offers six different states to define the status of a specific block in a pipeline.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stopped</td>
<td>The block is not running (default state)</td>
</tr>
<tr>
<td>running</td>
<td>The block is currently processing items</td>
</tr>
<tr>
<td>completed</td>
<td>The block has completed processing items</td>
</tr>
<tr>
<td>stale</td>
<td>The block is outdated due to a change upstream</td>
</tr>
<tr>
<td>broken</td>
<td>The block has a problem with its connections</td>
</tr>
<tr>
<td>failed</td>
<td>The block was not able to process the items</td>
</tr>
</tbody>
</table>

Once a pipeline is configured, it can be run by clicking the ‘Run Pipeline’ button, which will start the processing of items on every Dataset block child on the pipeline. When any block is running, neither the pipeline nor any individual block can start running. During this time, making any changes to any blocks is not advised because they might be overlooked in the current pipeline run. The figure below shows a running pipeline.
After the pipeline is done running, the items on each of the blocks can be inspected closer by selecting a block and clicking on either ‘Input Items’ or ‘Output Items’ tabs. Each item attribute column can be sorted ascending or descending by clicking the arrow next to its name, as shown for `breaking_distance(ft)` below. These sorts are performed over the entire dataset, not just the data showing on the current paged results.

For blocks with multiple upstream or downstream blocks, the items are shown separately, as shown below for the `CloneBulk` block.
The strong TFOD framework and interface give Rich the ability to easily analyze his car data in a flexible, repeatable, and transparent manner.

**TFOD Next Steps**

Another goal of TFOD was to maintain a history of runs and make it viewable for the data analyst from the interface. Though this is not yet available, the framework was designed with this goal in mind. When a block is run, new items are added to it and assigned a run number. On rerun, old items are marked as inactive and new items are added with the current run number. Only the current items are viewable from the interface. Creating a viewable history would mean storing run numbers for block configurations as well and exposing a way to select a run number from the interface for which to show the pipeline configuration, blocks and items included. This viewable history would greatly expand the power of TFOD by providing the capability to return to previous states of an analysis at any time.

Furthermore, the researchers are transitioning from streaming and bulk blocks to a single batch block. As described above, this transition will offer the maximum flexibility in how the pipeline is executed and providing a mechanism for optimizing runtime. Additionally, the researchers want to explore means to have artifact or display blocks that are streaming. An example is a scatterplot matrix, which populates with points as they exit the pipeline, allowing the analyst to view the results as they complete rather than waiting for the entire process to execute. This could allow an analyst to identify some error in the pipeline configuration before running the pipeline for hours on a large dataset.
FACT SOFTWARE STACK, DEVELOPMENT ENVIRONMENT, AND HOSTING

One of the primary goals for FACT 2.0 was to derive a development environment which eased the onboarding process and transitioned well from the RT-117 FACT development team to future Government and contractor developers. Delivery of the FACT can now be accomplished by sharing the source code with the Government and having them following the exact same process and procedures followed by the FACT 2.0 developers.

TECHNOLOGY STACK SELECTION

The objective of this task is to identify the allowable front and back-end technologies that will be approved for Information Assurance purposes. This includes a survey of prospective back-end server hosting sites across the Military Services and DoD, and Service specific limitations on front-end web browsers and related applications.

Requirements for this task include conducting a thorough review of existing technology alternatives to provide the required functionalities for FACT 2.0. This will lead to defining the (known) desired server stack. GTRI then needs to provide this stack documentation for approval, along with supporting documentation. Unless specifically required, this stack will be based on no/low-cost to Government software wherever possible (e.g. open source, free software).

SELECTED TECHNOLOGIES

GTRI conducted a series of evaluations for various technologies to include in the stack. Referred to as ‘The Policy of the Shinnies’ (TPS) reports, GTRI has conducted 14 such studies thus far, ranging between selection of operating systems, databases, to architectures and programming patterns. Each report included between two and six potential solutions. The reports and findings are available to the Government.

The majority of the current stack is depicted in Figure 16. It includes MongoDB as the database, MongoEngine as the Object Document Mapper, RDFlib and BerkeleyDB to manage, query and store the RDF data for semantic querying, OpenMDAO as the execution engine, Django as the back-end web-framework, DjangoRestFramework to provide a RESTful interface to the backend data, and the ability to support web-applications or an IPython\(^{16}\) notebook interface for rapid prototyping and one-of data manipulation. Other important elements of the stack include Apache as the webserver, and, as previously mentioned, Redis to manage a queueing system, and supervisor to managing daemons.

Some of the elements described in this section may be removed or modified for the final deployed environment, e.g., IPython Notebook interface, but the existing stack is considered mature enough at this stage that only small modifications are expected.

\(^{16}\) http://ipython.org/
DEVELOPMENT ENVIRONMENT CONFIGURATION MANAGEMENT

All FACT software development was conducted using VirtualBox\textsuperscript{17} virtual machines (VMs) running Linux (CentOS 6). There are several reasons for this:

1. Most of the developers are running Windows, and FACT is designed to run in a Linux server. This allows the developer to have his own Linux environment on his own machine.

2. This allows for a consistent environment and configuration across all developers and testers on the project. It also mirrors the production environment as closely as possible.

3. Each developer has their own machine and will not influence other developers as they would in a shared resource. They can experiment and make changes and the environment can be restored to a pristine state by re-creating the VM.

4. Testing can always be done with a clean VM, so the whole environment can be understood and nothing is installed by hand.

For most of FACT 1.x development, a VM was created by hand and people would download the common VM image to work from. As changes were needed, they were done by hand, and a new image was loaded to a

\textsuperscript{17} https://www.virtualbox.org/
common server periodically. Eventually, after a number of iterations, the image had artifacts of previous development, and recreation of the environment from scratch was difficult.

To remedy these issues a modified approach using Vagrant\textsuperscript{18} and Packer\textsuperscript{19} has been adopted. Vagrant is a command line wrapper around VM technologies, specifically VirtualBox in the case of FACT 2.0. Vagrant makes it easy and convenient to manage VirtualBox for a development environment. Packer is a tool used to build a VM base image (called a “basebox”) for Vagrant.

Here are the steps used to create a running development environment:

1. Packer starts with a standard CentOS 6.6 ISO image and installs a clean version of CentOS into a new virtual machine suitable for Vagrant and VirtualBox.
2. Packer runs a set of provision scripts, that install all the software needed and sets up the machine environment to run the FACT application.
3. Packer then saves this VM image to a basebox file that is used by vagrant.
4. The basebox image is put on a location that is accessible by developers.
5. The FACT code base in git contains a file called “Vagrantfile”. When in the directory containing this file, running “vagrant up” will download and install the basebox image (if not already installed), and then start a new, clean VM using this base box.
6. Some further provisioning is done to add resources needed by this specific version of FACT.
7. The code base is then “mounted” into the vagrant VM so that the code the developer is working on is used by the VM.
8. The VM forwards ports (such as port 80 for the web server) to ports on the developer’s host machine (such as 27380). Then the VM web server can be accessed by a browser with a URL such as http://localhost:27380/.

A developer or tester only needs to check out the code and type “vagrant up” and point their browser to this localhost URL to start working. The vagrant box can be started again from scratch by destroying the vagrant VM and starting it again:

text

```
vagrant destroy
vagrant up
```

If changes are needed to the base box, the packer provision scripts are modified, and the process is started again from step 1.

This method is completely automated and all the code is stored in the git version control system. In this way the provision scripts document the entire environment. With slight modification, the scripts can be run against a non-Vagrant VM to set up a more permanent environment. Further, since the provisioning scripts are text-based files, git’s powerful capabilities are best utilized to track and understand how the provisioned environment has changed overtime. Furthermore, if a developer requires a specific new package, whether it be front-end or backend, a bower.json or requirements.txt file, respectively, can be updated allowing a developer to \textit{vagrant provision} when conducting a peer review of a feature or bug branch.

\textsuperscript{18}https://www.vagrantup.com/
\textsuperscript{19}https://www.packer.io/
The researchers are considering another possible shift in the provision shell scripts, shifting them into different technologies, such as ansible\textsuperscript{20}, to make them more flexible and easily run against different machines, such as VMWare, cloud services such as Amazon EC2, and physical hardware.

An additional option, which was released during FACT 2.0 and being considered, is using containerized technologies such as docker\textsuperscript{21} to speed up the time to modify and launch a development VM, and to better separate technologies (such as database, web server, and API server) to simulate a compartmentalized production environment. This technology can also be used to consistently manage and distribute simulations such as trade space runs.

### SOFTWARE STACKS

The FACT application can be thought of as having two parts, a front-end software stack running in the browser, and a back-end software stack running on the server.

The front-end stack is primarily based on Google’s AngularJS web framework, and a number of other javascript libraries including d3, restangular, bootstrap, ngSocket, and some libraries written for FACT.

These front-end files are served by apache (httpd) on the server. The bower\textsuperscript{22} system is used to download and manage the versions of all the libraries, and the gulp\textsuperscript{23} system automates the handling of these resources, as well as preparing and optimizing all these components for delivery by apache.

The server running backend stack is primarily based on the Python web framework Django and communicates with the front-end code using a REST protocolRep. The backend utilizes several databases (mongodb, mysql, redis), and uses OpenMDAO, IPython notebook, and a number of other Python libraries. Django runs in apache using the mod_wsgi apache module.

The FACT application is database driven and all the data resides in the database. The backend interface (using Django) provides data to the front-end code (using Angular and running in the browser). The front-end code then formats this data as HTML so it can be viewed in the browser.

Although all the different parts are integrated, they are written as modules so they can be re-used and can sit relatively independently, allowing easier development and rapid creation of new modules. Although complete independence is very difficult in an application such as FACT, this goal is kept in mind when developing the different components to try to enable rapid and flexible future development.

\textsuperscript{20} http://docs.ansible.com/
\textsuperscript{21} https://www.docker.com/
\textsuperscript{22} http://bower.io/
\textsuperscript{23} http://gulpjs.com/
Now with the release of FACT 2.0, where do we go from here?

The first goal of the researchers is to apply the new framework to a handful of real world problems, working alongside other engineers to understand how the framework performs on a significant problem with a collaborative team. It is by the use of FACT 2.0 that the developers will determine the capabilities most needing to be developed going forward.

A second but equally important goal is to train Government engineers and software developers on how to continue development of FACT 2.0 with new modules and capabilities. Much thought and consideration went into the architectural decisions for FACT 2.0, which ensured that the desired flexible and modular architecture was realized. The researchers which developed this base now want to see other, new developers take the framework and build tailored solutions to assist various systems engineering acquisition challenges.

Additionally, not all of the capabilities present in FACT 1.x have been ported over into the new 2.0 Framework. Through the use of FACT 1.x, some of the capabilities were deemed unnecessary, but in order to properly populate the built-in feature set of FACT 2.0, the researchers are looking to gather a quorum of stakeholders to review the previous set of features from FACT 1.x, become familiar with the existing FACT 2.0 capabilities, and develop a roadmap for the re-introduction of capabilities into the new framework. It is a goal of the researchers that the prioritization of features be informed by use of teams utilizing FACT 2.0 for present problems.

In parallel to FACT 2.0 development, the research team has been able to support other systems engineering problems, most specifically challenges that fit in the category of portfolio management. FACT 2.0 is still a framework best suited for the design and evaluation of a system, backed with a set of models and simulations, which predict the behavior of said system in some environment. GTRI is interested in combining the FACT 2.0 capability and other portfolio management techniques (like those discussed in the SERC RT-112 effort) into a single comprehensive systems engineering capability. FACT 2.0’s modularity allows for portfolio problems to be represented and models integrated; the researchers are interested in building-in specific modules and features for the evaluation of a portfolio of systems. This would further improve the capability of evaluating a single system as its performance within a system of systems; representing this performance perspective in the tools and visualizations offers better and more complete information to decision makers.

There are many paths to the continued development and application of FACT. GTRI anticipates even the paths not discussed above, to see how FACT matures through its use and application to DoD acquisition programs. The researchers look forward to seeing how others adopt and adapt FACT to meet these new challenges.
The following students supported the research and development conducted under this Research Topic.

<table>
<thead>
<tr>
<th>Name</th>
<th>Degree, Major</th>
<th>Expected Graduation Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Butler, Barry</td>
<td>BS, Aerospace Engineering</td>
<td>May 2016</td>
</tr>
<tr>
<td>Carroll, Jordan</td>
<td>BS, Computer Science</td>
<td>May 2015 (graduated)</td>
</tr>
<tr>
<td>Eppinette, Joshua</td>
<td>BS, Computer Science</td>
<td>May 2016</td>
</tr>
<tr>
<td>Ham, Cu Su</td>
<td>BS, Aerospace Engineering</td>
<td>May 2017</td>
</tr>
<tr>
<td>Mark, Thomas</td>
<td>BS, Computer Science</td>
<td>December 2014 (graduated)</td>
</tr>
<tr>
<td>Scoppino, Giuseppe</td>
<td>BS, Computer Science</td>
<td>May 2016</td>
</tr>
</tbody>
</table>
**Publications and Presentations**

**Conference Papers (refereed)**


**SERC Invited Talks**

REFERENCES


