

Establishing model-to-model interoperability in an engineering workflow

by

Gerrick Bivins

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Mechanical Engineering

Program of Study Committee:
Kenneth M. Bryden, Major Professor
Arne Hallam
Richard A. Lesar
Mark Mba-Wright
Abhijit Chandra

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2019

Copyright © Gerrick Bivins, 2019. All rights reserved.

TABLE OF CONTENTS

	Page
LIST OF FIGURES	v
LIST OF TABLES	viii
NOMENCLATURE	ix
ACKNOWLEDGMENTS	x
ABSTRACT.....	xi
CHAPTER 1. INTRODUCTION	1
1.1 Overview.....	1
1.2 Organization	9
CHAPTER 2. BACKGROUND	12
2.1 Monolithic Modeling and Modeling Frameworks.....	12
2.2 Engineering Modeling within a Cloud-Computing Environment	14
2.3 Microservices.....	16
2.4 Integration in Cloud-Computing.....	18
2.5 Application Programming Interfaces (API).....	21
2.6 System Models as Microservices and the Problem of System Integration.....	24
2.7 Semantic Interoperability.....	28
2.8 Conclusion	30
CHAPTER 3. A SPECIFICATION FOR AUTONOMOUS SYSTEM INTEGRATION.....	32
3.1 Overview.....	32
3.2 The Challenge of Communication Between Dynamically Coupled Systems	32
3.3 Application Coupling Interface: A System Integration Specification	36
3.4 JSON Schema	39
3.5 Summary	45
CHAPTER 4. ENTERPRISE INTEGRATION PATTERNS IN SYSTEM MODELING.....	46
4.1 Overview.....	46
4.2 The Current Landscape.....	46
4.3 Quick Wins	48
4.4 Comprehending Realistic System Models.....	49
4.4.1 Directed graphs and model execution, workflow representation	50
4.4.2 Orchestrating execution of directed graph workflow	52
4.5 FMS 2.0: Introduction of Enterprise Integration Patterns	54
4.5.1 Refactoring	56
4.5.2 Approach	57
4.5.3 Workflow context.....	58
4.5.4 Relevant message system integration patterns for routing	59
4.5.5 Implementation.....	63

4.5.6 Framework.....	64
4.5.7 Asynchronous subsystem execution of modeled workflows	65
4.5.8 Routing inbound messages.....	69
4.5.9 Routing internal messages.....	70
4.5.10 Routing outbound messages.....	70
CHAPTER 5. LEVERAGING THE ACI IN SYSTEM MODELING.....	74
5.1 Overview.....	74
5.2 General Setup.....	75
5.3 Discovering Subsystem Models	77
5.4 Authoring ACIs	77
5.5 Publishing and Registration.....	78
5.6 Constraint Microservice.....	79
5.7 Protocol Microservice.....	80
5.8 Registration Microservice.....	81
5.9 Designing realistic systems.....	83
5.10 Workflow Microservice.....	84
5.11 Applications of the ACI	86
CHAPTER 6. AN EXTENSIBLE FRAMEWORK FOR SYSTEM MODELING/DESIGN.....	91
6.1 Overview.....	91
6.2 Contributing a Web Enabled Subsystem Model.....	92
6.3 Creating a Web Enabled Subsystem Model	92
6.4 Creating an ACI for a Web Enabled Model	95
6.5 Registering an ACI for a Web Enabled Model.....	104
CHAPTER 7. AUTONOMOUS SYSTEM INTEGRATION USING ACI'S	105
7.1 Overview.....	105
7.2 Architectural Considerations for Subsystem Models as Microservices in a Cloud Environment	105
7.3 High Level Architecture of a Decentralized Design Environment.....	111
7.4 Web Client Application	113
7.5 Desktop Client Application	117
CHAPTER 8. CONCLUSIONS AND FUTURE WORK	121
8.1 Overview.....	121
8.2 Conclusion	121
8.3 Future Work.....	122
8.3.1 Self organization.....	122
8.3.2 Breadth of systems model domains.....	123
8.3.3 Additional protocols for endpoints and routing	123
8.3.4 Constraint verification.....	124
8.3.5 Human computer interaction.....	124
8.3.6 Registration process	125
8.3.7 Transient workflow execution information.....	125
8.3.8 Parameters	125

REFERENCES	127
APPENDIX A. CONSTRAINTS MICROSERVICE API	135
Endpoints	135
Models	140
APPENDIX B. PROTOCOLS MICROSERVICE API DOCUMENTATION	144
Endpoints	144
Models	149
APPENDIX C. WORKFLOW SERVICE API DOCUMENTATION	152
Endpoints	152
Models	169
Example usage	175
APPENDIX D. REGISTRATION MICROSERVICE API DOCUMENTATION	180
Endpoints	180
Models	185
APPENDIX E. ATTRIBUTIONS	188

LIST OF FIGURES

	Page
Figure 1.1: Engineering design and modeling environment leveraging cloud technologies (Bryden, 2014).....	4
Figure 2.1:General sequence diagram of remote procedure a call for an application.	23
Figure 2.2:REST API defining the protocol for a service that calculates the sum of input numbers.	26
Figure 3.1: Two perspectives of an integration specification. Left, a model developer declares operational concerns of a model, such as execution protocol and parameter constraints of the model. Right, validation and recommendations are yielded to the system modeler/designer during the process of coupling system models.....	35
Figure 3.2: A sample ACI for an “Addition” model that uses Amazon Web Services Simple Queue Service for its protocol	38
Figure 3.3: Protocol and Protocol Details declaration for a REST protocol.	41
Figure 3.4: Protocol and Protocol Details declaration for a model using Amazon Simple Queue Service as its API	42
Figure 4.1: Example workflow message as described by Suram et al. (2018).	47
Figure 4.2: A system composed of ACIs grouped by execution levels as determined by traversing a directed graph of the connected parameters.	51
Figure 4.3: A “simple” complex system model design representing the aggregation of time varying data from sensors.....	52
Figure 4.4: The “Router” pattern as defined by Hoppe and Woolf (Enterprise Integration Patterns, 2019).....	60
Figure 4.5: The “Splitter” pattern as defined by Hoppe and Woolf (Enterprise Integration Patterns, 2019).....	61
Figure 4.6: The “Aggregator” pattern as defined by Hoppe and Woolf (Enterprise Integration Patterns, 2019)	62
Figure 4.7: Flow diagram for asynchronous subsystem execution using a messaging system architecture and enterprise integration patterns.....	66

Figure 4.8: Pipes and Filters architectural style for a messaging system (Enterprise Integration Patterns, 2019).	67
Figure 4.9: Enterprise Integration Patterns applied to the engineering modeling/design environment following a messaging system architecture.....	68
Figure 5.1: General microservices (μ) architecture used for supporting tools.	75
Figure 5.2: An example of a call to the API (/constraints/validate) for validation of a constraint using cURL.....	79
Figure 5.3: An example of a call to the API(/constraints/verifyExpression) using cURL.	79
Figure 5.4: Example request to query registered protocols of the environment using cURL.	80
Figure 5.5: Example request to register a new ACI for a modeled subsystem in the design environment.....	81
Figure 5.6: Sequence diagram for the registration of an ACI.....	82
Figure 5.7: Autonomous Routing Engine leverages the protocols declared in ACIs to orchestrate execution of the prescribed system workflow using EIP in the FMS.....	88
Figure 5.8: Autonomous Integration Processing Engine leverages the parameters declared in the ACIs to derive a directed graph of ordered execution of subsystems from a validated linking of ACIs from the Autonomous System Design Service.	88
Figure 5.9: Autonomous System Design Service recommends links between subsystems based on the parameters defined in a given set of ACIs.....	89
Figure 5.10: Services developed leveraging the ACI, enabling automation of various aspects in system modeling and design.	90
Figure 6.1: Publishing a model written in the Python programming language as a web service using messaging protocols.	94
Figure 6.2: Manually created ACI for a model that adds its inputs.....	96
Figure 6.3: Simple web-based form for creating an ACI using an open source form generator.	97
Figure 6.4: Simple validation provided by the generated form.	98
Figure 6.5: A generated ACI is shown in a confirmation dialog of the editor.	99

Figure 6.6: Block Factory demo from Googles Blockly.....	100
Figure 6.7: Blockly workspace for creating ACIs using visual programming.	102
Figure 6.8: Using Blockly to configure ACIs.	103
Figure 6.9: ACI generated for a model using a visual programming framework, Blockly.	104
Figure 7.1: System model diagram of the energy needs subsystem of a third world village (MacCarty, 2015).	107
Figure 7.2: Microservices architecture for subsystem models calculating Annual Energy Consumption portion of the energy sub need system model for a single chosen task, device and fuel combination.	109
Figure 7.3: Architecture of a decentralized design environment, that leverages microservices defined in this research for autonomous system integration.	111
Figure 7.4: Specifying the location of the repository of registered ACIs.	113
Figure 7.5: The library of available subsystems accessed by querying the ACI Repository Microservice.	114
Figure 7.6: The system modeling/design canvas.	115
Figure 7.7: Information of subsystems available to the user for composing systems by querying the ACI Registration Microservice.	115
Figure 7.8: Desktop client application component allowing the user to specify the location of the ACI Repository.	118
Figure 7.9: Desktop version of the modeling/design environment with the same “Addition” model as described in Figure 6.7.	118
Figure 7.10: Annual energy consumption model of Figure 6.2, modeled and designed in the desktop client.	119
Figure 7.11: Results retrieved from within a desktop client application interfacing with the FMS and the autonomous system modeling/design services.	120

LIST OF TABLES

	Page
Table 1.1: Components of a cloud-based systems modeling environment as defined by Suram et al. (2018).	7
Table 3.1: Submission status values for an ACI.....	40
Table 4.1: Level to subsystem relational table for a system derived using parameter information of the ACIs of the subsystems.	51
Table 4.2: Basic components of an enterprise messaging system (Hopfe & Woolf, 2004).....	55
Table 4.3: Channel mappings used by an enterprise integration inbound “Router pattern” implementation for design environment incoming messages.	70
Table 4.4: Channel mappings used by an enterprise integration outbound “Router pattern” implementation for design environment outgoing messages.	71

NOMENCLATURE

API	Application Programming Interface
CAD	Computer Aided Design
CRUD	Create Read Update Delete
DSL	Domain Specific Language
HTTP	Hyper Text Transfer Protocol
JSON	JavaScript Object Notation
REST	Representational state transfer
URL	Uniform Resource Locator
UUID	Universally Unique Identifier

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Mark Bryden for his mentorship over the years of my graduate career. His patience, guidance, encouragement, and forward-thinking inspired the effort for the ideas in this dissertation. I would also like to thank my committee members Dr. Richard LeSar, Dr. Abhijit Chandra, Dr. Mark Mba-Wright, and Dr. Arne Hallam, for their support throughout this research.

To my wife, Coleen, thank you for encouraging me to take on this challenge. Your loving support and the sacrifices you've made throughout this journey made all of this possible. To my parents, Ray and Flo, and my sister Shauna, your faith, love and prayers have always kept me going when the challenges of life seemed insurmountable during this process. I would also like to thank Sunil Suram for paving the way and laying the foundation for this work and of course for his friendship over the years.

ABSTRACT

The modeling tools available for engineering design and analysis are traditionally created in isolation with features and capabilities geared toward a particular domain, that is, that many traditional engineering modeling tools are isolated, unable to readily connect to or be integrated into a larger tool set. With the advent of cloud computing and the success of delivering applications built using a microservices architecture, a more modern approach would allow an engineering design application to be composed of smaller independently developed and contributed applications within an environment capable of executing those applications without modification to the environment. This is somewhat analogous to an application store with one main difference. In the engineering design and analysis case, the goal is to enable the coupling of the applications together to perform higher level analysis, whereas in the application store, most applications are used independently.

This work introduces an Application Coupling Interface (ACI), for declaring the semantics of the application programming interfaces (APIs) of a modeled subsystem, a central repository providing access to curated web enabled engineered subsystems via ACIs and an extension of an existing cloud enabled engineering modeling/design environment to incorporate a new messaging system capable of autonomously orchestrating the execution and exchange of data between the subsystems. Together, these components provide the basis for an extensible analysis and design platform that accelerates discovery and innovation through the promotion of contribution and reuse of web enabled engineering models.

CHAPTER 1. INTRODUCTION

1.1 Overview

The term “Industry 4.0” is widely used to describe a future in which increasing wireless connectivity, reduced latency, edge devices, and cloud computing coupled with emerging manufacturing technologies, machine learning (ML), and other technologies change society by creating smart factories, smart cities, smart farming, and other smart entities. While acknowledging the critical importance of artificial intelligence (AI), 5G, and other technologies, one of the cornerstones of this revolution is the integration of all types of information sources—data, sensor, model, human input, and others. The rise of the Internet of Things (IoT) and the Industrial Internet of Things (IIoT) has led to efforts in providing technical solutions to handling the unprecedented amounts of data generated by these technologies (Dastjerdi and Buyya, 2016). Recent research predicts that there will be 50 billion edge devices worldwide by 2030 (R. Miller, 2019). Developing applications that leverage a mixture of edge computing (intelligent edge, e.g., Microsoft Azure, 2019) and cloud processing (intelligent cloud, e.g., Microsoft Azure, 2019) is one of the biggest challenges of Industry 4.0. Meeting this challenge requires interoperability between the various cyber-physical systems. This integration/federation of devices is creating a new computational fabric and is providing new opportunities for engineering and science. An essential requirement of this integration is the full development of semantic interoperability, where the meaningful context of the information exchanged between systems (and components of systems) enables the integration and execution of these systems (Nilsson & Sandin, 2018).

While integration, federation, and semantic interoperability are central to this coming smart world, it is striking that in contrast to enterprise computing, much of engineering and scientific computing is still deployed as stand-alone applications. The ecosystem of tools and

technology available in the domains of engineering design and scientific modeling while plentiful, are highly specific and often isolated to a particular domain of the research. Many organizations and institutions attempt to develop their technologies and software in hopes of accelerating discovery and innovation; however, often, these efforts cannot be easily shared or leveraged in conjunction with other systems. Efforts at integration are usually conducted within the bounds of computational frameworks in which specific rules guide the development and modification of codes to fit within a specific framework but are not widely extensible to other computational frameworks or problems. Although there has been an effort across various industries to share innovation through open-sourcing their codes, the publication of findings and results, and providing common tools for foundational mathematical and scientific computation, today integrating these custom-developed engineering and scientific software systems is laborious and error-prone.

Academic institutions such as the Texas Advanced Computing Center (TACC, 2019), publishes their software and provide hardware services for research in computation and visualization related to engineering and science. Similarly, a national research laboratory, Lawrence Livermore National Laboratory, maintains a popular software portal (LLNL, 2019) for the use of the tools it develops in research. The Open Science Foundation (OSF, 2019), a consortium for promoting collaboration in research, maintains tools for researchers to query and publish data related to their research. There is even an open-source code repository maintained by the United States government (Code.gov, 2019) for developing and providing access to publicly available data. Although there are plenty of options, these tools and systems are disparate and do not have the necessary elements to support semantic interoperability.

Much of this can be attributed to the fact that researchers in the scientific and engineering domain are not focused on the complex task of providing an integration solution for use in other systems, instead they are concerned with leveraging existing available technology to integrate solutions within the context of their own custom solutions tailored to the requirements and constraints of their domain. Whether or not the tools they leverage and develop internally can be integrated with systems used by other scientists and engineers external to their research is not their primary mission. Making technology accessible to others is a business on its own, let alone providing an integration platform as a service (Domo, 2019). The temporal and financial cost of supporting such an effort is not in line with the mantra of research and development in science and engineering. There are some groups of software developers and researchers who are working together to provide tools to aid in advancing engineering and scientific businesses and research. These tools, however, tend to address a specific high-level domain such as visualization (VTK, 2019), simulation (SimScale,2019), computation (OpenFoam, 2019), or design (FreeCAD, 2019). Ideally, scientists and engineers could pick and choose parts of these tools as needed to compose a broader application, but no cross-discipline, widely adopted integration platform exists.

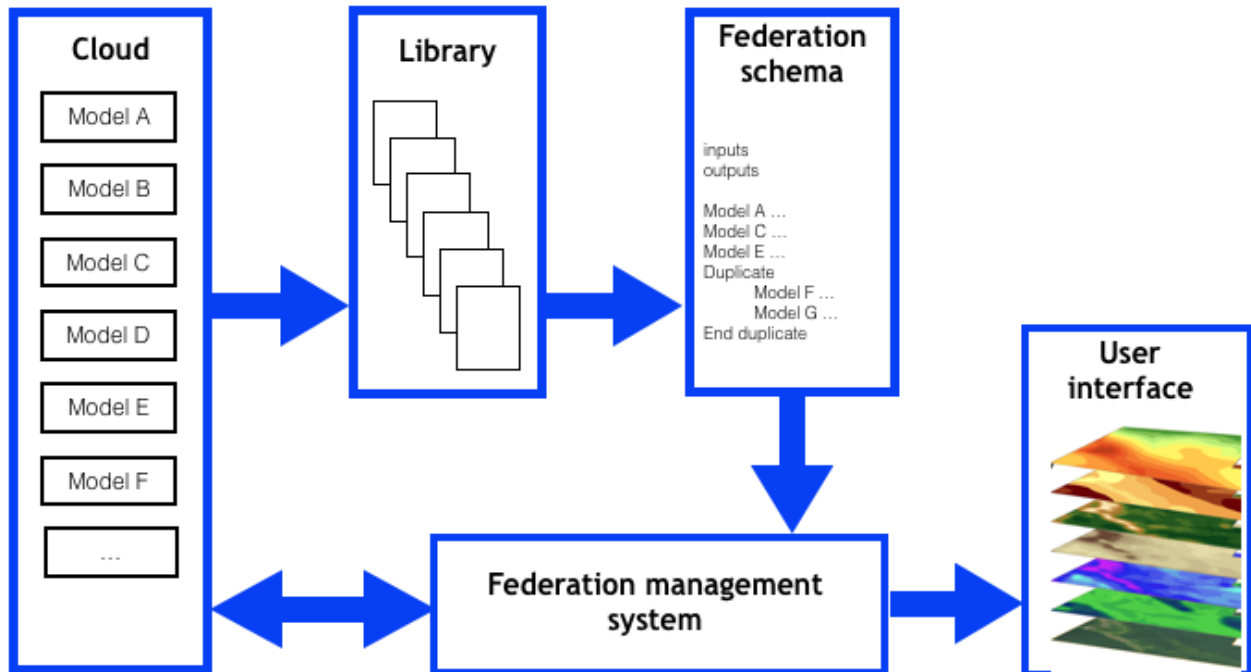


Figure 1.1: Engineering design and modeling environment leveraging cloud technologies (Bryden, 2014).

Recognizing this challenge, Bryden (2014) has proposed a novel modeling/design environment in which microservices within a cloud platform are leveraged to provide a “decentralized integrated modeling environment appropriate for engineering and scientific computing.” The goal of this work is to provide an environment in which independently developed models could be deployed and integrated into a wide range of applications (systems models) in the same manner as app development for the web. This federated modeling architecture defines an environment in which for a given system domain model builders can create and contribute stateless models that represent single functionality subsystems that are then available as microservices (Wikipedia, Microservices, 2019). These stateless models can be executed independently or linked together within the environment, representing a potentially complex system of analysis. The execution of the models is orchestrated by making the results of the execution of each model available to the collective model. As a starting point the

infrastructure for this modeling/design environment requires the development and deployment of a number of components including

1. an overall architecture;
2. an integration schema for the models that supports semantic interoperability;
3. an easily implemented method for bringing new and existing models into the environment;
4. a methodology for integrating the models together and creating the solution path through the system of microservice models;
5. a solution schema that manages the information and calls the microservice members;
6. front end components that support exploration of the available models, system assembly, and validation;
7. a cloud-based library of available models; and
8. user interfaces, including analytics to support exploration and understanding of the results.

Suram, MacCarty and Bryden (2018) developed the initial overall architecture, which includes components 1 and 5 of the needed infrastructure. This research develops component 2 and parts of components 3 and 4.

The initial overall architecture developed by Suram et al. (2018) can be summarized as a composition of several core components with the responsibilities as outlined in Table 1.1. This architecture lays the foundation for a platform whose capabilities can be easily extended by external contributions similar to other open-source technologies while evolving the overall platform. At a high level, the goal is to allow independently developed models to be registered with the design environment by developers, akin to the process of submitting a software

application to one of the popular mobile device application stores. At the heart of the architecture is the federation management system (FMS), which is responsible for orchestrating the execution of a user-designed system. The workflow is essentially an ordered execution (directed graph) of the subsystems required to represent the overall system. During execution, the platform maintains state, accumulating the necessary information from the preceding model's result to move forward to the next model in the workflow. The platform processes the workflow, schedules execution of the first model(s) in the flow, and upon a response from that model, proceeds to marshal the results to the next model(s) scheduled for execution. This sequence repeats until all models have executed or an error has occurred. As stated by the authors (Suram et al. 2018), information exchange and protocols are vital components for the successful implementation of the architecture. Stateless microservices represent the models, which means access and metadata about the information from the successful execution of one model to other models within the workflow has to be clearly defined as well as easily distinguishable. Otherwise, linking of the models becomes more of a task requiring too much of the technical networking and architectural knowledge inherent to the model developers as opposed to the system designer.

Table 1.1: Components of a cloud-based systems modeling environment as defined by Suram et al. (2018).

Component	Description
Solver Subsystem	Individual microservices registered with the platform representing subcomponents of an overall system.
Queuing System	Execution of the workflow via linear queuing of data exchange between subcomponents.
Federation management system (FMS)	Orchestration of data passing and execution of models for user submitted workflows.
Data Access	Persistence of the workflow execution state (model output for an execution).
Message contracts	JavaScript Object Notation (JSON) definition of FMS workflow information + data exchanged within the workflow.
Namespaces	Scratch space assigned to a registered model for storing intermediate information during an execution.
Web application programming interfaces	APIs to access the system.

The original federated modeling architecture and its implementation in software provide an environment in which model builders can create and contribute models that represent smaller singularly focused subsystems as microservices. These models, when appropriately registered within the FMS, can be utilized to design complex systems without requiring explicit knowledge of the implementation details of the models, thus providing a working federation modeling architecture. However, although this environment provides an extensible platform, it does not yet support the integration of models without user intervention. Currently, most of the workflow construction is a manual process of specifying input and connectivity within a particular data exchange format and then manually specifying the solution path.

This thesis develops the initial schema and algorithms to enable autonomous linking of models together consistent with a federated modeling architecture. To do this, the research

initially poses the question, how can a system be defined such that a modeler (e.g., engineer, scientist, researcher) can externally compose computational models and then contribute these computational models as elements for use as subsystem models to enable others (here referred to as systems builders or modelers) to easily use these computational models in arbitrarily assembled systems? Although these systems builders are experts in the system of interest, they may not be (and are likely not) experts in each of the domains covered by the models they use, and so the computational models distributed as microservices need to be able to stand on their own. This requires significant data orchestration and management of the execution of the involved subsystems to provide confidence in the system's ability to capture the behavior of the real-world system being modeled. Similarly, how can a subsystem developed independently of the system execution environment be contributed to the system while explicitly proclaiming its intent and means of execution without restricting the flexibility of choice of the developer? These questions and their relation to dynamic system integration are the main topics of this research.

To address these questions, this research proposes a simple lightweight specification for integrating scientific and engineering models without user intervention. Similar to an application programming interface (API), this specification provides a mechanism to abstract system integration information known by individual model developers into a form that is publishable to and consumable from a modern scientific and engineering modeling and design environment via modern web technologies. This integration specification is provisionally referred to in this work as an application coupling interface (ACI).

In this thesis, this integration specification, ACI, is then realized as a suite of software services running on cloud architecture. These services leverage the information provided in the

context of the ACI to replace various operations that currently require manual intervention during the process of scientific and engineering design. These processes include registration and visualization of models within the design environment, defining and editing of a system in the environment for execution, and validation of integration or “linking” of models within a system of models. Furthermore, the information provided by the specification can be leveraged by the design environment itself to enhance its capability to execute systems with complex integration and connectivity requirements (i.e., similar to a directed graph) while reducing the amount of data exchanged in the system by allowing concurrent model execution, as compared to the synchronous model execution of current architectures. This research will show that by incorporating the usage of services developed around the ACI into an existing scientific and engineering design environment, it will allow the iterative exploration of systems of varying complexity while taking advantage of the capabilities of modern cloud architectures. These components together provide the basis for an extensible engineering design and analysis platform allowing contribution and reuse of subsystems for accelerated discovery.

1.2 Organization

The remainder of this dissertation is organized as follows:

- Chapter 2 provides background on cyber physical modeling and the microservices architecture pattern of software development. It also discusses a recently developed scientific and engineering modeling environment that leverages the cloud architecture and microservices as a basis for describing some concrete problems around system integration.

- Chapter 3 introduces the definition of the ACI, which enables us to address the gap of manual system integration, particularly in the area of scientific and engineering design.
- Chapter 4 goes into detail about the messaging system of an engineering design environment and how incorporating the information provided by the ACI allows us to leverage enterprise integration patterns (Hoppe and Woolf, 2004) within an existing scientific and engineering design environment. In doing so, the environment is significantly enhanced in its ability to handle the increasingly complex yet more realistically defined systems of models.
- After defining the ACI, Chapter 5 describes the implementation of a set of microservices developed around the concept of the ACI that provides metadata about the ACIs for a model. It then defines a set of higher-level architectural components composed of the aforementioned tools that provides autonomous functionality for a client application to leverage.
- Chapter 6 discusses the implementation of the extensible design environment running on the cloud and the process of contributing subsystem models to the design environment, thus providing dynamic system integration of the environment. By creating and registering an ACI for the subsystem model, it can be curated and contributed to a central repository for access in the design environment.
- Chapter 7 discusses a prototype implementation of a design environment web application. The chapter goes into details about how the components of the application leverage the system ACI to provide an intuitive and extensible design

environment that can design and execute system models, moving closer toward the self-organizing systems described by Bryden (2014).

- Finally, Chapter 8 discusses the results of this research and summarizes areas of future research.

CHAPTER 2. BACKGROUND

2.1 Monolithic Modeling and Modeling Frameworks

With some notable exceptions, today there are two primary ways by which large-scale engineering and science models are built and deployed—monolithic models and closely coordinated modeling frameworks. In both cases the resultant model is generally deployed as a single software package on a specific set of hardware which may or may not be cloud-based. In the case of monolithic models, generally one team (or a small set of teams) develops and maintains the model. This maintains internal consistency but sacrifices scalability, flexibility, and extensibility to other modeling projects (Jamerillo, Nguyen, and Smart 2016). Specifically, the tightly coupled modeling approach required in monolithic modeling significantly slows changes and updates to the model and limits the use of the model for purposes not originally intended. An alternative to the monolithic modeling approach has been the development of modeling frameworks.

The applications of modeling frameworks are relatively broad, and the term “framework” can refer to both software libraries and software applications. In the Concise Oxford English Dictionary, framework is “a basic structure underlying a system, or concept, or text” (Soanes and Stevenson 2004). Generally, in systems modeling *framework* refers to “a software application that is the basic structure utilized to integrate, simulate, and understand complex systems” (Muth and Bryden 2013). Alternately Rizzoli et al. (2008) working in the area of environmental modeling and assessment defined a framework as “a set of software libraries, classes, and components, which can be (re-)used to assemble and deliver an environmental decision support system (EDSS) or an integrated assessment tool (IAT)”. Following this definition, Lloyd et al. (2011) divided modeling frameworks into “traditional vs. lightweight” based on “invasiveness,”

which they defined as the “degree to which model code is coupled to the underlying framework.”

Padula and Gillian (2006) introduce a number of ideas to the discussion of frameworks, including the need for validation and verification, knowledge capture, and user access to the results. As noted by **Bryden and Muth (2016)**, “one of **Padula and Gillian’s (2006)** key ideas is that many frameworks center on creating data repositories that tie information to the components they represent. These repositories then enable the users of the frameworks to seamlessly query information on a per-component basis.”

There are a large number of open and closed source modeling frameworks available addressing a wide range of problems. Searching Google Scholar with the terms “modeling framework” for articles published in 2018 and later yields approximately 16,600 results including articles titled “Water sector assumptions for the shared socioeconomic pathways in an integrated modeling framework,” “An integrated-process-property modeling framework for additive manufacturing,” and “An integrated modeling framework for energy economy and emissions modeling: a case for India.” These and other integration frameworks are generally based on coupling the models together by linking the inputs and the outputs via message passing between the component packages (i.e., models) such that adherence to a given data standard is maintained or alternately requires user intervention to identify and manage the data flow, and in many cases both. In addition, the concept of framework is most often utilized as a method for assembling and managing a project rather than ensuring the project is extensible. Because of this, many of these software frameworks are specific to the applications and are not readily extensible to other domains (or other problems within the same domain). There are a number of general-purpose frameworks, and the environmental modeling community has been active in the development of a number of general-purpose frameworks. Reviewing these frameworks **Suram**

et al. (2016) noted that the environmental modeling community has been active in the development of a number of general-purpose model integration tools. All of these systems require models to have initialize, run, finalize, get, and set functions for basic control over the models; provide a code-based method for connecting the models together, using some form of XML file or some other type of configuration file, to create a low-level model-to-model interface; and require an agreed upon global ontology describing the variables passed between models. Nearly all of these general-purpose integration frameworks require that the models in the system use the same programming language.

From this it can be seen that while modeling frameworks seek to simplify the engineering modeling workflow of model development, management, and deployment, the overhead costs of ensuring that models work together in a formally structured environment (i.e., a framework) make it difficult to create an adaptive and lightweight system of models that can be reused and extended to new applications.

2.2 Engineering Modeling within a Cloud-Computing Environment

At a high level the goal of the larger project that encompasses this research work is to create an easily extensible integration environment for scientific and engineering computing that is cloud native, supports user development and interaction with the integration environment, and can be coupled to a growing set of edge devices. That is, the integration environment being developed should be able to readily support interactive decision making about complex systems, create and maintain live digital twins, and support the development of cyber-physical systems including cyber-physical modeling. Suram (2016) proposed the following six principles for this type of distributed modeling environment:

1. The models and data developed for an engineering application should be deployed as an Internet based service.

2. The models used in the design process should be easily composable into complex systems of models capable of answering critical engineering questions. That is, models should be able to be invoked as and when needed.
3. These models must be readily publishable by their developers for use in systems models and analyses.
4. Models must be able to exchange information with each other via an intermediary service and must be able to join a federation of models in order to solve a larger set of problems.
5. To reduce the time needed to compute an answer from a detailed model, hybrid models consist of detailed computational models and reduced order models (ROMs). ROMs should be constructed on-the-fly in a manner that is transparent to the user and compatible with systems modeling.
6. Engineering teams should be able to substitute one model in a system of models for another model (with more or less detail or information) without disrupting the system of models.

Suram (2016) noted that implementing these principles requires a fundamental paradigm shift in the current modeling and design paradigm. In today's modeling paradigm

- Most detailed models enter the design and decision-making process as a single piece of information,
- When a detailed model is needed the starting point is often an entirely new model, and
- The creation of a large, complex systems model requires the development of a global ontology, a set of coupling protocols that is accepted and used by model builders at their points of interaction between the models.

As noted in Chapter 1, Suram et al. (2018) developed and deployed a cloud-based system modeling environment consistent with principles one and five discussed above. Critical to this implementation of this cloud-based systems modeling environment was the use of a microservice architecture and stateless computing.

2.3 Microservices

A microservice architecture pattern is based on the concept of composing “*small, autonomous services ... that are focused on doing one thing well*” (Newman, 2015) to form a distributed application. Rather than specifying a strict standard for defining a microservice, when developing a microservice a general set of principles should be followed as outlined by Newman (2015):

- A microservice should have a single responsibility and maintain autonomy.
- A microservice should isolate its failures so as not to impact other functionality within an application not related to the microservice.
- A microservice must be built in a way that they can be managed and deployed independently.

The popularity of microservices has arisen for a number of reasons such as reuse and composability of smaller functional components into a larger application that is loosely coupled to any one of the individual services it is composed of (Newman, 2015). Even so, their usage in a distributed application comes with its own set of concerns, requiring solutions that are distinct from those used in traditional monolithic application development, particularly in the areas of fault tolerance, scale, resiliency and idempotency (Newman, 2015). An application built on microservices has to deal with the fact that internet access is not 100% reliable, which means one or more of the microservices composing the application is likely to fail at some point in its

lifetime. This has to be planned for as part of the development of the services and within the context of the application.

Patterns have been developed and are available for many of these microservice architecture concerns, particularly in the area of connectivity. Robust implementations of resiliency and fault tolerant patterns are available in off the shelf technologies such as Resilience4j, which “*is a lightweight, easy-to-use fault tolerance library inspired by Netflix Hystrix, but designed for Java 8 and functional programming.*” (Introduction, Resilience4j, 2019). This open source library addresses concerns of resiliency such as retrying failed requests, caching, and circuit breaking.

Related to issues arising from failures in connectivity, any kind of failure during a data transaction causing an inconsistency in the integrity of the data could be catastrophic to the application if handled improperly. In most cases, distributed applications need to come to terms with the tradeoffs of handling data, commonly referred to as the CAP theorem (Gilbert and Lynch, 2002). The CAP theorem states that a distributed database can only have two of the following three properties:

- C—High data consistency, meaning at any one time if the data is accessed it is the same no matter which application is querying it.
- A—High data availability, meaning the service responsible for serving data from the database is available all the time, regardless of the consistency of the data.
- P—Partition tolerance, meaning the system can deal with network failures due to data replication.

While the choice of choosing two of the three is a design consideration that depends on the needs of the application, there are also off the shelf technologies targeted at these concerns, such as Apache Spark (Zaharia et al., 2012) or Apache Kafka (Kreps et al., 2011).

Another area of concern when it comes to distributed application development using microservices is that of integrating these microservices into functionality sufficient to deliver value to an end user in the application. Following a microservices architecture pattern when developing a distributed application implies that the application is likely to consist of many small services that need to communicate with each other over various web protocols. The integration of the microservices is challenging and requires substantial expertise to properly orchestrate the execution of the microservices, ensuring integrity of the application. Simplifying the process of integration has recently been the topic of research leading to integration languages such as Ballerina, “*a statically typed concurrent programming language, focusing on network interaction and structured data*” (Weerawarana et al., 2018). Although a tool or language such as Ballerina provides a mechanism for a developer to integrate a system of network services, it is targeted at the developer community as opposed to an end user who is not skilled in the art of software development or systems architecture.

2.4 Integration in Cloud-Computing

In Suram’s doctoral thesis (Suram, 2016) discussed above, the question of interactive integration of models and model management was left for further research. As systems models grow in complexity and size, manual integration of the components models creates significant challenges in deployment and maintenance similar to the drawbacks of framework packages—that is, the need for an agreed upon ontology and manual routing and assembly of component models. As originally envisioned by Bryden (2014), this integration challenge would be handled by the development of a library of standard contracts that could be cited by the metadata

associated with each model, enabling each model to self-identify to the larger system, which could then assemble the model with limited intervention by the systems modeling team and without the need for manual assembly and routing. The contract approach, however, is challenging in that the input and output data are not just shared between two models in a sequential fashion. Rather, the data are generated by one (or more) model(s) and parts, and this data are then needed by multiple downstream models.

This problem of system integration is not unique to the development of science and engineering software applications. The most common solution adopted for building large complex models in enterprise computing and software development is to acquire the resources of a software development team skilled in the specifics of distributed application development. This team is either currently knowledgeable in the application programming interfaces (APIs) of the microservices involved and begins to write code to integrate the usage of the microservices via their APIs, or they set out to learn the new APIs of those microservices they are unfamiliar with. This so-called middleware is the glue code where the orchestration of business logic is occurring in many of today's technology applications ("Middleware", n.d). Of course, this manual process of system integration has a significant upfront cost. Because the integration between any two or more APIs is almost certainly unique, any changes to the system (add, remove, or substitute in a new subsystem) essentially triggers a repeat of the process to adapt to the new system. For a business whose technology is following a microservice architecture pattern, a single microservice within that architecture and how it is used may not change often (quarterly/yearly). However, in a highly exploratory environment, such as those in the areas of research and development of science and engineering, the overall system assembly and the use and reuse of sub-models is apt to fluctuate often. In an ideal situation, an environment supporting distributed

scientific and engineering design could accommodate fluctuating system integration requirements without necessitating manual code changes.

System integration in the context of software technology is the process of assembling smaller software subsystems to represent a larger system (“System integration, n.d.”). The current rise in popularity of microservices architecture has led to system integration of distributed enterprise applications being an area of focus for many organizations (App Developer, 2018). The issues concerning coordination and execution of the subsystems as well as the handling of information exchange between the distributed subsystems led to the development of Enterprise Integration Patterns (EIP) (Hoppe, 2002), a set of cookbook techniques and principles to address the common architectural issues encountered in enterprise application development.

Many commercial businesses have been highly successful delivering products that leverage a microservice architecture. Netflix™ and Uber™ have risen to become global technology companies whose technical engineering success has been directly attributed to application development using microservices architecture (Nginx, 2015) (Uber, 2016, 2019) and enterprise integration patterns for addressing system integration issues. Yet, when looking at scientific and engineering technologies, they have been slower to adopt and migrate to this style of development for their technical solutions (IDC, 2019). Only recently have there been serious efforts at reimagining the approach to scientific and engineering design leveraging cloud technologies such as OnShape (2019). While the cloud can certainly address the scale and most cost concerns associated with science and engineering problems, such as data access and parallel computation, system level modeling, design and execution are still a significant challenge for adoption of a cloud-based solution. Both of these concerns lie in the area of system integration.

Some researchers have taken the approach of defining a domain specific language (DSL) or something similar to address the problem of integration (Balasubramanian, 2009). However, in most cases this only addresses the modeling aspects of integration, i.e., defining relationships between entities within the system. And because of the static nature of a domain specific language, these approaches cannot help with the iterative and exploratory aspect of design and modeling within science and engineering.

2.5 Application Programming Interfaces (API)

Application programming interfaces (APIs) are the means to programmatically access the functionality of a particular piece of technology. An API's documentation explicitly defines the functionality and how to invoke the functionality of the technology. The "how" is typically referred to as the protocol. APIs are very clear in expressing protocols and functionality however exchanging information between two or more APIs requires expert knowledge of the APIs.

When APIs are referred to, most people think of the REST (Representational State Transfer) architectural style for the World Wide Web that is based in the concept of statelessness (Fielding and Taylor, 2000), where statelessness refers to the fact that multiple invocations of the API with the same input will yield the same output. Consider a web-based model, plus-one, which acts as a counter. The plus-one model adds 1 to the last number each time is called. In the stateful implementation of the plus-one model where the model is tracking the results of the last invocation internally, multiple invocations of the model will each yield a different result. In contrast, a stateless implementation of the plus-one model requires a number as its input and then uses this number as the value to increment by one. Multiple invocations of the stateless version with the same input would always yield the same result regardless of the order of invocation or the number of invocations of the model. In the stateful case the plus-one model maintains the

state (i.e., the current count), and in the stateless implementation the state must be maintained by the system.

The recent rise in popularity of distributed application development and the use of microservices has led to an increased focus on the design and standardization of how REST APIs are described (OpenAPI, 2019). The OpenAPI Initiative (OAI) “*is focused on creating, evolving and promoting a vendor neutral description format*” (OpenAPI, 2019). These efforts are required for effective programmatic use and comprehension of APIs and the systems they represent; however, they are not necessarily concerned with the autonomous integration of systems providing the functionality of these APIs.

Given that REST is an architectural pattern, any API that follows the constraints of a client-server architecture in that it is stateless, is cacheable, has a uniform interface, and uses a layered system style can be considered to be a RESTful API. REST is one of the more commonly used protocols for public APIs because these are human readable and easily accessible. However, there are other protocols for communication between services within a cloud architecture that also provide stateless means of communication.

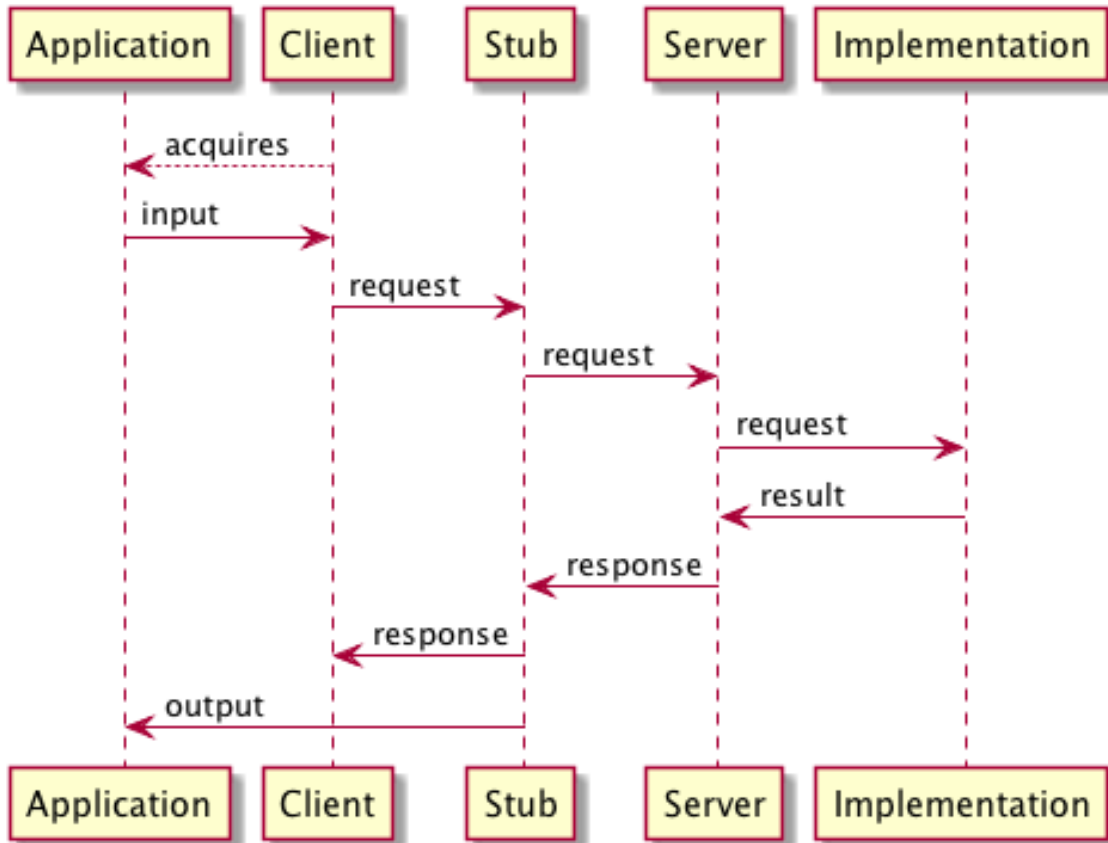


Figure 2.1: General sequence diagram of remote procedure call for an application.

One such protocol is a remote procedure call. A remote procedure call (RPC) is an HTTP protocol in which code executing on one server calls code that is actually executed on another server or process space. The code, however, is written as if it were calling the method or procedure on the calling server.

As illustrated in Figure 2.1., the general flow for an application communicating with a service using a remote procedure call as an API protocol is as follows:

1. Application acquires a Client for the method call to invoke.
2. Application sends input for a request to the Client.
3. Client calls the 'stub.'
4. Server, implementing the 'stub' receives the request.

5. Server performs the method based on the request.
6. Server creates the response and sends it back to the Client.
7. Client receives the response and sends it to the Application.
8. Application continues execution.

Another protocol for communication between services is message queuing. A message queue *“is a form of asynchronous service-to-service communication used in serverless and microservices architectures. Messages are stored on the queue until they are processed and deleted. Each message is processed only once, by a single consumer. Message queues can be used to decouple heavyweight processing, to buffer or batch work, and to smooth spiky workloads”* (AWS, 2019). In this pattern, there are generally three parts:

- Queue—Collection of a messages of a topic.
- Producer—Places messages on the Queue for a given topic.
- Consumer—Processes messages off the Queue for a given topic.

There are various implementations of message queues with subtle differences in how messages are stored in the queue and how they are consumed from the queue.

2.6 System Models as Microservices and the Problem of System Integration

In the design environment developed by Suram et al. (2018), system models are implemented as microservices and the system execution environment is designed upon a microservice architecture pattern. Many enterprise computing applications use a microservice architecture. In this architecture a single application is developed by orchestrating the communication of separate services, each of which represent an independently running business functionality. The execution of the services is predicated via their published application programming interfaces or APIs.

A common scenario for the development and delivery of business functionality as software in a microservice architecture can be described as follows: a business development manager decides that enough of the market is requesting a certain new feature or functionality. They work with product managers across different functional areas (database, frontend server side, etc.) to develop the functional specifications required to meet the needs of the demand. The product managers then work with the technical leads of the various teams to add specifications to the requirements and break down the specifications into technical deliverables. Once there is an agreement between all parties, the development of the feature is scheduled, and the development team begins work. Each team then works on a small but functional subset of the feature and creates a deliverable in the form of a microservice. Then there are teams responsible for writing the code to wire these subcomponents together to execute the business layer using the independently developed microservices. If the requirements change, more code needs to be modified or even new code has to be written to satisfy the requirements.

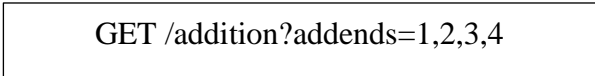
In this iterative development process, the cost implied by manual system integration is accepted as part of the process of the development of applications. For a domain such as science and engineering, changing the connections and links between subsystems and adopting a platform that incorporates models as microservices, which brings an additional “cost” for manually integrating the system, is a significant barrier.

In science and engineering, the system designer’s time is best spent in modeling, experimentation, and analysis. Ideally, to most effectively leverage the benefits of a modeling environment, where the subsystems are implemented as microservices, the aspect of modifying the system to handle configuration changes related to model linking and system design should be abstracted in a way that maintains the seamless exploratory nature of science and engineering

solutions. In other words, large temporal context switches are not conducive for keeping the system designer engaged in their activity or in supporting science exploration.

A given protocol for an API of a model is sufficient information for executing the functionality of the implementing service. In a typical scenario involving web services, a request conforming to the API of the service is invoked. The service typically has a validation layer for verifying the parameters of the request. Upon successful validation the request parameters are routed on to the implementation layer for execution of the model. A response with a code indicating success or failure of any kind (including validation) is returned. The response usually contains information of varying usefulness (as implemented by the developer) as to the results or its access or the reasoning for any error. These runtime validations and conditions are signals that there is some internally defined binding that is not easily expressible by the model's API. Of course, the variables and responses are stated clearly in the API, and in some cases this is sufficient.

Consider a single model that takes two inputs as an example. The explicit definition of the API, as shown in Figure 2.1, states how to call it, and it can be implied that the query parameter named "addends" allows a specification of some number of parameters that the model will attempt to add and return as the result.



```
GET /addition?addends=1,2,3,4
```

Figure 2.2: REST API defining the protocol for a service that calculates the sum of input numbers.

What is not clear from the API is the information pertaining to the inputs themselves. For instance, in some languages two strings can be summed, or for an even more abstract example, a number can be interpreted as a character and a set of characters can be summed, as in

concatenated into a single set of characters. If the model has a RESTful API, the URL for the resource is typically a “string” and it's up to the model implementation to interpret the parameters and validate them before executing.

Consider a technology company creating a new distributed application using a microservice architecture pattern. The integration of the various microservices in the traditional sense with only the information provided by the API of the microservice is typically performed during a standard development cycle. The iterative process of learning the APIs and integrating them to perform the required business functionality has relatively low impact on the end user of the application. In fact, until the microservices are fully integrated into the application, the effect of calling the APIs incorrectly only impacts the development of the application. This is a normal part of the development cycle, and its impact is limited to the development team. Once the integration of the APIs is complete for the business layer, it will have been tested sufficiently before being deployed. Also, as part of the development cycle, the final feature in the product will contain user interface level validation. That validation will be delivered as a way of keeping the user within bounds of the operating range of the static set of services that have been integrated to create the delivered feature. Once the feature is deployed, this microservice level code will have a very low probability of getting invoked with incorrect data.

In contrast, the integration of microservices representing subsystem models in an engineering design environment, needs to be dynamic to accommodate the exploratory nature of science and engineering design; redefining the system by changing the order of execution of the subsystems involved or interchanging subsystem models to evaluate trade-offs between execution speed and accuracy, for example, is a typical design scenario for a single workflow. Remapping of the overall system is essentially redefining the system integration of the design

environment. The execution of the model being designed is not protected due to the nature of this dynamic system integration. Without the protections in place that are outlined above, the cost of trial and error of iterative design is exorbitant. A system designer modeling and executing a complex scientific workflow cannot afford to wait 30 minutes to eventually have a failure due to one of the models resulting in an input of the wrong type. Similarly, if the output of a model resulted in an input that was “out of range,” it could also be costly. For scientific and engineering design to fully take advantage of the benefits of cloud technology and a microservices architecture as described by Suram and Bryden (2018), the issue of “dynamic system integration” needs to be addressed, particularly in a way that has a minimal negative impact on the users of the environment in terms of complexity of comprehension. Ideally, there is a representation of the subsystems and their API protocols that enable the system integration can be performed autonomously by the design environment.

2.7 Semantic Interoperability

The ubiquity of data combined with continually increasing availability of internet access and the relatively low cost of sensors used in instrumentation of systems has led to an evolution often termed Industry 4.0. This has also led to an emergence in the research of cyber-physical systems that refer to the integration of real-world physical objects with virtual computational representations. The physical system is instrumented to analyze and collect data about its operational conditions and state in real time. This information is also sent to the virtual components for a more complex analysis where any results requiring attention can be used to predict and even change operational conditions of the real-world system (Alam and Saddik, 2017). This wealth of available data can also be tied to the resurgence of popularity of artificial intelligence and machine learning technologies. For example, these technologies underpin the concept of the digital twin where the entirety of the data-driven virtual replica of the physical

system which, powered by AI, is theoretically able to respond and react in the same manner as the real-world system (Koulamas and Kalogeras, 2018). As the technologies supporting cyber physical systems become more mature, the issue will be less about deriving the algorithms and computational models to accurately represent the physical systems of the world but rather how can the information of these digital twins be integrated such that the impact of interactions between the systems provides an even more realistic representation of the real-world physical systems.

Looking forward to a world with 50 billion edge computing devices, each providing data that will be linked together to build value, it becomes clear that automated integration of information is a critical area of research and development. An essential part of this is ensuring that these interacting information sources are able to attribute the same meaning to data which is exchanged, that is, there is semantic interoperability between the various information sources (Murdock, Bauer, Bassbouss, and Loginov 2016). Murdock, Bauer, Bassbouss, and Loginov (2016) in their white paper “Semantic Interoperability for the Web of Things” identify two types of semantic interoperability.

Semantic interoperability is achieved when interacting systems attribute the same meaning to an exchanged piece of data, ensuring consistency of the data across systems regardless of individual data format. This consistency of meaning can be derived from pre-existing standards or agreements on the format and meaning of data or it can be derived in a dynamic way using shared vocabularies either in a schema form and/or in an ontology-driven approach. In this paper we will use the term "data-model based semantic interoperability" to refer to the former, and "ontology based semantic interoperability" to refer to the latter.

Currently, the common approach to the information (and model) integration problem in framework packages and other applications is to implement an *a priori* fixed global ontology in which consistency of meaning between packages is maintained by a set of standards/agreements, e.g., knowledge of data formats, communication protocols, access rules, or APIs,—that is, data-model based semantic interoperability. While effective, this data-base semantic interoperability requires that new applications and models know, understand, and comply with the existing rule sets, limiting the extensibility of the system and the ability to add new or disparate models.

Through dynamic derivation of meaning an ontology based semantic interoperability approach overcomes the challenges of a fixed global ontology but poses its own challenges. Specifically, dynamic derivation requires the development of a shared vocabulary and meaning. A number of researchers have proposed that metadata can provide the link needed to derive meaning, however, the implementation of metadata to achieve derived meaning is a complex and multilayered process (Murdock et al., 2016).

2.8 Conclusion

This research is focused on the challenge of developing self-identifying models that are able to be integrated into an engineering modeling workflow with a minimum of interaction. Consistent with that, a lightweight method is needed for models to dynamically identify how they can be integrated into a system and to provide the needed data to update the existing ontology. As envisioned here, these models exist within a closed environment in which the federation management system accepts membership and updates the system ontology based on information provided by the model in the form of metadata. Chapter 3 extends the concept of an API to provide the needed information in a form, provisionally referred here as an ACI (Application Coupling Interface,) that can be automatically read by the federation management system. In Chapter 4 the federation management system is then updated to ensure semantic

coherence is maintained (i.e., the model inputs and outputs are consistent with each other) and to manage the information flow.

CHAPTER 3. A SPECIFICATION FOR AUTONOMOUS SYSTEM INTEGRATION

3.1 Overview

As outlined in the previous chapter, an API is not sufficient for addressing the concerns related to the “dynamic system integration” that comes with the exploratory nature of system modeling/design. The intent of an API is an expression of usage, a protocol targeted at the programmatic invocation of the functionality exposed through the API. Although the availability of APIs somewhat lessens the burden of programmatic integration of systems, an environment providing autonomous integration of systems requires context about the information exchanged between systems that are not available in an API alone.

This chapter proposes a solution targeted at providing autonomous system integration of externally contributed subsystems models to a system model/design. Derivation of the solution is arrived at by considering the challenge of abstracting the orchestration of information exchange between independently developed web-enabled systems such that neither a system model developer contributing a subsystem model to the design environment nor a system designer composing a system from externally contributed subsystem models is impacted by changes in the design of the system.

3.2 The Challenge of Communication Between Dynamically Coupled Systems

There are many examples of technology companies developing distributed applications successfully by adopting a microservices architecture pattern. However, the process of iterating through various attempts of coupling the microservices via their API, i.e., integrating the system, is internal to the development process. The delivered product is a statically coupled system in which the user interface constrains the input to the system.

In contrast, in a system modeling/design environment, the delivered product is a dynamically coupled system, defined as appropriated by the system modeler/designer, and the success of execution is left up to the system modeler/designer to apply the proper constraints. While this may be achievable by someone with intimate knowledge of all the operating constraints of models relevant to their domain that are available in the system, it is not likely that this is the case in a decentralized modeling/design environment.

From a high level, the knowledge required of a user of a system modeling/design environment with the intent of composing and executing a system in an environment whose models are an ecosystem of subsystems implemented as microservices is as follows:

1. What is the availability of models within the environment applicable to the modeled system?
2. What are the capabilities of available models within the environment?
3. What are the requirements and constraints of the available models within the environment?

While the system designer can refer to a catalog containing all the available models in the environment, ascertaining the capabilities and requirements of those models to determine which are relevant to the actively designed system can be laborious, particularly if the system designer is unfamiliar with all the models available. This information is particular to each modeled subsystem and most likely resides with the developer who contributed the model.

From the subsystem model developer's perspective, the information they can provide to the system designer is vital. However, the context of the usage of the model is unknown, so the best tool available to communicate the intended usage of the model is currently through the API. The traditional context where an API is most useful is from the perspective of another developer

performing system integration of various APIs. If the developer wants to contribute their subsystem model to a modeling/design environment, allowing it to be dynamically integrated with other compatible subsystem models, the developer needs a way to provide the relevant integration information in a form consumable by the environment such that the environment can perform the integration without requiring manual (code) changes to the environment.

Again, from a high level, the knowledge required of a model developer with the intent to contribute models as microservices to an ecosystem of subsystems to be integrated and executed as part of the modeled system in a decentralized modeling and design environment is as follows:

1. What is the required information needed to integrate a model within the design environment?
2. What is the process of specifying the information necessary to integrate a model within the design environment?
3. For a given system integration specification, what is the process to register that information within the design environment?

In traditional engineering design systems, system integration is not as big of a concern because the system designer and the model developer are one and the same or on a small team and can communicate directly with each other. However, in a “decentralized modeling environment” (Suram et al., 2018), interchanging models that are not well understood and contributed from anyone with access and knowledge of the modeling/design environment comes at the cost of system integration, which can be prohibitive.

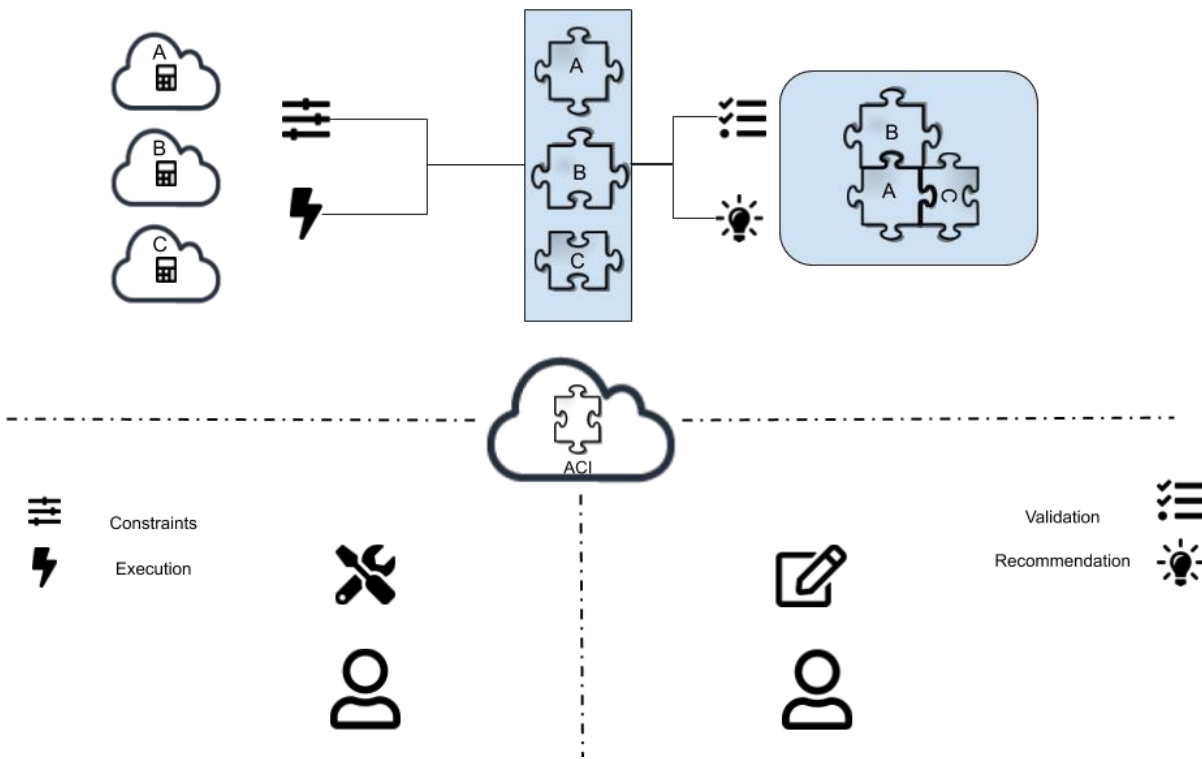


Figure 3.1: Two perspectives of an integration specification. Left, a model developer declares operational concerns of a model, such as execution protocol and parameter constraints of the model. Right, validation and recommendations are yielded to the system modeler/designer during the process of coupling system models.

What is desired is a way for the model developer to create models as they see suitable. If the developer then wishes to provide access to their model functionality within a “federation” (Suram et al., 2018), the developer is provided access to tools that allow them to define the model API protocol, the constraints of the inputs to the model, and other information that helps uniquely define the model within the federation. Upon submission of this information it would be organized according to an ontology, whose semantics describe the properties about a model, how to invoke and consume information from a model, and various information about the model’s behavior, i.e., a system integration specification.

Given an environment that provides access to a collection of system integration specifications, a user working within the environment can access system integration specifications as the domain proxy objects to the modeled subsystems. If the environment is able to leverage the system integration specifications to communicate with the APIs of the modeled subsystems for system execution as well as validate data exchange between subsystems, the user of the environment is suddenly enabled to iteratively build systems without requiring changes to the configuration or implementation of the environment. The user could assemble larger, more complex systems simply by using these proxies while the environment leverages the underlying system integration specifications as the proxy to the model and its API to validate and orchestrate execution of a designed system.

3.3 Application Coupling Interface: A System Integration Specification

The goal is to define a simple specification that can be easily expressed by a developer with the intent to contribute models as microservices to the design environment while providing enough information to a system so that it can validate and execute a system composed of these models as microservices. Noting the complexity of engineering system modeling, the ACI of this research is not exhaustive. However, it is sufficient for proving the concept and is flexible enough to modify it as necessary.

The Application Coupling Interface (ACI) is a lightweight schema definition written in JSON that defines the necessary means for integration of microservices with a well-defined API suitable for scientific and engineering design and simulation. The ACI consists of four distinct parts:

- *Metadata*—This information about the model that uniquely defines it within the federation. Some of this is explicit, like a globally unique identifier, while other parts can be unstructured, descriptions or tags for examples.

- *Protocol*—This is descriptive information about the protocol for invoking the API of the model. While this is unique to each protocol, the details have been abstracted in a way that the system can validate the ACI upon submission to verify if the defined protocol is supported.
- *Parameters*—These are the inputs and outputs as defined by the API that provide context such as units and measurement type. These can be of scalar form or vector (array) types with the only restriction that the array be of a single quantity type. These are used for linking validation during system integration.
- *Constraints*—These are tied to parameters and provide validation during linking and also during execution before passing quantities along to the next model as input.


```

{
  "uid": "b725a17b-046a-4ba8-8df9-0a78fd8e3d89",
  "name": "Addition",
  "protocol": {
    "protocolKeys": [ "io-fms-sqs-rout{
  "uid": "b725a17b-046a-4ba8-8df9-0a78fd8e3d89",
  "name": "Addition",
  "protocol": {
    "protocolKeys": [ "io-fms-sqs-routing-queue-name-key" ]
  },
  "protocolDetails": {
    "io-fms-sqs-routing-queue-name-key": "addition"
  },
  "messageType": "addition",
  "variables": [ {
    "uid": "ba56ffc2-acc7-46df-add3-d629fffb0d06",
    "name": "addends",
    "unit": "N/A",
    "typeName": "array",
    "quantity": "N/A"
  } ],
  "results": [ {
    "uid": "76851b90-4d3b-4720-9a77-edf7d54bc85d",
    "name": "total",
    "unit": "N/A",
    "typeName": "float",
    "quantity": "N/A"
  } ],
  "constraints": [ ],
  "version": "1",
  "description": "Adds inputs as numbers",
  "status": "SUBMITTED",
  "tags": [ "sample", "math", "addition" ]
}ing-queue-name-key" ]
},
"protocolDetails": {
  "io-fms-sqs-routing-queue-name-key": "addition"
},
"messageType": "addition",
"variables": [ {
  "uid": "ba56ffc2-acc7-46df-add3-d629fffb0d06",
  "name": "addends",
  "unit": "N/A",
  "typeName": "array",
  "quantity": "N/A"
} ],
"results": [ {
  "uid": "76851b90-4d3b-4720-9a77-edf7d54bc85d",
  "name": "total",
  "unit": "N/A",
  "typeName": "float",
  "quantity": "N/A"
} ],
"constraints": [ ],
"version": "1",
"description": "Adds inputs as numbers",
"status": "SUBMITTED",
"tags": [ "sample", "math", "addition" ]
}

```

Figure 3.2: A sample ACI for an “Addition” model that uses Amazon Web Services Simple Queue Service for its protocol

3.4 JSON Schema

As detailed earlier, the structure of the ACI is simple enough to be described in any web standard format for this work JSON Schema Draft 4 was chosen. The parts of the specification and how they are represented in JSON are detailed in the following sections.

3.4.1 Metadata

Each ACI registered within the environment has a section of required information that generally describes the model and its capabilities. It is required to have a universally unique identifier (UUID) and a name. The name can be anything the developer likes but it should typically be something that would be displayed to someone looking at a collection of specifications, so it is recommended to be human readable. The UUID on the other hand is something that the design environment can use to distinguish the specification from others, so it should be generated once upon the creation/registration of the specification. It is intended that this identifier will span the lifetime of a particular ACI even if other content within the specification changes.

Schema representation:

- *uid*—This property is the string representation of the UUID of the ACI. As of JSON schema version draft 4, UUIDs aren't supported as a primitive type, so this is a string. Validation of the format should be done upon registration.
- *name*—A human readable string for the name of the model represented by this specification.
- *description*—A human readable string for a description of the model and its capabilities.
- *tags*—An array of literal characters that can be used as an aid when applying search algorithms against a set of ACIs. These are optional.

- *version*—A literal character that is intended to help manage when changes occur in the definition of a particular ACI. These are typically numbers that increment with each published set of changes to the specification and are not necessarily related to changes in the API.
- *status*—This is an indicator of the state of the ACI with respect to the design environment. Table 3.1 lists the status states that are currently defined as of the time of this research.

Table 3.1: Submission status values for an ACI

Status	Description
SUBMITTED	The developer has submitted the specification for approval and registration within the design environment.
ACCEPTED	The ACI submission has been approved and validated for integration within the design environment. It registered and available for use within the environment.
REJECTED	The ACI submission has been rejected and is not registered and is not available for use within the environment.

3.4.2 Protocol

Recalling the various web standards available for calling a model such as REST or RPC, defining a generic specification could be tricky. However, in this research, a loose definition of the protocols was defined and is currently supported by the modeling/design environment. These registered protocols can be leveraged by an ACI editor and registration tools to restrict or validate the ACIs.

With this in mind, the protocol can be defined in two parts, the protocol description itself and the expected parameters or details required for specifying the protocol. For example, a

protocol definition for a model that is accessed using REST requires a URI for the resource endpoint and possibly an http request method. However, when thinking about a model that is based on messaging queues such as RabbitMQ (RabbitMQ, 2019) or Amazon’s Simple Queue Service (SQS) (Amazon, 2019), the protocol is quite different. It can be boiled down to simply needing the name of the queue to send messages to. To address this, the protocol is represented as a set of “keys” and a corresponding set of “key-value pairs” that together define a protocol. For anyone (or machine) processing the specification, the protocol can be easily understood, and the code can be developed to handle an individual protocol type by simply accessing the keys and then using the keys to process the registered values as defined.

As an example, a protocol and the protocol details for a REST model could be represented in the following manner:

```
{
  "protocol": {
    "protocolKeys": [
      "io-fms-rest-method-key",
      "io-fms-rest-path-key",
      "io-fms-rest-resource-path-key"
    ]
  },
  "protocolDetails": {
    "io-fms-rest-method-key": "POST",
    "io-fms-rest-path-key": "/world-peace",
    "io-fms-rest-resource-path-key": "http://problem.solver.org"
  }
}
```

Figure 3.3: Protocol and Protocol Details declaration for a REST protocol.

In Figure 3.3, the values corresponding to the keys of the protocol are defined for access to the model API, the actual method in the resource for invoking the model, and the HTTP method. Some resources have the same method for creating/reading/updating/deleting of a resource with only a variation in the http method. The details then specify the values for the corresponding keys, which can be used to reconstruct the method to call for execution of the

model. How parameters are sent to the model (path, query, or payload) could also be specified here but that is not described in the above ACI.

For a model using SQS as its API, the protocol and protocol details could be represented as follows:

```
{
  "protocol": {
    "protocolKeys": [ "io-fms-sqs-routing-queue-name-key",
      "io-fms-sqs-dead-letter-queue-name-key" ]
  },
  "protocolDetails": {
    "io-fms-sqs-routing-queue-name-key": "addition",
    "io-fms-sqs-deadletter-queue-name-key": "addition-dlq"
  }
}
```

Figure 3.4: Protocol and Protocol Details declaration for a model using Amazon Simple Queue Service as its API

In Figure 3.4, the protocol details declare the name of the message queue that the model is configured to respond to as well as the name of the dead letter queue where the model will send any errors that are registered. This is just an example of how a model could be registered. Another registration process could be that this information is created for the model developer and the infrastructure is setup for the developer. After vetting, the developer of the model is given proper access credentials to set up their model to access the queue created during registration.

SQS access requires proper authentication (as should most web services) for access. This could be incorporated as part of the registration of the ACI and the appropriate steps can be taken to provide access to the model from the modeling/design environment using tools such as cloud formation to create the appropriate Virtual Private Clouds (VPCs) (AWS, 2019) and Identity Access Management (IAM) (AWS, 2019) roles. The implementation details for establishing these within a cloud hosting environment is out of the scope of this research, but more details of what could be investigated in this area as it pertains to the cloud hosted modeling/design environment are left for future work.

Schema representation:

- **Protocol**—A grouping defining the protocol to execute the model. It contains the keys and the associated key/value pairs needed to reconstitute the execution of the registered model for a given protocol
 - *protocolKeys*—The set of keys unique to the protocol
 - *protocolDetails*—The values specific to a given ACI for the associated keys in the protocol

3.4.3 Parameters

The input to a model as well as its output can be represented as a combination of scalar values and as a vector of scalar values. These scalars, as pertaining to engineering and science, often have units of measure. To define these inputs and outputs, define each as a vector of parameters and group the inputs and outputs into separate vectors.

Schema representation:

- **Parameter** - An input to or an output of an API of a subsystem modeled used in a microservice architecture:
 - *name*—Human readable name
 - *uid*—Universally unique identifier (UUID)
 - *value*—Scalar or array of scalars
 - *unit*—Unit of measure that is consistent for the value
 - *typeName*—Representation of a “scalar” or an “array of scalars”
 - *quantity*—Quantity of the unit of measure (length, time, or velocity for example)

3.4.4 Constraints

With the *Parameters* defined as above, it is apparent that actual values are not part of the specification. This choice provides flexibility in the design of a system that provides validation of the values. Although a system can validate the matching of inputs to outputs while the user is designing the system by verifying that an output parameter definition is compatible with the input parameter definition for a set of given ACIs, validation of values of the quantities (magnitudes) produced from an execution of the model complying with the model constraints requires knowledge of ranges or operating conditions of the model.

Therefore, a *Constraint* is defined that applies to a particular Parameter and can be used to validate further that the output of an upstream model satisfies the conditions necessary to be used as input of a downstream model. This validation can also be leveraged by a user interface to restrict any user input values from being entered that may potentially cause an error in the operation of the subsystem model.

A *Constraint* can be expressed simply as a literal character arithmetic expression that can be interpreted by an expression parsing algorithm. There are various commercial and open source expression parsing frameworks available and the choice is not as important as providing mechanisms to express constraints and have the evaluation of constraints available within the environment. In fact, supported expression parsers and languages can be published by the development team of the modeling/design environment, and specifications can be validated against this during registration in similar validation tasks as the supported protocols.

Each constraint is defined by a literal character expression and mappings to bind Parameter values to variables in the expression. The expression is resolved and evaluated by an expression parsing algorithm by a provided framework.

Schema representation:

- **Constraint**—Conditions applicable to Parameter values supplied as input to a subsystem:
 - *uid*—Universally Unique Identifier
 - *expression*—Expression defining the constraint
 - *expressionVariable*—String representation of the variable in the expression to map to the value of the quantity that will have the constraint applied
 - *parameterId*—Universally Unique Identifier of the applicable Parameter

3.5 Summary

The ACI presented in this chapter provides a sufficient basis for abstracting system integration information of scientific and engineering design models developed using a microservices architecture pattern. By representing integration information as a proxy object to the underlying API of the model, a design environment built to leverage microservices can access the APIs of the microservices that implement the subsystems.

With the ACI established, the next chapter presents the challenges of orchestrating the execution of the subsystems of an intricate system model/design, which will be outlined, and a novel solution for autonomous system integration using ACIs.

CHAPTER 4. ENTERPRISE INTEGRATION PATTERNS IN SYSTEM MODELING

4.1 Overview

Chapter 3 introduced the ACI as a means of autonomously integrating subsystem models developed using a microservice architecture pattern into an engineering modeling/design environment. Chapter 3 also introduced tools to author, publish, and query information about modeled subsystems using their ACIs. In this chapter, rather than developing a new engineering modeling/design environment, two specific areas of the architecture presented by Suram (Suram et al., 2018) will be significantly enhanced such that the information provided by ACIs is used to enable autonomous complex workflow execution orchestration. In doing so, ACIs advance the capability of the environment to handle significantly more realistic system designs without adding additional complexity to the interactions or new requirements of the user.

4.2 The Current Landscape

The architecture outlined in Suram and Bryden (Suram, 2018) describes a system in which a user submits a prescribed execution schedule for the set of models representing subsystems within the overall system. The core of the system is the FMS, which handles the actual orchestration of execution, i.e., exchanging data on time for the execution of the appropriate subsystem. These subsystems are developed using a microservice architecture whose API is that of a messaging queue protocol. The FMS communicates with modeled subsystems by sending a message to the models prescribed topic on the queueing system. Upon receipt of a message, the model interprets the data it requires for input and executes. Upon completion, the modeled subsystem returns a message to the FMS on a prescribed queue. For this specific queue, the FMS knows the previous model completed, and the information is ready to pass along to the next model. This process repeats for each subsystem model until failure or completion of all

modeled subsystems in the prescribed workflow have executed. The FMS then returns the final message to the user who initially submitted the request.

```
{
  "solverlist":["addition", "multiplication"],
  "workflowID":" 80fc2d83-7fd7-45b4-a787-77012b587cc6",
  "input":8,
  "input2": 2,
  "input3": 3,
  "output": 30
}
```

Figure 4.1: Example workflow message as described by Suram et al. (2018).

There are two aspects of the scenario described above that can be problematic for a user attempting to leverage this design environment. Firstly, the workflow submitted to the FMS is required to be complete and correct in all aspects. Although its structure is simple, as shown in Figure 4.1, the derivation of the contents is not as intuitive. The main challenge is determining the correct order of execution of the models. In the current architecture, the user is required to know the correct order of execution, which is adequate for simple linear workflows where all data is accessed from each subsystem sequentially; however, to model more complex designs, it becomes quite cumbersome for a user to determine the proper order of execution of the subsystems. It is also very tedious (error-prone) for a user to determine the compatibility of available models for inclusion in an existing system model design.

Creating an appropriate user interface can partially address these issues but recalling that the current system is using a message queue as an API, the system itself cannot perform validation of data exchange between subsystems. The only validation that can be preemptively performed (before executing the workflow) is to verify that the message queue corresponding to the subsystems within the execution list is accessible. Beyond this, any other validation occurs

during the execution of the workflow, where it could be potentially devastating in terms of time lost, as described earlier.

In a queue system, the assumption is that the subscriber (consumer) explicitly knows the information they are interested in, and that is all they care about. In the FMS architecture, the FMS is only responsible for sending messages to the queues of the subsystems (based on its registry of models) as prescribed in the workflow, and it is blissfully unaware of what information it is sending along to the next queue in the workflow. So, if the order in the user-submitted workflow is incorrect, the FMS will send a message to the model with insufficient information to successfully execute. Similarly, as a response is received from the successful execution of a model, the FMS does not have enough information in place to determine if the received message has all the required input for the next prescribed subsystem in the workflow. The same is true in the case where execution times are various for models that are required to be completed before executing a model downstream in the workflow; a message queuing system alone isn't sufficient to manage the timing required to wait until all necessary information is ready before proceeding, i.e., sending the message along to the next queue.

4.3 Quick Wins

The previous section described some issues that prevent the engineering modeling/design environment developed by Suram (Suram et al., 2018) from being easily leveraged for the modeling of a complex engineering system. As a quick solution, the development of a simple user interface that leverages information from the ACI to build the systems conforming to the specification of existing linear workflows may suffice. There is already support for integrating subsystems with a message queue API as their protocol. By creating user interactions that check input parameter type definitions against output parameter definitions when a user attempts to make a connection in a linear workflow, visual feedback could be provided to accept or reject the

connection. By establishing a valid connection between subsystems, the list of queue names required for the solver list can be looked up using the protocol keys corresponding to the queue name of the protocol. The user could specify input parameters in the workflow message as before, and this would be sufficient for allowing the user to submit a “grammatically correct” workflow representing a designed system to the FMS. Although this is an improvement towards the productization of the design environment introduced by Suram (Suram et al., 2018), this does not address the issues of complex systems design or iterative exploration of modeled systems. However, with some modifications to the messaging system of the engineering design environment of Suram (Suram et al., 2018), the ACI provides enough additional information to the system to automate the orchestration of asynchronous subsystem execution, as required for more realistic complex systems.

4.4 Comprehending Realistic System Models

Simulating real-world systems requires a significant amount of data orchestration between the subsystems. Allowing the user to express their design intent to the environment in a way that provides the system enough information to execute the system accurately and as efficiently as possible is a challenge. Although some systems present the user with all the available options and settings, these are most relevant for addressing the system scheduling and execution, i.e., system integration concerns, as opposed to the modeling and design aspects that are of user concern. Simplifying the expression of the system design while allowing the underlying environment to extract the orchestration complexity should be an area of focus for any newly developed engineering design environment. The following sections describe a unique representation of the execution of the workflow derived from information available in the ACI.

4.4.1 Directed graphs and model execution, workflow representation

Accurate execution of subsystem models where the order determined by resolving the dependent parameter inputs with available outputs can be visualized as a directed graph consisting of relations between subsystem models within the system.

Bang-Jensen and Gutin (2008) describe a directed graph by the following:

“In formal terms, a Directed Graph is

- *V is a set whose elements are called vertices, nodes, or points;*
- *A is a set of ordered pairs of vertices, called arrows, directed edges (sometimes simply edges with the corresponding set named E instead of A), directed arcs, or directed lines.”*

For the workflow execution within the engineering modeling/design system, simply creating nodes for the models and edges as connections between the models is not sufficient for the case of a decentralized design environment. If it were, it would mean that every model connection would need to supply all the required inputs to execute a subsystem from a single subsystem output, essentially coupling the subsystems to each other. A linear workflow execution definition restricts the subsystem model developer to adhere to a standard that may not comply with their intended usage, therefore lessening the desire to contribute their subsystem models to the design environment. Ideally, the goal is that the developer of a model publishes their intended usage of the model, and the system can validate whether or not the model is suitable for use within the modeled system.

A directed graph, where nodes represent an instance of an ACI in a workflow, and the edges represent the bindings of an input parameter to an output parameter of a subsystem, allows subsystems to be developed as loosely coupled as possible and allows flexibility in the granularity of the assembly of a system.

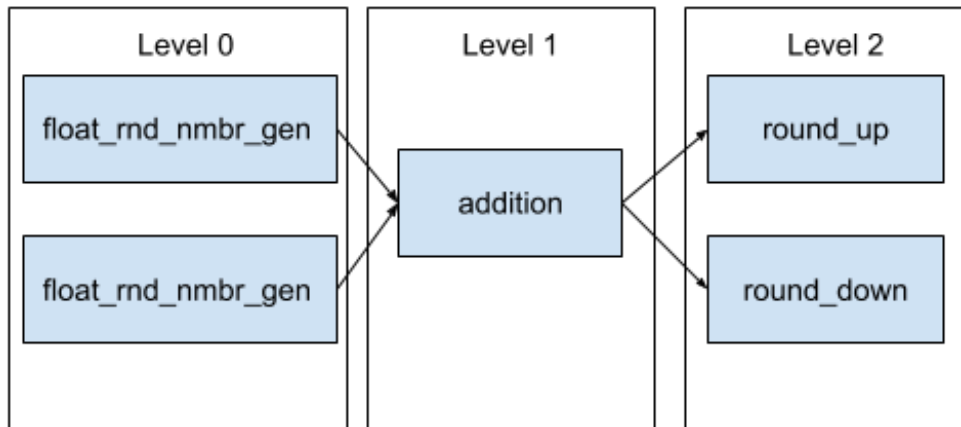


Figure 4.2: A system composed of ACIs grouped by execution levels as determined by traversing a directed graph of the connected parameters.

This graph could then be traversed breath first, and then for each level of the graph, a reference for each subsystem whose parameter is a node at that level could be stored, resulting in a relational table of a level to subsystems.

Table 4.1: Level to subsystem relational table for a system derived using parameter information of the ACIs of the subsystems.

Level	Sub Systems
0	<code>float_rnd_nmbr_gen</code> , <code>integer_rnd_nmbr_gen</code>
1	<code>addition</code>
2	<code>round_up</code> , <code>round_down</code>

Leveraging the directed graph removes the onus from the user for prescribing the execution order, which is not easily derived by a user. It also allows the user to define realistic system models, and the environment would be capable of resolving the arbitrarily complex

workflow execution order using the directed graph. The graph is then used to generate the table, which is submitted as the “workflow execution order for the system.” Execution of the workflow, however, requires that the environment is capable of efficiently executing the models prescribed in the relational table that is the new representation of the workflow. The following section describes how this challenge was solved, leveraging software architecture techniques widely used in the industry and the information available in the ACI.

4.4.2 Orchestrating execution of directed graph workflow

The ability to handle more complex workflows is not as straightforward as a solution as adding a user interface for a couple of reasons. As an example, imagine a scenario where the user would like to model a system in which two sensors send data for aggregation into tuples. Downstream models consume each tuple for further analysis, returning the result to the user.

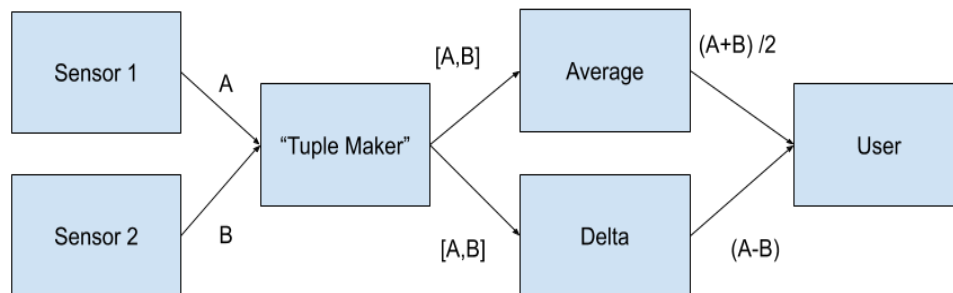


Figure 4.3: A “simple” complex system model design representing the aggregation of time varying data from sensors.

While this fictitious scenario is trivial in describing a system to be modeled, the system is sufficient for illustrating the same challenges that are present when modeling more realistic systems. For instance, the models representing the sensors are not guaranteed to send information at the same time, but it is a requirement of the system that the “Tuple Maker” model send corresponding tuples, i.e., data from sensor 1 and sensor 2 that occurred at the same time,

downstream to multiple models in the system. This challenge is also an issue for the sensors in the real world, and a developer would need to accommodate for this, perhaps correlating the data using logic based on time information available in the data read from the sensor.

There are other reasons for correlating data from different models within a workflow; however, in the case of modeling realistic systems, correlation of data produced and consumed by the related subsystems is essential to ensuring the correctness of the results, particularly if the design environment is running on a cloud architecture. In this case, multiple users have access and the ability to submit workflows for execution. The correlation of subsystem workflow executions should be accessible to the overall environment in a way that allows the system to execute multiple user workflows simultaneously and send data from upstream execution to downstream models while keeping the data and any results of transformations of that data, distinct within the context of a given workflow.

In the next phase of execution workflow for our “fictitious” system, the tuple output from the “Tuple Maker” model is required as input to both the Average model and the Delta model. These two models are entirely independent and can execute simultaneously rather than in a sequential manner, which would help decrease the overall time of execution of the designed system. The asynchronous execution of subsystem models requires the modeling/design environment to comprehend the sending of a single message to multiple models while again correlating the original data within the context of the workflow.

The problems outlined here can be generally stated as *“how can information and data be successfully ushered through a system where multiple transformations of the data and processing can simultaneously occur while maintaining its original context.”* This problem is one among many other related issues common to “Enterprise Applications,” which, as described by Fowler

(2002), *“are about the display, manipulation, and storage of large amounts of often complex data and the support or automation of business processes with that data.”*

Traditional engineering modeling/design applications are not thought of as “Enterprise Applications”; however, the concept of “routing information” is particularly of interest. Any industry-proven solutions suitable for handling issues of routing information between decoupled applications should be given consideration when developing a new distributed application, including those for engineering design and modeling. By leveraging the information from the ACI, they can be incorporated into the engineering design environment to overcome the challenges of routing information throughout the system while maintaining the originating context. In the next section, “Enterprise Integration Patterns” (Hoppe & Woolf, 2004) are introduced.

4.5 FMS 2.0: Introduction of Enterprise Integration Patterns

The heart of the engineering modeling/design environment developed by Suram (Suram et al., 2018) is the FMS which orchestrates data exchange between and execution of decoupled microservices by producing messages on a prescribed queue of a message broker such as RabbitMQ (RabbitMQ, 2019) or a managed solution such as AWS Simple Queue Service (Amazon, 2019) within a message-oriented middleware architecture.

A messaging system is a great solution that provides asynchronous, programming language agnostic, reliable communication between distributed applications. However, there are requirements to use it effectively. Hoppe and Woolf (2004) describe a set of patterns that guide developers in overcoming challenges often encountered when deciding to implement an architecture based on a messaging system.

Table 4.2: Basic components of an enterprise messaging system (Hoppe & Woolf, 2004)

Component	Description
Channels	The means in which data is transmitted within the system.
Messages	The immutable information that is sent from one system to another through a channel.
Pipes and Filters	Process a message and its information before sending it along to a channel.
Routing	Determines which channel a message should be sent to the next channel when it is not explicitly declared as part of the original message.
Transformation	Convert a message to a format compatible with a receiving channel.
Endpoints	Exposed means of communicating with applications external to the messaging system (Hoppe & Woolf, 2004, p. 95).

Considering the current architecture of the modeling/design environment of Suram (Suram et al., 2018) in terms of these components, the main message is the user-submitted workflow. The message payload contains input values relevant to the subsystems and an ordered list of message queues corresponding to subsystems in the system to call. Upon receipt of the initial workflow message, the message payload is transformed by adding information about the current execution step and forwarded to the channel corresponding to the subsystem of the current execution step. That channel then sends the message to the appropriate queue of the messaging broker endpoint with the corresponding input. The architecture receives messages from subsystems on an "incoming" endpoint.

When received, the message is transformed in the same manner as before, updating the current execution step and then forwarded to the next channel according to the subsystem list, repeating until the end of the list is reached. The system forwards the resulting message to a

prescribed "user result" channel, which then sends the result of the execution as a message to the queue of the messaging broker endpoint.

Although the original architecture defined by Suram (Suram et al., 2018) does not explicitly declare its usage of any of the components of a messaging system, it is clear that any principles or patterns that apply to a messaging system also apply to their defined architecture.

Simplification of the responsibilities of the underlying architecture of the modeling/design environment can be achieved with the use of some specific messaging components combined with the information of the ACI to enhance the overall modeling/design environment.

4.5.1 Refactoring

Recalling that the information of the ACI defined earlier in this writing can be used to facilitate the generation of a modeled system, and its subsystem execution workflow, the architecture of the design environment itself can be modified to accommodate these changes. A small change that yields a significant impact is the definition of the workflow execution. The workflow execution is currently defined as a linear list. The FMS orchestrates workflow execution, accumulating the results of each model into the messages forwarded in the workflow. Although the subsystems are stateless, this accumulation translates into the workflow execution message maintaining state during its execution. Without that state, access to information from upstream execution becomes challenging to manage. Accumulating the results is a requirement due to the synchronous execution nature of the current architecture. However, if the definition of the workflow execution is modified to allow defining asynchronous execution of subsystems, it would require the FMS architecture to handle asynchronous execution of subsystems, which will force state management to move outside of the workflow message and into some other aspect of the system. The integration patterns defined by Hoppe & Woolf (2004) provide advice in this area.

The general issue outlined with the asynchronous workflow execution is that of correlation. The implicitly managed state maintained by modifying the workflow message needs correlation outside of the messages sent to the modeled subsystems. For a given level of our new workflow execution definition, the FMS should wait for the receipt of the results of all executed subsystems for that level and then aggregate the results into a single message with appropriate input for the next level of subsystems. Because the engineering design environment presented by Suram can support multiple users accessing any of the models within different modeled systems (Suram et al., 2018), asynchronous workflow execution requires that information needs to be correlated in the context of workflows throughout the lifecycle of the execution to ensure the correctness of the result.

4.5.2 Approach

- To successfully perform the asynchronous model execution as described in the previous section, these are the areas to consider during its execution:
- When submitting a workflow for execution, the context of the workflow should be established and be uniquely identifiable.
- When executing a level of a workflow that has multiple subsystems assigned to it, messages should be sent to each of the subsystems that contain appropriate inputs for the execution of the subsystem maintaining the context of the workflow.
- When receiving messages from a subsystem, the system should maintain the context of the workflow by correlating messages of the workflow execution level using the workflow context.

4.5.3 Workflow context

The context of a workflow merely is transient state information about the workflow that can be accessed and managed by the system. For asynchronous execution, the two pieces of information are a unique identifier for the submitted workflow and the currently executing level of the workflow. With this information, the progress of the execution of the workflow can be tracked, the correlation of messages with a workflow occurs naturally.

Given a workflow execution derived from a directed graph of subsystem parameter bindings, as defined earlier, the execution of the entirety of that workflow would be defined as follows for each level of execution:

1. Upon receipt of a workflow execution, generate a workflow context
 - a. Generate a UUID for this execution.
 - b. Set the “current workflow level” to the initial level, “0”.
 - c. Create an initial message for workflow execution, specifying initial values in the payload of the message.
2. Process a level in the workflow
 - a. If completed at all levels, send result message to the user that submitted the workflow.
 - b. If not completed at all levels
 - i. For the “current workflow level” create a message for each subsystem prescribed to that level.
 - ii. Set an attribute of the message to the value of the workflow context identifier.
 - iii. Set parameter values in the payload of the message relevant to the subsystem’s input parameters as defined by the ACI.

- c. Send the individual message to each subsystem using the protocol specified by the subsystem's ACI.
 - d. Track the total of messages sent for this level.
3. Upon receipt of a subsystem execution complete message
 - a. Check the workflow context attribute of the message.
 - b. Group received messages by workflow context identifier and workflow level.
4. Verify that the number of messages received for that workflow context and workflow level is equal to the number of messages sent for that level for the workflow context.
 - a. If they are equal
 - i. Aggregate the messages for that group into a single message.
 - ii. Send the aggregated message to be processed, step 2.
 - b. If they are not equal, wait for next receipt of message, step 3.
5. Send result message to the user that submitted the workflow.

4.5.4 Relevant message system integration patterns for routing

Properly implementing an architecture for asynchronous workflow execution, as described in the previous section, has challenges categorizable under concerns of routing messages through the system. This routing is required to exchange information between loosely coupled applications typically using the APIs of the application, and recalling from Chapter 1, is a systems integration issue.

Hoppe and Woolf define integration patterns for messaging systems as “*nuggets of advice that describe solutions to frequently recurring problems*” (Hoppe & Woolf, 2004).

When designing a new messaging system, these patterns, based on the experience of developing messaging systems by experts within the industry over time, can be consulted for guidance.

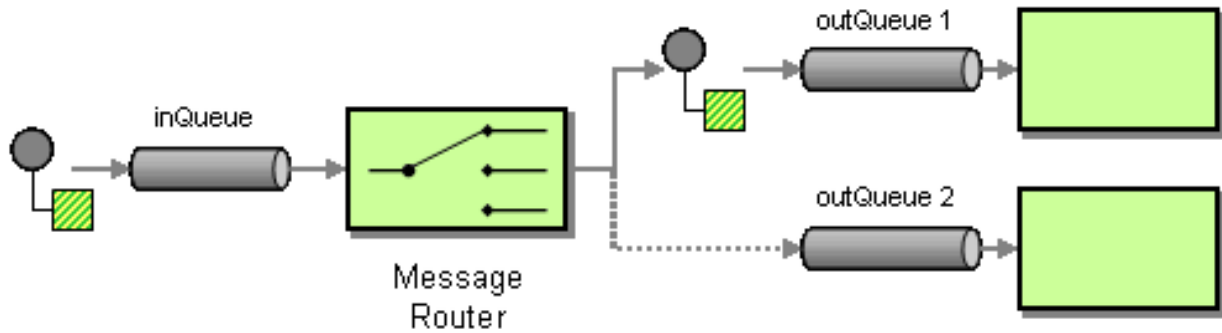


Figure 4.4: The “Router” pattern as defined by Hoppe and Woolf (Enterprise Integration Patterns, 2019)

Hoppe and Woolf define general routing concerns and propose a solution pattern as follows:

“How can you decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions?... Insert a special filter, a Message Router, which consumes a Message from one Message Channel and republishes it to a different Message Channel depending on a set of conditions.” (Hoppe & Woolf, 2004, p. 78).

Although generally broad, the statement primarily applies to the architecture of the engineering design environment sending messages to the appropriate subsystem for execution. It does not cover the issues of receiving a single message and sending relevant information of that message as distinct messages to several different subsystems. For that, there is a more specific routing pattern, the “Splitter” pattern.

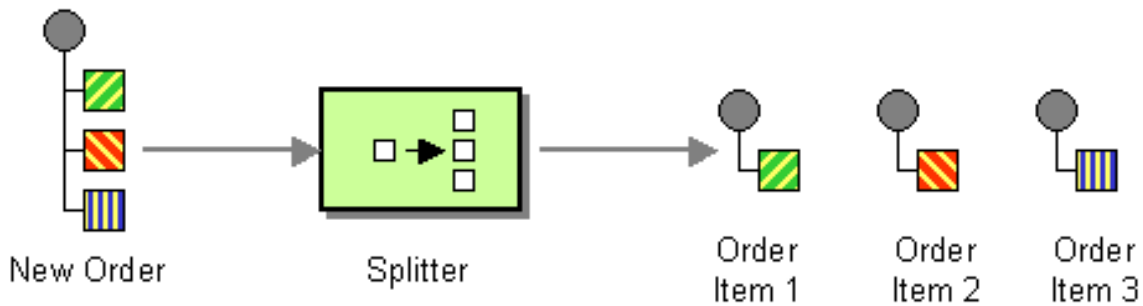


Figure 4.5: The “Splitter” pattern as defined by Hoppe and Woolf (Enterprise Integration Patterns, 2019).

“How can we process a message if it contains multiple elements, each of which may have to be processed in a different way?...Use a Splitter to break out the composite message into a series of individual messages, each containing data related to one item.” (Hoppe & Woolf, 2004, p. 259).

The “Splitter” pattern applied to single workflow messages distributes specific portions of the workflow message to subsystems with only relevant information to each subsystem, allowing asynchronous execution of the subsystems.

Another aspect of routing to be considered for asynchronous execution of subsystems is receiving messages from subsystems of a particular level for a workflow context. Once a message has been split into multiple messages and forwarded to their respective subsystems, how can the system correlate a message received from a subsystem to an originating workflow context. For messaging correlation concerns, Hoppe and Woolf define the "Aggregator" pattern.

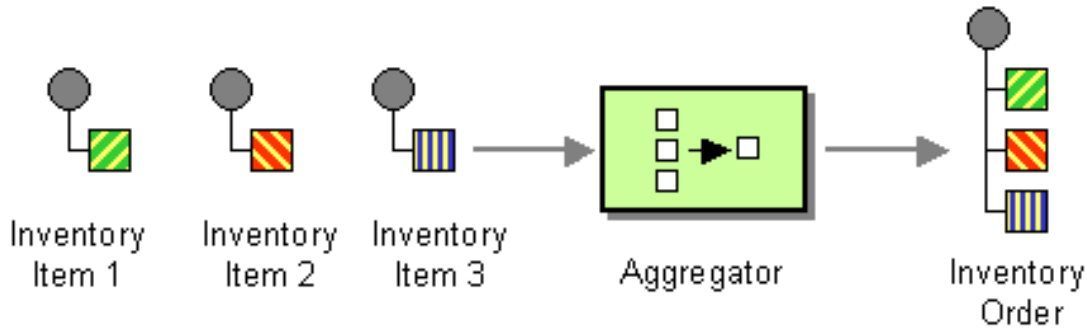


Figure 4.6: The “Aggregator” pattern as defined by Hoppe and Woolf (Enterprise Integration Patterns, 2019)

“How do we combine the results of individual, but related messages so that they can be processed as a whole?...Use a stateful filter, an Aggregator, to collect and store individual messages until a complete set of related messages has been received. Then, the Aggregator publishes a single message distilled from the individual messages.” (Hoppe & Woolf, 2002, p. 268).

The Aggregator pattern is a stateful pattern that relies on three key concepts:

- Correlation—How does the system identify related messages?
- Completeness Condition—When has the system received all (or enough of) the correlated messages?
- Message Aggregation—How does the system combine the correlated messages once the Completion Condition is satisfied?

An implementation of the "Aggregator" pattern must work in concert with a routing integration pattern defined upstream of its execution. The implementation of the "Splitter" must provide the correlation context to the messages as well as an indicator of what condition should be satisfied before aggregation of a group of related messages occurs, to accommodate asynchronous execution of models, as in the case of an engineering modeling/design environment.

The "Aggregator" implementation is where the state tracking occurs but is only required to be maintained within the system implementation of the Aggregator pattern as opposed to ensuring it is on the data being exchanged between the subsystems.

By using the enterprise integration patterns for concerns of routing within an engineering design environment, the Aggregator pattern in concert with the Splitter pattern provides the necessary mechanisms for orchestrating the asynchronous execution of realistically designed system models, whose execution order dependency is that of a resolved directed graph of bound parameters between the subsystems within the modeled system.

4.5.5 Implementation

The previous section outlines the requirements of an engineering design environment capable of orchestrating the successful execution of a user-defined system of realistic complexity, composed of subsystems represented as web services using a microservice architecture pattern. The following is an outline of the changes to the messaging system of the engineering design environment of Suram (Suram et al., 2018) which incorporate ACIs and enterprise integration patterns to address the issues of autonomous system integration and more realistic workflow design and execution as outlined in the previous sections:

- Conversion of the FMS to a messaging system capable of defining and using specific components as prescribed by Hoppe and Woolf (Hoppe & Woolf, 2002).
- Implementation of specific messaging components capable of leveraging information from ACIs of subsystems of a workflow:
 - Message System Endpoint
 - Sends and receives messages to/from the system using a prescribed messaging protocol.

- Sends and receives messages to/from subsystems using the defined protocol defined in the ACI.
- Splitter sends correlated asynchronous messages to subsystems within a workflow context based on the protocol details of the subsystem ACI.
- Aggregator receives correlated messages from subsystems for a workflow context based on the workflow context level and workflow context identifier.
- Router routes messages to the next message channel, particularly in the context of the next subsystem, using the protocol details of the ACI. Other conditions are applicable such as sending responses to the submitter of the system workflow.
- Workflow tools creates execution workflow from directed graph of parameter relationships.

4.5.6 Framework

Investigation of several off the shelf Open Source technologies, such as Apache Camel (Apache, 2019) and Spring Integration (Spring Integration, 2019) was performed, in hopes of expediting the modifications of the FMS to leveraging the ACI and the enterprise integration patterns for a messaging system, For this work, Spring Integration framework was eventually chosen purely based on the author's familiarity with the Spring Framework (Spring Framework, 2019) and time constraints. The purpose of Spring Integration as detailed on its main website is as follows:

“Extends the Spring programming model to support the well-known Enterprise Integration Patterns. Spring Integration enables lightweight messaging within Spring-based applications and supports integration with external systems via declarative adapters. Those adapters provide a higher-level of abstraction over Spring's support for remoting, messaging,

and scheduling. Spring Integration's primary goal is to provide a simple model for building enterprise integration solutions while maintaining the separation of concerns that is essential for producing maintainable, testable code.” (Spring Integration, 2019)

The Spring Integration framework supports most of the integration patterns for a messaging system as defined by Hoppe and Woolf (Hoppe & Woolf, 2002) and is written in the Open Source Java (Oracle, 2019) programming language. The work presented here was not explicitly designed for a particular implementation of the enterprise integration patterns, so the details of the capabilities of the Spring Integration framework will not be discussed unnecessarily in this work.

When describing implementations of the referenced integration patterns, the discussion will be limited to the details specific to the context of the engineering modeling/design environment architecture. In particular, the implementations of the “Splitter,” “Aggregator” and the general “Router” pattern that leveraged the information in the ACI of Chapter 3 as used in the system, are presented.

4.5.7 Asynchronous subsystem execution of modeled workflows

Once a message requesting execution of the modeled system is submitted to the environment, the smaller independent tasks, can be defined as follows:

- Route a message to channel for processing based on the channel it was received in.
- Route a message to channel for processing based on the channel it was received in.
- Create an order dependent workflow execution for the subsystems of the designed model using ACIs.
- Create an initial workflow execution message containing the workflow execution and the values set for the initial level of subsystems using ACIs.

- Process a workflow execution message sequentially by each level where each subsystem within a level can be processed independently (asynchronously).
- Send a workflow execution message to a subsystem based on its protocol and protocol details of its ACI.
- Correlate by workflow context level, a workflow execution message received from a subsystem with others received into a single workflow execution message.
- Route a message to outbound Endpoint based on state of the execution of the workflow.

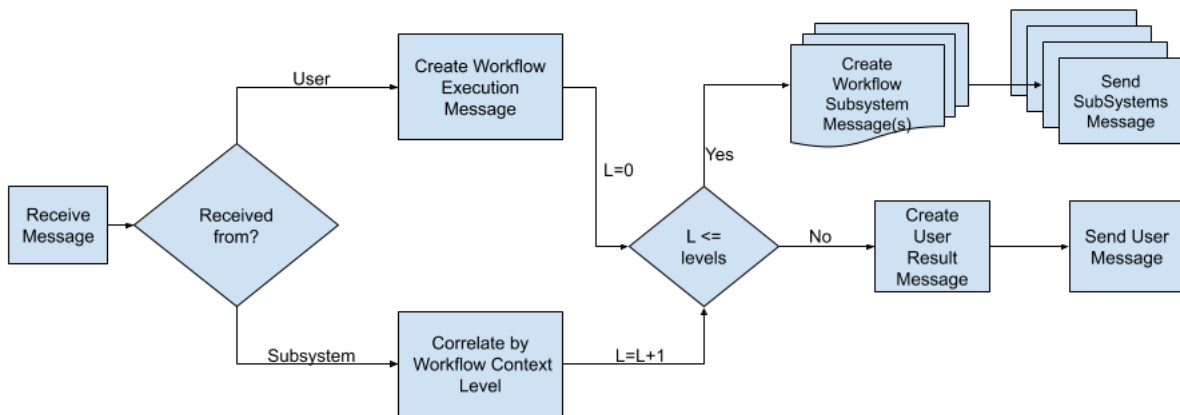


Figure 4.7: Flow diagram for asynchronous subsystem execution using a messaging system architecture and enterprise integration patterns.

As illustrated in Figure 4.7, an engineering design environment implementing a messaging system architecture can simplify the complexity of managing the asynchronous subsystem execution by breaking the execution down as a series of routing and transformations of the incoming message based on the origination of the incoming message. These transformations and routings, when chained together are described by Hoppe and Woolf (2002, p. 70) as tasks of a Pipes and Filter architecture style.

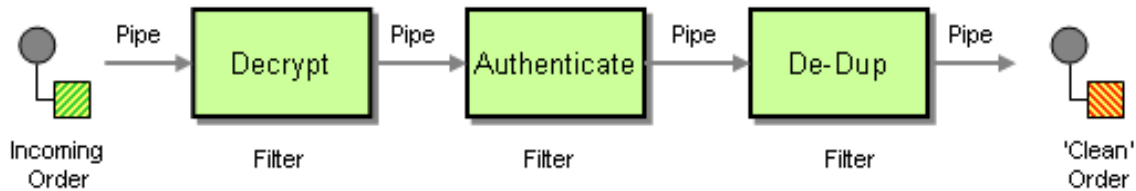


Figure 4.8: Pipes and Filters architectural style for a messaging system (Enterprise Integration Patterns, 2019).

“In a messaging system, the Pipe and Filters architectural style is recommended to break down the overall task of executing the workflow into smaller tasks that can be accomplished as independently of each other as possible” (Hoppe & Woolf, 2002, p. 70).

In this architecture style, “filters” perform any processing where the input is a message, and the output is the result of processing the message. “Pipes” connect filters and define the path of flow for the message between the “filters”. In the case of executing a user-designed system, modeled as an ordered execution of subsystems, the following two distinct routing and transformation filters for processing are obtained by combining the previously listed responsibilities.:

- Workflow Execution Task (Received from User)
 - Transformer - Create the workflow execution order message, grouping subsystems by level, based on the directed graph of dependent parameters created from the initial system execution request and the information in the ACIs of the subsystems.
 - Splitter - For each level remaining in the workflow execution order, create a subsystem execution message for each subsystem of that workflow context level.

- Router - Send each subsystem execution message to the appropriate subsystem using the protocol details of the subsystem's ACI.
- Workflow Level Subsystem Execution Task (Received from Subsystem)
 - Aggregator - Create a single workflow message from 1 or more subsystem execution response messages. State of the workflow context is updated by the Aggregator to reflect the current level of execution.
 - Router - Send a workflow execution message based on the remaining levels of workflow execution
 - If there are levels remaining in the workflow execution order, send the message through a Pipe whose Filter is the Splitter of the Workflow Execution Task for each subsystem of that workflow context level.
 - If there are no levels remaining in the workflow execution order send a message to the user response channel

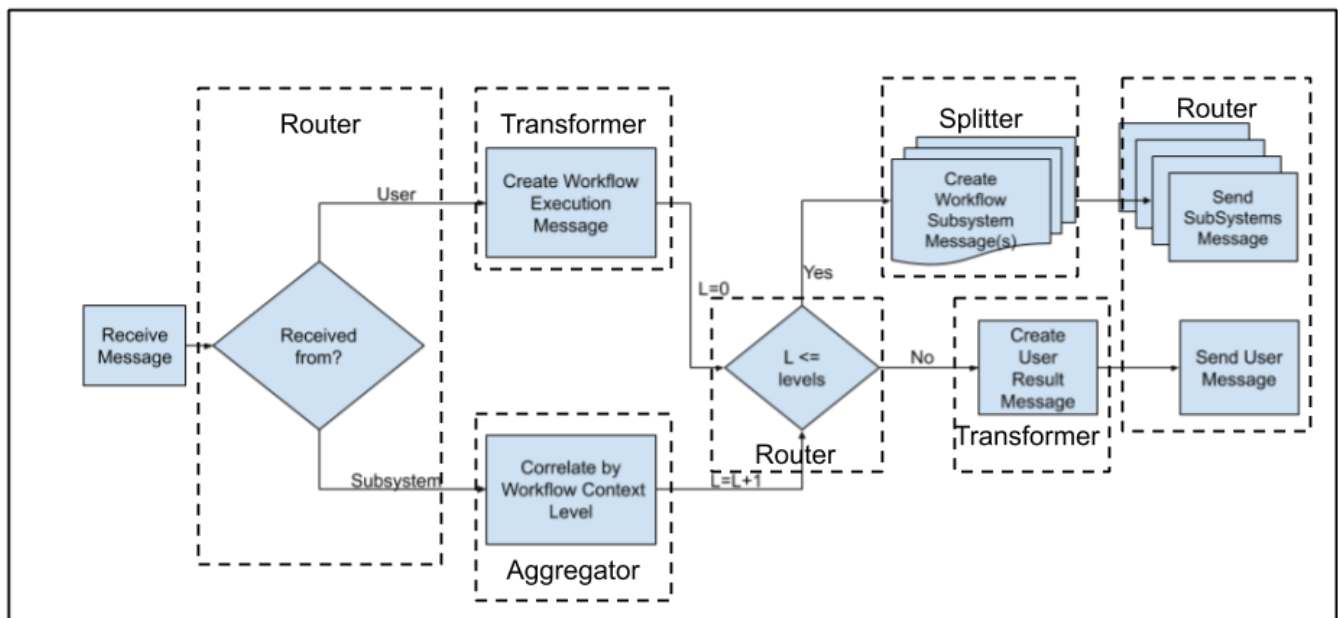


Figure 4.9: Enterprise Integration Patterns applied to the engineering modeling/design environment following a messaging system architecture.

With the “filters” defined, it is a matter of prescribing the “pipes”, i.e., message channels, connecting the “filters” necessary to perform the tasks of the system execution.

4.5.8 Routing inbound messages

The functionality of the engineering design environment, as defined in this work, is invoked using the protocols of the messaging broker. As such, there is defined a primary messaging endpoint that communicates with a message broker, such as RabbitMQ (RabbitMQ, 2019) or Amazon Simple Queue Service™ (Amazon, 2019), that is able to subscribe and consume any of the messages posted to topics as designated by the implementation of the messaging system, i.e., the engineering design environment. By connecting the messaging endpoint to a router implementation that forwards the incoming message to the channel designated for either the "Workflow Execution Task" or the "Workflow Level Subsystem Execution Task", the system can dynamically process the relevant messages. The router implementation developed in this work inspects the headers of the received message for the queue/topic in which it was received. This information is provided by the implementation of the message broker, in that it adds this information as an attribute in the header of the message. The router implementation can then forward the incoming message to the appropriate channel. Although this is specific to the implementation, the only requirement of a router implementation is that it can explicitly determine which task to invoke based on the incoming message. For example, the content of the body of the message could also be inspected to determine the destination channel.

The following table describes the channels that must be accessible in the message brokering system from both the engineering design environment and external systems, such as a user interface for submitting system designs for execution or an external subsystem used by a modeled subsystem, to successfully communicate between the systems.

Table 4.3: Channel mappings used by an enterprise integration inbound “Router pattern” implementation for design environment incoming messages.

Inbound Messaging Channel	Intent	Publisher	Message Content	Destination Task
Execution Request Channel	Execute a designed system	External system requesting the simulation of a designed system	User designed system	Workflow Execution Task
Subsystem Response Channel	Result of an execution of a subsystem	Subsystem	Results of an execution of a subsystem as defined by the ACI	Workflow Level Subsystem Execution Task

4.5.9 Routing internal messages

Internal routing of messages occurs internal to the messaging system of the engineering design environment and is dictated by the current workflow context level. If there are no remaining workflow levels in the workflow execution context, the message is routed to an outbound channel. If there are remaining levels, the message is routed internally to the channel of the “Splitter” to continue processing the workflow.

4.5.10 Routing outbound messages

Similar to inbound messages to the environment, there are outbound messaging endpoints registered in the environment corresponding to the supported protocols of the modeling/design environment. One is required for communicating with the user and is analogous to the incoming messaging endpoint in that it should be of the same protocol as the incoming messaging endpoint for consistency. It could be confusing for a developer interacting with the design environment to publish a message to the modeling/design environment and then have to make an HTTP request using a REST protocol to retrieve the results. As described in an earlier chapter, the protocol-service is available to communicate with developers the supported protocols within the context

of the modeling/design environment so that the ACI of a subsystem is only allowed to specify such protocols. These same protocols correspond to the available outbound messaging endpoints, which the outbound router pattern implementation uses to send execution requests to a subsystem model. In this work, only a single messaging endpoint was implemented using the same protocol the messaging broker used for sending messages to the user. For sending messages to a subsystem model, the queue/topic of the message broker that a message is published by the engineering modeling/design environment is defined by the protocol details of the ACI.

Table 3.3 describes the channels that must be accessible in the message brokering system from both the engineering design environment and external systems, such as a user interface for receiving messages from.

Table 4.4: Channel mappings used by an enterprise integration outbound “Router pattern” implementation for design environment outgoing messages.

Outbound Messaging Channel	Intent	Consumer	Notes
Execution Response Channel	Response of simulation of a designed system	External system requesting the response to a request for the simulation of a designed system	As supported by the protocols defined in the protocol-service system
Subsystem Channel	Request execution of a Subsystem	Subsystem	Derived from Protocol Details

The flexibility provided by the router pattern implementation combined with the information of the ACI of a subsystem is a crucial component to automating the system integration of the environment. By abstracting the API protocols of the subsystem into the protocol details, an implementation of the router pattern can “dynamically integrate” two decoupled systems by merely providing the appropriate value bindings as inputs in a message

and invoking the API as prescribed by the ACI. A change in the connectivity of the designed system has little to no impact on the architecture of the messaging system as long as the protocol is supported by the environment. Similarly, the connectivity of subsystems is unrestricted by the protocol of any particular subsystem to exchange information. This is only limited by the protocols that are supported by the messaging system of the overall design environment. Once support for a new protocol is added to the messaging system of the design environment, and the protocol is published, new subsystems matching that protocol can be registered within the system used for modeling and design.

To add support to the messaging system of the engineering design environment for new protocols between subsystems, the following is required:

- Create the new protocol specification for the protocol-service.
- Add a new outbound messaging endpoint that can send messages with a provider of the new protocol. This may require a transformer to convert the output to a format suitable by the consuming API.
- Add a new inbound messaging endpoint that can receive messages with the new protocol.
 - May require a transformer to convert the input to a format consumable by the messaging system (attributes and payload).
 - Route this to the channel prescribed for the Workflow Level Subsystem Execution Task.
- Update the logic of the outbound router pattern implementation to route messages to corresponding outbound channel as dictated by the new protocol details.
- Publish a new version of the messaging system of the engineering design environment.
- Publish the new protocol specification to protocol-service

The enhancements to the messaging system of the engineering design environment of Suram (Suram et al., 2018) described in this research provide an environment capable of handling realistically designed system models while automatically updating the integration of the decoupled subsystems of the overall model. However, recall that the enhanced messaging system requires as input a set of connected ACIs whose parameters have been bound and validated. In Chapter 5, a new service that leverages the parameter and constraint information of the ACI is defined to simplify the process of defining inputs to the engineering design environment that are compatible with its enhanced messaging system.

CHAPTER 5. LEVERAGING THE ACI IN SYSTEM MODELING

5.1 Overview

With the ACI defined, there is now a basis for creating tools to address the issues defined earlier in providing dynamic system integration as a feature of an engineering modeling and design environment. The set of tools directly addresses the requirements of model developers with the intent to contribute their modeled subsystem to the engineering design environment as a microservice. Recalling from earlier, those are:

- What is the required information needed to integrate a model within the design environment?
- What is the process of specifying the information necessary to integrate a model within the design environment?
- For a given ACI, what is the process to register that information within the design environment?

In Suram (Suram et al., 2018), the authors described a need for an ontology describing the models but also a need for the models to be available to the environment without modification to the existing infrastructure, allowing a developer to freely contribute their work to something akin to an App Store™ (Apple, 2019). An ACI facilitates this by providing a mechanism in which a developer can clearly state the purpose of their contribution. An ACI also declares how to leverage the contribution in the context of a more extensive system. It is implicit that the developer is also agreeing to comply with a defined mechanism for integrating their models within the engineering modeling/design environment.

To further aid in the contribution of models within the environment, a few essential tools are required. There needs to be a repository or library of registered and validated ACIs available to

access the design environment. There also need to be some tools for creating a new ACI and editing an existing integrations specification. Once created, there should be a way to submit, validate, and register the specification with an existing engineering design environment.

The next section describes in detail an implementation of these tools and their architecture and how they relate to incorporating the ACI into the engineering design process. The tools were developed following a general microservice architecture pattern with a RESTful API and optional data access and persistence layer, as depicted in Figure 5.1. The services were developed only to the extent that they are sufficient to validate the concepts developed in this research and their merits to solving the problem of dynamic system integration in an extensible, modern engineering design environment.

5.2 General Setup

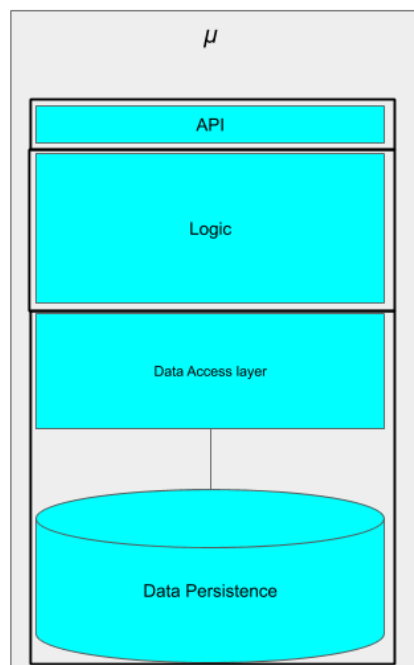


Figure 5.1: General microservices (μ) architecture used for supporting tools.

Each service uses off the shelf open-source software to bootstrap the components together into a deployable artifact. While deployment and issues of development operations (DevOps) are outside of the scope of this research, overall, this research maintained best practices and leveraged industry-standard tools while making some concessions for time. The services are designed using a general microservices architecture pattern, depicted in Figure 5.1, with a RESTful API defined for its protocol. The code is compiled and “containerized” (Google, 2018), and the container is published to a registry where they are accessed by any container orchestration platform such as Kubernetes (Kubernetes, 2019) or Amazon Elastic Container Service (Amazon, 2019).

The development of the code for the services uses open source off the shelf technologies. SpringBoot (Spring Boot, 2019), an open-source technology written in Java 8 (Oracle, 2019) programming language, provides the framework for quickly bootstrapping a web server and providing a RESTful API.

Apache Maven (Apache, 2019) is the build tool used for the microservices outlined in this work. Maven has a mature plugin ecosystem, which was leveraged to help automate some portions of the deployment, such as generating container images from the built code (Google, 2018) and eventual publishing of the artifacts to the Amazon Elastic Container Registry (Amazon, 2019). Provisioning of the services leveraged Fargate (Amazon, 2019) tasks using Amazon Elastic Container Service (Amazon, 2019). All code for these services is hosted in a Git source code repository, leveraging built-in continuous integration and continuous deployment facilities to automate the build, test, and deployment of the services during the development of the services.

5.3 Discovering Subsystem Models

In the context of the ACI, a “library” is simply a repository of submitted, validated, and registered ACIs that are web-accessible to an engineering modeling/design environment. An instance of the ACI is simply a JSON file conforming to the specification defined in the previous section; any web-accessible repository can host ACIs.

For the implementation in this research, a NoSQL (non-relational) database MongoDB (MongoDB, 2019) was chosen, running as part of a Fargate Task (Amazon, 2019) in an Amazon Elastic Container Service (Amazon, 2019) framework. The ACI Repository microservice provides access to the specifications in the library, following a simple Create Read Update and Delete (CRUD) style API generated using an open-source off the shelf technology, Spring Data Rest (Spring, 2019) which with MongoDB integration. The only real requirement for hosting a library of ACIs is that the API of the hosting service needs to provide easy access to the ACIs from any other decentralized service. Registered ACIs could just as easily reside in a cloud-hosted storage location such as Amazon Simple Storage Service (Amazon, 2019) or even a repository within GitHub (GitHub, 2019).

5.4 Authoring ACIs

As described earlier, the ACI schema is simple and relatively straightforward to understand; however, manual editing of the specification could be error-prone. This “ease of use” issue is easily addressable by creating an editor for the ACI. The main requirement is that the editor is capable of validating the input against a JSON schema for the proper structure. There are plenty of open source technologies available that a developer could use as an extensible desktop integration platform technology, like NetBeans (Apache, 2019) or Visual Studio Code (Microsoft, 2019), that provides out of the box JSON file editing and format validation. There

are even tools that will generate a web UI form based on a given JSON schema that can be used to create the editor.

Once the specification is created, and its structure validated in an editor, it needs to be registered with the existing library. Although access to the ACI Repository API is available via its RESTful API, directly calling this API is not recommended. In general, this API is an internal API to be called by other preprocessing services.

For example, requesting to create a new ACI in the repository will skip required steps such as validation of the constraints and protocols within the ACI and could cause problems down the line at runtime when it is too late in the process, leading to significant frustration that would cause a user in the modeling/design environment to reach out to the developer to determine the cause of the problems. Separate microservices have been created around managing access to registering ACIs within the environment to avoid encountering this and other possible frustrations after the ACI has left the developers' control.

5.5 Publishing and Registration

The publishing process is slightly more complicated than just the CRUD semantics of a standard datastore API. As described earlier, there are validation steps to perform on the submitted ACI before registering it within the library of available specifications. Rather than modifying the repository API to encompass this functionality, separate microservices are created to encompass the various validations and to perform the registration. One microservice has the responsibility of verifying the capability of expression parsing and validation. The other microservice maintains the responsibility of verifying functionality and validation of protocols. At the moment, these are the only two steps of the registration validation process, but splitting these out into separate microservices provides flexibility in the system so that as registration requirements evolve, the capability can be incrementally developed and tested without impacting

the current implementation. As it is deemed ready, the new capability can be integrated, and a new version of the registration microservice can be published.

5.6 Constraint Microservice

The constraint microservice is responsible for validating any constraints specified for parameters in a given ACI. The constraint expression is a literal expression that evaluates to “true” or “false,” and the constraint microservice provides two endpoints, one for verifying a given value against a constraint that is intended to be used by the modeling/design environment at runtime during the execution of a workflow. A separate call is used for verifying the expression syntax against the current expression evaluator for usage during the registration process.

The implementation in this research uses EvalEx (EvalEx, 2019) to evaluate and validate arithmetic constraints and have exposed a simplified version of their API as a RESTful protocol. Below is an example of a call to the API for validation of a constraint using cURL (curl, 2019).

```
curl -X POST "http://localhost:8081/constraints/validate" -H "accept: */*" -H "Content-Type: application/json" -d "{ \"expression\": \"a < 2
```

Figure 5.2: An example of a call to the API (/constraints/validate) for validation of a constraint using cURL.

For verifying the syntax of a Constraint expression, a similar API is available. Below is an example of a call to the API (/constraints/verifyExpression), again using cURL (curl, 2019).

```
curl -X POST "http://localhost:8081/constraints/verifyExpression" -H "accept: */*" -H "Content-Type: application/json" -d "{
  \"expression\": \"1 < a <= 2\", \"variableBindings\": [ { \"expressionVariableName\": \"a\", \"parameterType\": \"float\", \"valueAsString\":
    \"5\" } ] }"
```

Figure 5.3: An example of a call to the API(/constraints/verifyExpression) using cURL. It should be noted that while the payload of the request is the same as the validate API, the “valueAsString” field is ignored in the “/constraints/verifyExpression” API, as this method is only validating the syntax of the expression. The full API is listed as an Open API (OpenAPI, 2019) specification in Appendix A.

5.7 Protocol Microservice

The Protocol Microservice is responsible for providing access to protocols available and supported by a design environment. For the developer of an ACI, this microservice provides methods that can verify a protocol declared in their ACI by validating its expected keys and its defined “*protocolDetails*” with those of a given protocol registered in the design environment. The service also provides the currently registered Protocols supported in the design environment, showing the protocol’s required keys whose values should be set in an ACI “*protocolDetails*” mappings.

During the development of the tools in this research, the default supported protocol is for a messaging queuing system. Implementations of other protocols such as REST is left for future work. Below is an example of a GET request to the API (/protocols) for checking which protocols are available to the design environment, using cURL (curl, 2019).

```
curl -X GET "http://localhost:8082/protocols" -H "accept: */*"
```

Figure 5.4: Example request to query registered protocols of the environment using cURL.

The response from the call is a JSON object with the array of available protocols and its expected keys, as described in section 2.2.2. The values of the “*protocolDetails*” are specific to the ACI as described earlier in this chapter and should be specified when creating a new ACI. Appendix B contains the listing of the full API as an Open API (OpenAPI, 2019) specification.

5.8 Registration Microservice

The registration microservice is the service that a model developer will interact with when they are ready to contribute their model(s) to the modeling/design environment. The service does not maintain any state internally. It is merely a service to run the prerequisite validations on a given ACI before submitting the ACI to the library.

The validation process intends that a developer creates a new ACI and then invokes a request to the registration microservice with a JSON payload consisting of the ACI that the developer intends to register. Upon receipt of the request, the registration service delegates validation of the ACI to the Constraint Microservice and the Protocol Microservice to validate the specification given. Upon successful validation, as determined by the two services, the registration service invokes the integration service repository API to create a new specification in the repository. If validation fails or the submission itself fails, a response with relevant information will be returned. Otherwise, the response will return the location to access the ACI within the library. This location is not necessarily needed outside of the design environment because the ACI repository API will provide this access as needed. The API for registering a new ACI using the registration microservice is shown below using cURL.

```
curl -X POST "http://localhost:8080/integration/specification/register" -H "accept: */*" -H "Content-Type: application/json" -d "{ \"uid\": \"1e2b1d64-36c8-4177-b7e1-75598bb34a55\", \"name\": \"Test Registration\", \"protocol\": { \"protocolKeys\": [ \"io-fms-sqs-routing-queue-name-key\" ] }, \"protocolDetails\": { \"io-fms-sqs-routing-queue-name-key\": \"test-registration\" }, \"messageType\": \"testregistration\", \"variables\": [ { } ], \"results\": [ { } ], \"constraints\": [], \"version\": \"1\", \"description\": \"Testing Registration\", \"status\": \"SUBMITTED\", \"tags\": [ \"sample\", \"registration\", \"test\" ] } }
```

Figure 5.5: Example request to register a new ACI for a modeled subsystem in the design environment.

In terms of the “SubmissionStatus”, the registration microservice sets the value to “SUBMITTED”. As of this writing, all ACIs that are of the state “SUBMITTED” are available to the modeling/design environment. The author envisions future implementations of other

services that provide further setup and verification of the submitted ACIs, and the corresponding states will be set based on the results of those services. Correspondingly any design environment could adjust to filter out available ACIs based on a tolerable level of “SubmissionStatus” (preferably “ACCEPTED”). Appendix D lists the full API as an Open API (OpenAPI, 2019) specification.

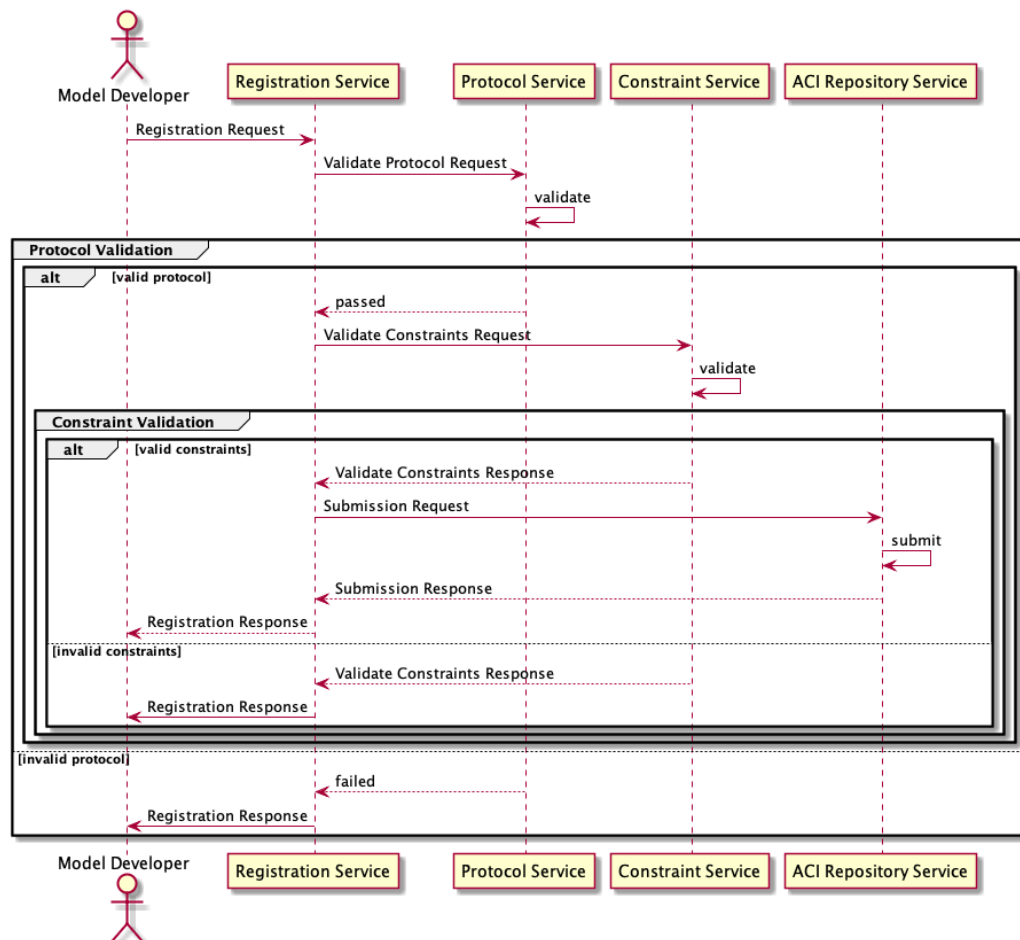


Figure 5.6: Sequence diagram for the registration of an ACI.

5.9 Designing realistic systems

Creating an environment for engineering design and analysis of a complex composition of subsystem models that is intuitive for a user is a challenging task. Most systems try initializing themselves in a way such that a user of the system can use it successfully without making many modifications to the settings outside of the realm of their expertise. For a so-called power user, “adjusters” are progressively exposed to provide more fidelity in the interface and inputs as a way to try to allow the user to provide as much detail as input to the system as needed to obtain a satisfactory analysis.

Bryden (2014) described an environment where the user “*chooses the subcomponents of the system they would like to design and linked them together*”... “*With the correct information, the technology systems of today should be able to tell a user if two components can link together relatively easily.*” Suram and Bryden (2016) laid the foundations for the architecture of such an environment, and this chapter lays out the enhancements to the environment to enable the system to handle the higher complexity of system models. That said, the problem of representing the modeled system in a way that can be leveraged by the enhanced messaging system of Suram and Bryden (2016) engineering design environment still remains; however, the information of the ACI should allow tools to be developed to solve some of the issues.

For example, the ACIs of two subsystems each express their required input types expected output types as well as details of restrictions on their inputs through their parameter and constraint definitions. This information can be used to validate connections and also recommend connections between subsystems by merely querying information about the subsystems in the environment through the library of registered ACIs. By aiding a user in this way, subsystem components can be added to a system by the user and connected while the underlying information is validated by the system using the relevant information of the ACI. Similarly, the

system could validate missing input or connections during the design of the system, preventing the user from submitting un-executable workflows to the messaging system of the engineering modeling/design environment. With that in mind, the next section describes an implementation of a microservice that was developed during this work to aid in simply creating and validating a workflow suitable for submitting to the engineering design environment for execution, a so-called workflow microservice.

5.10 Workflow Microservice

The workflow microservice is a composition of individual microservices (application) developed to simplify the creation of valid system design definitions suitable for execution in the messaging system of an engineering design system, such as defined by Suram et al., (2018) using the information of registered ACIs, as described in Chapter 2. It is an example API created as a conduit for a thin client application, to support defining connectivity of subsystems in a format suitable for generating the workflow execution in the context of the messaging system of the engineering environment described in the previous sections.

The service provides the following overall capabilities using a RESTful API as its protocol:

- Creating a new system representation (workflow).
- Adding/removing a subsystem to the system representation.
- Adding/removing user defined input from the system representation.
- Creating/removing connections between subsystems.
- Creating/removing parameter bindings between subsystem connections.
- Validating connections between subsystems.
- Binding user input values to input parameters of subsystems.

- Recommending parameter bindings for connections between subsystems.
- Validation of a system representation (workflow).
- Making a system representation of a workflow available to the engineering design environment.

The following areas of responsibility divide the workflow microservice API, which could ideally be split and managed as individual microservices, but in this work, maintained as one.

- Connections API
 - Creation and removal of “connections” between one or more subsystems represented by their ACIs.
 - Creation and removal of parameter bindings between one or more subsystems represented by their ACIs where a parameter binding represents the mapping of the result of one subsystem execution to the input variable of another subsystem.
- User Input API
 - Creation and removal of parameter bindings specified by the user.
 - Initialization and editing of parameter values specified by the user.
- Recommendations API
 - Recommend “connections” between one or more subsystems based on similarity of their parameters.
 - Recommend parameter bindings within connections between subsystems based on similarity of the parameters.
- Workflow API
 - Creation of new system execution workflows consisting of connected subsystems represented by their ACI.

- Addition and removal of a subsystem to a system execution workflow by referencing the subsystem's ACI.
- Validation of system execution workflows by verifying fulfillment of the subsystem requirements as defined by their ACI.
- Persistence of a system execution workflow that is accessible by the messaging system of the user.

The full API is listed as an Open API (OpenAPI, 2019) specification in Appendix C along with a programmatic illustration of the process of using the APIs in the workflow microservice to create a simple system model/design representation for a system that adds the user input and returns the result. The result of the illustration is a representation of the system that is compatible with the messaging system defined earlier in this work.

5.11 Applications of the ACI

Earlier, this work outlined the ACI and the details of various supporting services. Going back to the original idea of Bryden (2014), in which a decentralized system design environment aids a system designer in the composition of the digital representation of systems, i.e., cyber-physical system design, this research set out to develop the pieces of the environment that were remaining after the initial implementation of Suram et al. (2018). By introducing the ACI, the system modeling environment provides autonomous integration of the various subsystem models defined by an abstracted form published by their creator and consumed by the infrastructure on which they are running.

The four applications listed below, composed of the microservices mentioned earlier in this work, together provide the necessary pieces to begin implementing the ideas laid out by Bryden (2014) and demonstrated by Suram et al. (Suram, 2018). The services provide automation in areas of system modeling by leveraging the ACI, overcoming some of the

challenges of dynamically integrating systems as outlined earlier in this work. The composite services and a description of their functionality are as follows:

- **ACI Registration Service**—A service targeted at client facing applications that provides a means for subsystem model developers to extend the design environment by contributing reusable subsystem models. The subsystem models can then be accessed by a system designer in the environment, without requiring manual intervention by either the developer or the designer.
- **Autonomous System Design Service**—A service targeted at client facing applications that leverages an ACI to validate and recommend couplings between subsystem models by analyzing the parameters and constraints of the API as defined by a corresponding registered ACI.
- **Autonomous Integration Processing Engine**—A service targeted at use by other services, that leverages an ACI to generate a directed graph-like structure representing the required order of execution for a defined set of coupled subsystems by analyzing the parameters and constraints of the API as defined by a corresponding registered ACI.
- **Autonomous Routing Engine**—The messaging system of the FMS enhanced to autonomously route and transform messages between web enabled subsystems using enterprise integration patterns.

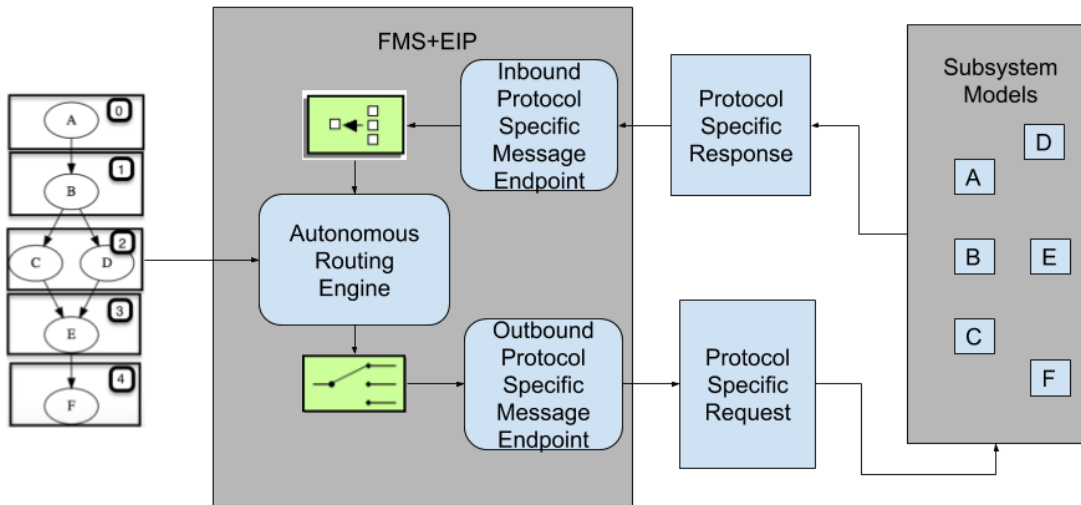


Figure 5.7: Autonomous Routing Engine leverages the protocols declared in ACIs to orchestrate execution of the prescribed system workflow using EIP in the FMS.

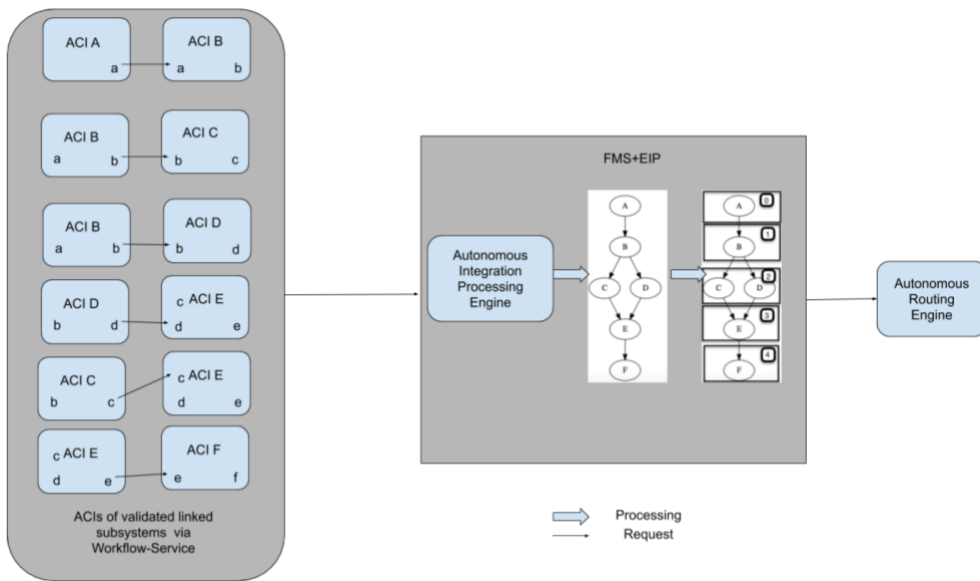


Figure 5.8: Autonomous Integration Processing Engine leverages the parameters declared in the ACIs to derive a directed graph of ordered execution of subsystems from a validated linking of ACIs from the Autonomous System Design Service.

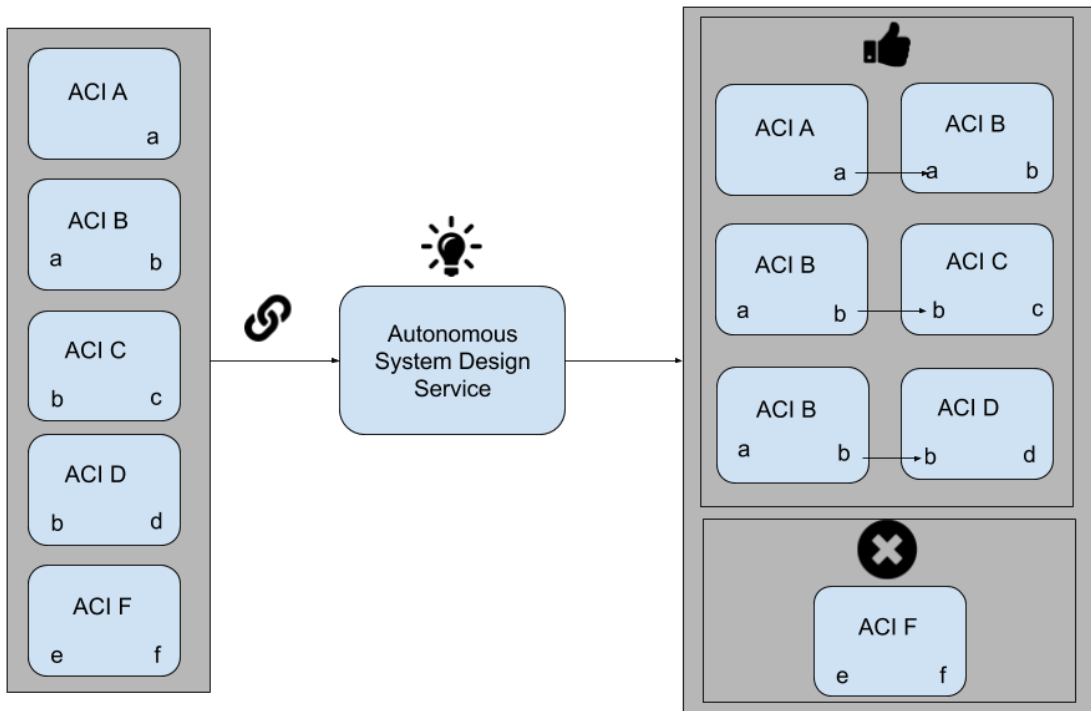


Figure 5.9: Autonomous System Design Service recommends links between subsystems based on the parameters defined in a given set of ACIs

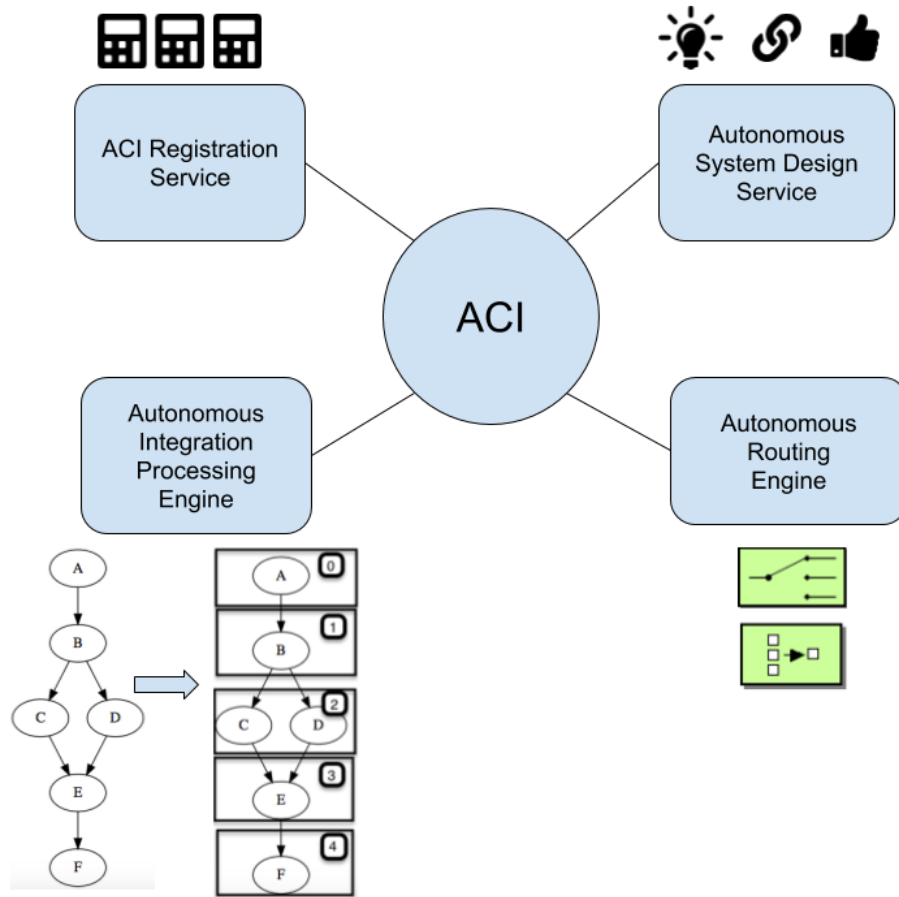


Figure 5.10: Services developed leveraging the ACI, enabling automation of various aspects in system modeling and design.

CHAPTER 6. AN EXTENSIBLE FRAMEWORK FOR SYSTEM MODELING/DESIGN

6.1 Overview

So far, this work has demonstrated how the ACI provides the abstraction of the communication and execution of the associated subsystem models of a larger system model. The details of how the messaging system of the engineering modeling/design environment can be enhanced by incorporating enterprise integration patterns and leveraging the ACI provides the capability for the system to correctly execute asynchronous workflows defined by complex user-defined systems modeled from real-world systems.

This work previously detailed the process of contributing individually developed subsystem models to the modeling/design environment by registering ACIs to a web-accessible central repository as well as the development of supporting tools and microservices that allow a user interface to be developed to access a deployment of the new FMS and ACI services to design and analyze systems of varying complexity.

This chapter will discuss the implementations of several prototype applications developed to leverage the services supporting the ACI. The prototypes are meant to demonstrate the contribution of modeled subsystems to the modeling/design environment by declaring an ACI and registering it with the central repository. These models are made readily available in the environment and can be used in the iterative design and analysis process by “linking” their representative ACIs in a declarative manner. As a proof of concept for future investigation, this chapter also demonstrates the idea of the modeling/design environment providing “hints” of compatible “linking” by merely leveraging information in the ACIs.

The new modeling/design environment was executed using a client application developed to support this research and deployed in a cloud hosting environment such as Amazon Web

Services (AWS, 2019). For the development of the required subsystem models as individual microservices, an API protocol that is a message queue was chosen. For each microservice subsystem, an ACI was created and registered with the design environment using the techniques and tools described earlier in this work. The microservices were each deployed and provisioned such that the services could consume messages from a messaging queue as prescribed by its ACI and could publish to the messaging queue corresponding to the workflow level execution task of the modeling/design environment as defined earlier in this work. The development of the subsystems and the microservices defining their functionality is not the main subject of this research, and the specifics of the implementation of the models are not discussed in detail here. Instead, the details of their requirements as it pertains to the system integration and microservice architecture principles (Newman, 2015) within the design environment are referred to in the context of their associated ACIs.

6.2 Contributing a Web Enabled Subsystem Model

A previous chapter described the registration of a microservice and its functionality as a means to allow contributions of independently developed web-enabled models to be registered with the modeling/design environment. The requirements to contribute a web-enabled system model are as follows:

- A web enabled system model, of which there is an API accessible by the messaging system of the design environment of one of its supported protocols.
- Access to the registration microservice of the design environment.
- A valid ACI for the web enabled system model that is to be contributed.

6.3 Creating a Web Enabled Subsystem Model

Web-enabled models can use various protocols for their APIs. For this work, the models use a messaging publisher and consumer architecture and the related protocols. Using Python

(Rossum, 1995) as the development language, a set of tools to create and publish a new model using this protocol can be managed in a few lines of code. Similarly, using managed cloud technologies for storage, messaging frameworks, and serverless computing, a model can be developed relatively quickly, and the tools will take care of setting up the infrastructure.

The process for creating the model begins simply by providing a developer with access to a template that has defined all the necessary methods for allowing the model to communicate with the deployed messaging system. Using a templating project generation tool such as cookiecutter (CookeCutter, 2019), the user will be prompted to specify some metadata information about the project, such as the name and the description. Upon completion, the new model will be generated in a directory of the model name and in that directory will reside a Python (Rossum,1995) file of the same name.

The result is a fully functional model and can be run using Python (Rossum,1995), but it is only implemented to return a message containing a sample response; it is left to the developer to create an implementation of the model by editing the code defined in the model file. The "main" function has a place holder for sample input that is useful for locally testing the functionality of the model using a local Python (Rossum, 1995) interpreter.

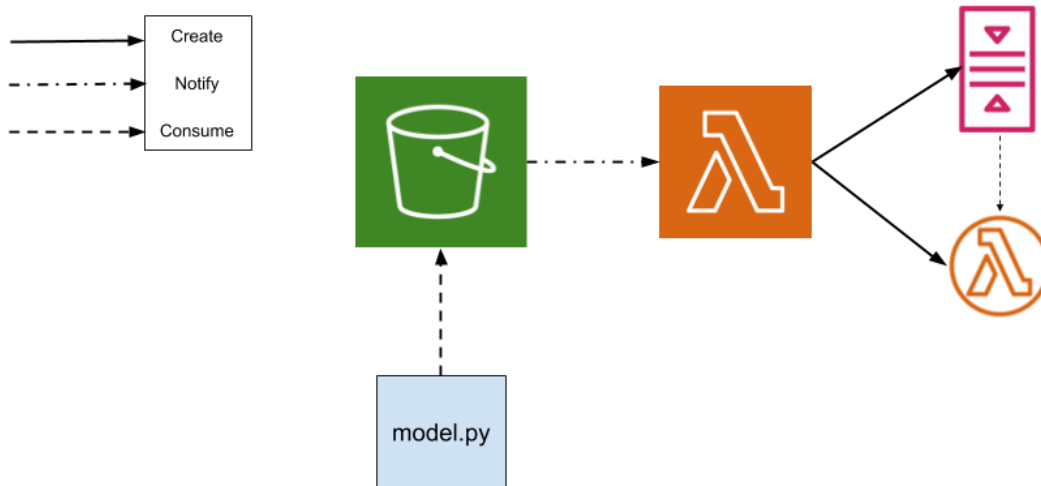


Figure 6.1: Publishing a model written in the Python programming language as a web service using messaging protocols.

Deployment into the managed cloud environment occurs once the developer is satisfied with the implementation of the model.

Figure 6.1 depicts a serverless computing pipeline established in this research to create the resources representing the web-enabled model. The pipeline is configured to respond to notifications of newly uploaded files to a specific cloud storage location. Upon receipt of a notification for a new model, the pipeline deploys the model as a new, separate serverless compute application and creates a message queue for communicating with the model. The pipeline then creates a notification mapping that executes the serverless compute application of the model upon receipt of a message in its queue. The resource identifier of the queue, which is the means of communicating with the model, is returned from the pipeline and should be set as the “queue name” in the protocol details of the corresponding ACI declared for this model.

Verification of the success of the deployment of the model and its ability to execute using the message queue can then be tested using the mechanisms provided by the cloud hosting

environment. In most cases, this can be achieved simply by sending a message to the message queue with the input parameters specified in the body of the message.

6.4 Creating an ACI for a Web Enabled Model

With a web-enabled system model satisfying the protocol requirements and providing access to the registration service, the generation of an ACI can be done in several ways with varying degrees of complexity. In the most basic case, the ACI has a defined schema, and rather than focusing on developing a user interface, leverage existing functionality of a JSON editor such as Visual Studio Code™ as outlined

here: https://code.visualstudio.com/docs/languages/json#_json-schemas-and-settings. The developer only needs access to the schema to allow the editor to validate any ACI. During this research, a Maven plugin has been developed to generate an updated schema specification whenever changes to the definition of the ACI specification are detected. Although this is certainly feasible and the editor will validate the syntax, there are still opportunities for errors, such as the “*uid*” fields. This field is a UUID (Universally Unique Identifier); however, in JSON, it is specified as a literal character array.

The effect of this error is a failure of registration, which could have been avoided. Similarly, the details of the protocol that the modeling/design environment supports should be explicitly defined by the protocol service, as described in Chapter 5, where the keys for a protocol are prescribed. Otherwise, these fields also cannot be validated until after the creation and attempt of registering (during the validation process) the ACI.

For the schemas that are available, there are published web tools that will generate generic forms for creating and editing an object according to its schema, parsing the schema, and creating a form specific to the type of the defined property.

```

{
  "uid": "b725a17b-046a-4ba8-8df9-0a78fd8e3d89",
  "name": "Addition",
  "protocol": {
    "protocolKeys": [
      "io-fms-sqs-routing-queue-name-key"
    ]
  },
  "protocolDetails": {
    "io-fms-sqs-routing-queue-name-key": "addition"
  },
  "messageType": "addition",
  "variables": [
    {
      "uid": "ba56ffc2-acc7-46df-add3-d629ffb0d06",
      "name": "addends",
      "unit": "N/A",
      "typeName": "array",
      "quantity": "N/A"
    }
  ],
  "results": [
    {
      "uid": "76851b90-4d3b-4720-9a77-edf7d54bc85d",
      "name": "total",
      "unit": "N/A",
      "typeName": "float",
      "quantity": "N/A"
    }
  ],
  "constraints": [],
  "version": "1",
  "description": "Adds inputs as numbers",
  "status": "SUBMITTED",
  "tags": [
    "sample",
    "math",
    "addition"
  ]
}

```

Figure 6.2: Manually created ACI for a model that adds its inputs.

The form contains the following fields and controls:

- Id:** A text input field.
- Uid:** A text input field containing the value `e5095312-2aec-4ea0-bc5f-ff4314831c7b`.
- Name:** A text input field.
- Protocol:** A section containing a list of Protocol Keys. One key is visible: `io-fms-sqs-routing-queue-name-key`. There is an `1` next to the key and an `Add item` button below the list.
- Protocol Details:** A section with an `Add` button.
- Message Type:** A text input field.
- Variables:** A section with an `Add item` button.
- Results:** A section with an `Add item` button.
- Constraints:** A section with an `Add item` button.
- Version:** A text input field containing the value `1`.
- Description:** A text input field.
- Status:** A dropdown menu.
- Tags:** A section with an `Add item` button.

At the bottom of the form are two buttons: `Generate` and `Validate`.

Figure 6.3: Simple web-based form for creating an ACI using an open source form generator.

An open-source example of such a tool is available at the following GitHub repository: <https://github.com/brutusin/json-forms>. Using this tool and directly loading the schema as described in the documentation, a resulting HTML web page will display a form that supplies input components to set values for an ACI. The library also provides minimal validation of the data entered into the form based on the schema. Extracting the generated data is also available through the library. Using a form such as the one in Figure 6.3, a developer with the intent to contribute a modeled system as a web service can specify the values required as defined by the schema specific to their web-enabled model to create an ACI directly in the form. A simple user interface for creating and editing an ACI can be integrated into any application as long as the ACI schema is made publicly available using a library such as json-forms.

The image shows a web form with several fields and buttons. A red box highlights a validation error message that says "Validation error" and "This field is **required**". The error is associated with the "Name" field, which is currently empty. Other fields include "Id", "Uid" (with a UUID value), "Protocol", "Protocol Keys" (with a list containing "io-fms-sqs-routing-queue-name-key"), "Message Type", "Variables", "Results", "Constraints", "Version" (with the value "1"), "Description", "Status", and "Tags". There are "Add item" buttons for the "Protocol Keys", "Variables", "Results", "Constraints", and "Tags" sections. At the bottom of the form are "Generate" and "Validate" buttons.

Figure 6.4: Simple validation provided by the generated form.

Figure 6.4 shows that by leveraging the schema to generate the web form, protections can easily be put in place to prevent incorrect data entry by defining validation of the fields within the form or using the built-in validation. For example, a regular expression validator can be added to the fields requiring UUIDs, to validate that the data entered into the field is of the proper format. In this case, this field is disabled, and the form auto-generates the UUID upon creating the new ACI. Similarly, submission of the new specification is performed by adding a user interface element that calls the registration service with the generated ACI using the API described in earlier in this work.

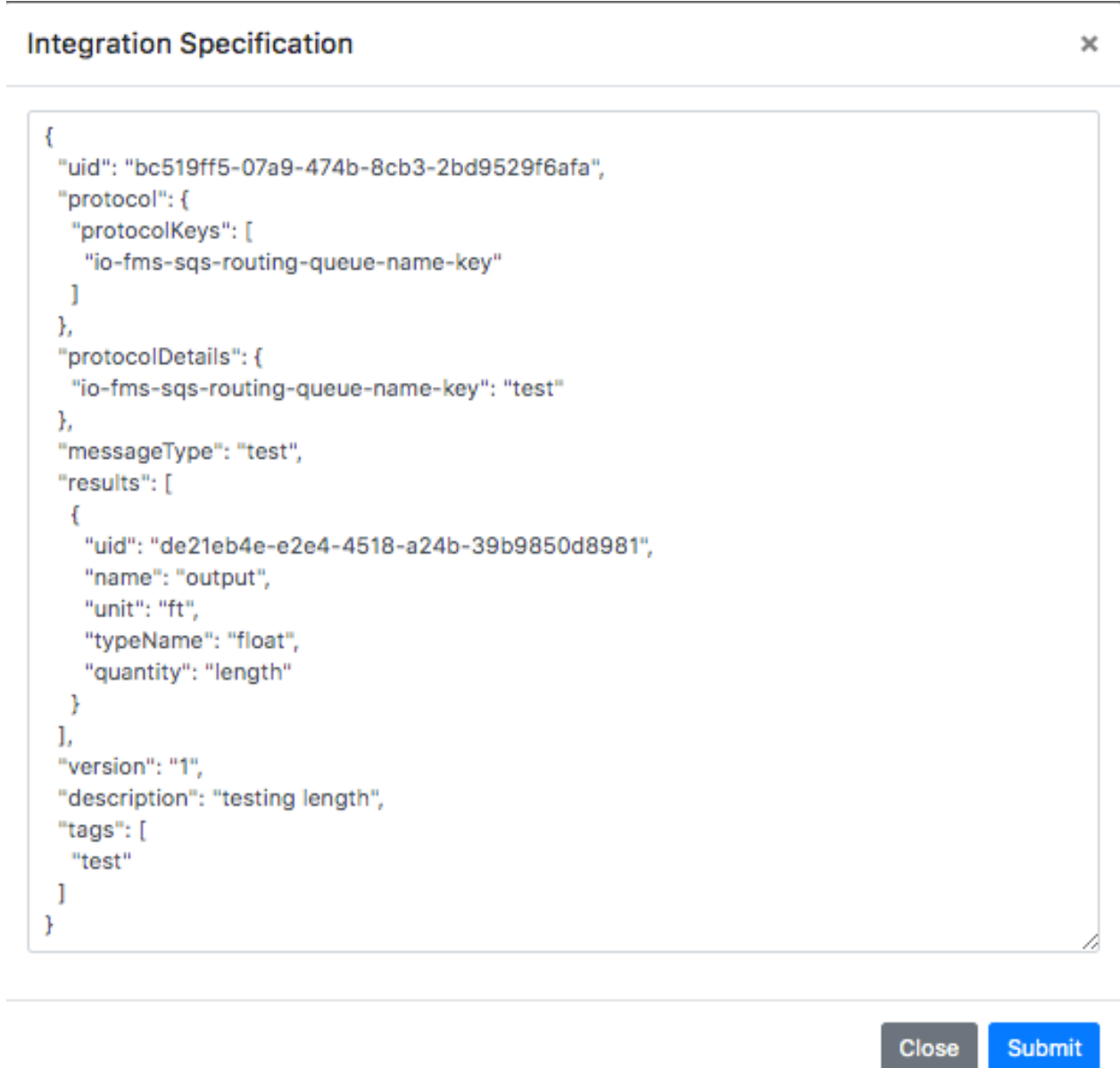


Figure 6.5: A generated ACI is shown in a confirmation dialog of the editor.

Visual programming is a paradigm in which the user creates programs in specific languages by visually assembling blocks representing the logic of the program rather than manually writing the code in a specific language. This has been used as a teaching tool to lower the learning curve for programming, specifically for younger audiences interested in

programming. There are a few examples of visual programming toolkits that provide a framework for building these programs, such as MIT’s Scratch (Resnick et al. 2009) and Google’s Blockly (Pasternak, Fenichel, and Marshall, 2017).

One specific use case of these tools is configuration (Pasternak et al., 2017). “Blockly is a library from Google for building beginner-friendly block-based programming languages” (Google Developers, 2019).

[Blockly](#) > [Demos](#) > Blockly Developer Tools

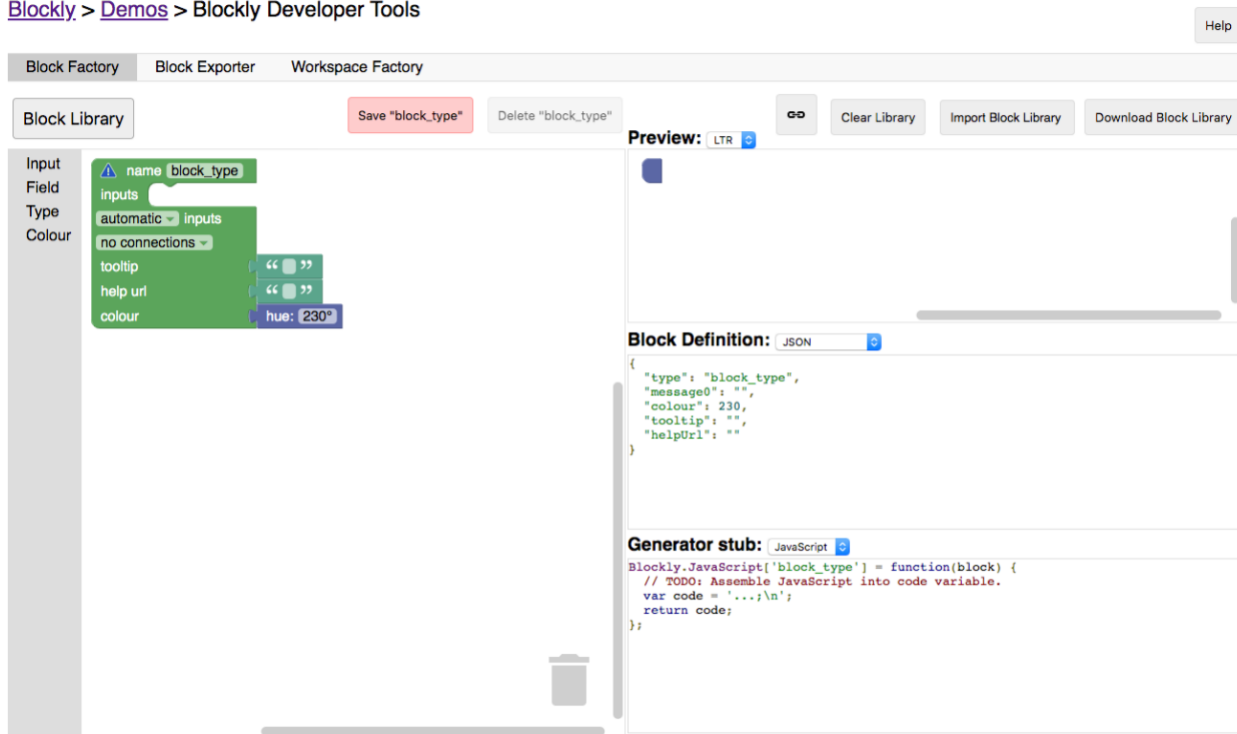


Figure 6.6: Block Factory demo from Googles Blockly

As a developer tool, Blockly provides a workspace for configuring and creating blocks that can be used in other Blockly applications and generating the code for them.¹ The tool has a “fixed” block with predefined inputs. Rather than the generated information being a program for

¹ <https://blockly-demo.appspot.com/static/demos/blockfactory/index.html>

execution, the generated code is the definition of a “Block” usable in other programs that support Blockly.

A similar application could further simplify the process of authoring an ACI. In a similar manner, a toolkit such as Blockly provides visual feedback and validation of data entry while also providing an intuitive “non-technical” interface for defining an ACI. The toolkit is highly extensible; to apply this concept to the creation of ACIs would require the following:

- Creating a web page to host the visual programming canvas.
- Defining blocks representing the configurable properties.
- Defining validation for the connectivity of the blocks.
- Defining a generator for converting blocks to the ACI format in JSON for submission to the registration service.

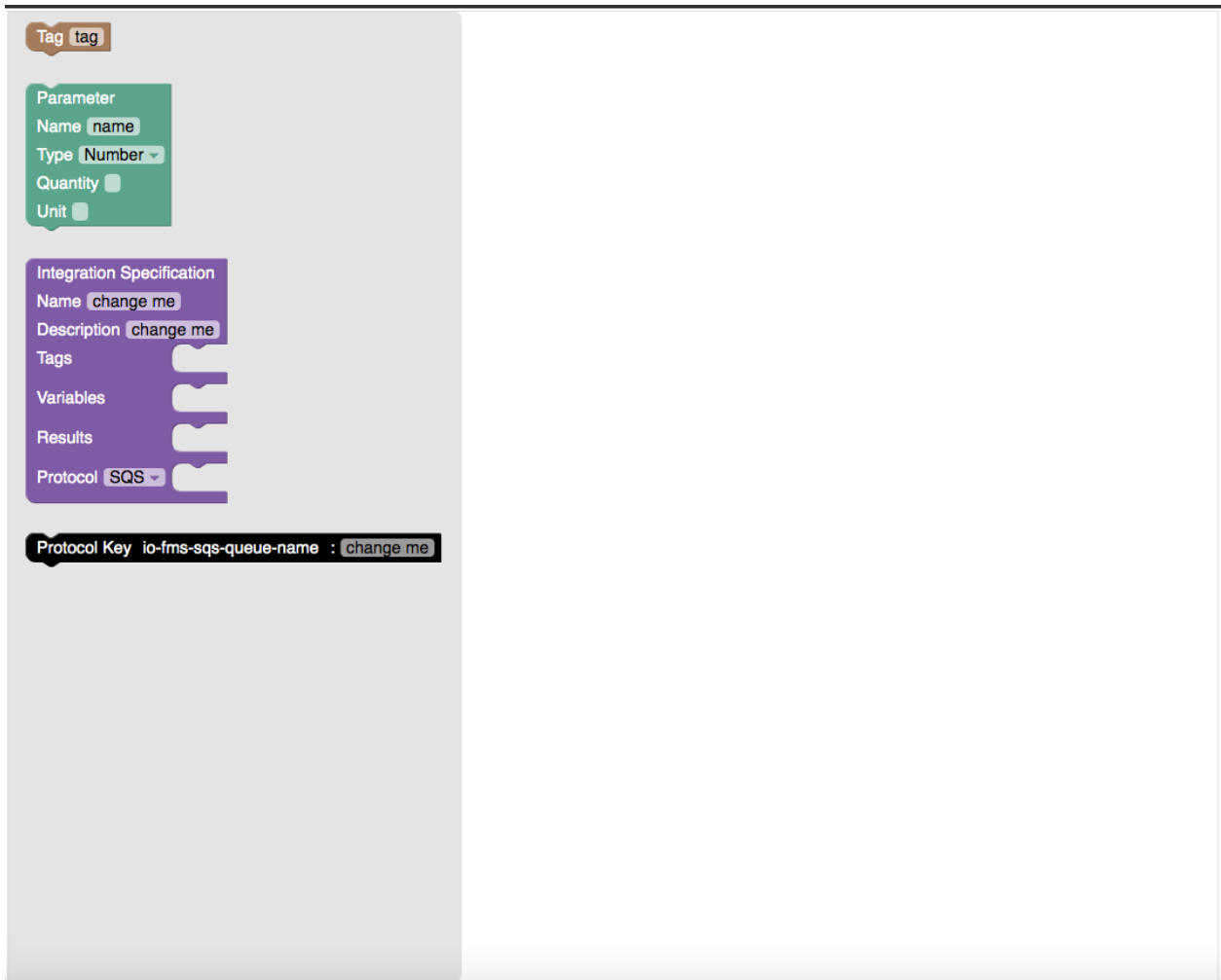


Figure 6.7: Blockly workspace for creating ACIs using visual programming.



Figure 6.8: Using Blockly to configure ACIs.

Figure 6.8 shows an ACI defined by placing blocks in the appropriate slots of a main “ACI” block. The input on these blocks is simple text entry and drop-down box selection. Generation of UUIDs for entries is not displayed on the blocks because that information is not necessarily relevant to the user. The system is only concerned with defining the ACI and therefore the UUIDs can be left to be generated when the blocks are converted to JSON.

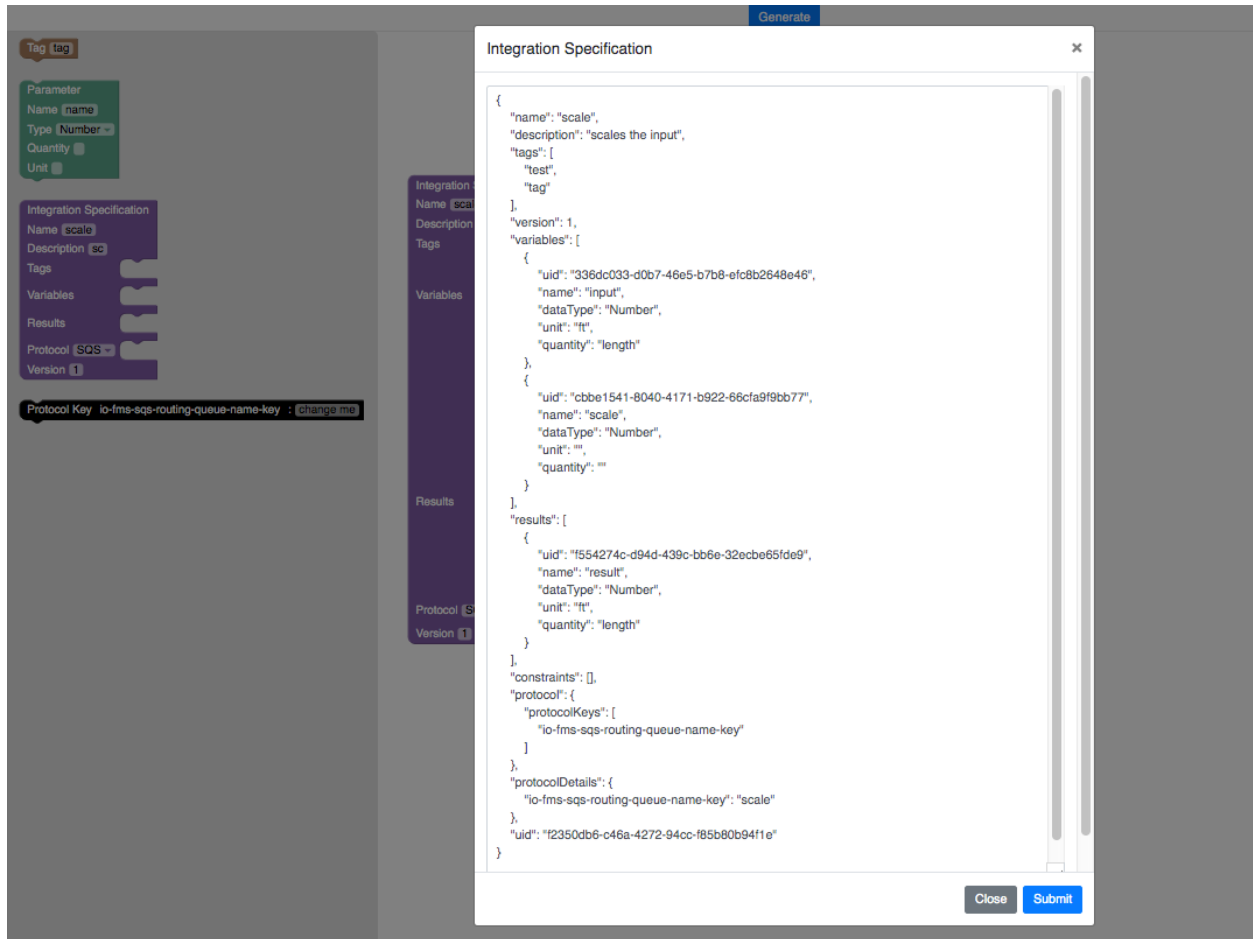


Figure 6.9: ACI generated for a model using a visual programming framework, Blockly.

6.5 Registering an ACI for a Web Enabled Model

With an ACI generated for a web-enabled model, the final step in contributing to the design environment is to register the web-enabled model with the design environment. As mentioned earlier in this work, the Registration Service is available for this. By submitting the generated ACI to the service, the defined protocol can be checked for the availability of the registered model. After successful validation, the specification status is changed to "SUBMITTED," and any remaining verification of the model and its interaction can be performed as deemed necessary by the maintainers of the design environment.

CHAPTER 7. AUTONOMOUS SYSTEM INTEGRATION USING ACI'S

7.1 Overview

Representing a real-world system by modeling and composing subsystems requires the user to design the orchestration of the execution of those subsystems and the exchange of the data constituting the results of the executions. Consequently, an application performing the orchestration of the execution should facilitate the execution of these subsystems as efficiently as possible while allowing the user of the application to express the intent of the design efficiently. The application should not require the user to create unfamiliar representations of the system model. The ACI and the enhancements of the FMS outlined in this work, provide the basis for such an application. As such, a prototype implementation of a web application was developed. The web application was written using modern HTML5 technologies for the components described. Once deployed, systems of varying complexity were designed and executed within the enhanced design environment leveraging the ACI and enterprise integration patterns incorporated in the messaging system of the FMS architecture to verify the effectiveness of the implementation.

7.2 Architectural Considerations for Subsystem Models as Microservices in a Cloud Environment

One advantage of representing subsystem models as microservices is that it provides the flexibility to design and develop the implementation of the model as necessitated by the developer. Developers creating new applications are easily afforded flexibility and freedom of design; however, to transition from traditional monolithic architectures to a microservices architecture requires careful consideration and planning (Balalaie, Heydarnoori, and Jamshidi, 2016). It is not just a matter of running the application on a server on the web or harvesting the algorithms from existing applications and putting them behind a REST API; the context of the

model usage needs to be understood. A careful analysis of the dependencies required to run the model in a stateless manner successfully should occur so that the transition to a microservice architecture ensures the execution of the model is purely independent of the existence of any external application or data source. Data, once accessible within the application running under the same process, needs to be available through separate microservices. Communication and information exchange between the model and the data is bound to the APIs of the services rather than the state of the implementation of the model. The microservice providing access to the data can express the form of the data in a fashion appropriate for external consumption, independent of the underlying storage mechanism. Similarly, if there are changes to the underlying data storage implementation (new choice of the storage provider, for example), external services that consume the data, are not affected by changes such as migrations of the internal data storage.

Another consideration is that of scale. For a traditional monolithic application restricted by the capacity of the resources available on the host server, the only means of handling more substantial data or more load is to scale vertically, i.e., spend money on more hardware (RAM, CPU, GPU) capable of handling the load; which is neither maintainable nor cost-effective considering the fluctuation in demand on an application. Architectures should scale horizontally, meaning, the application addresses the additional stress of more load by adding more instances of the microservices to accommodate the increase in demand. As demand decreases, microservices can be removed to control costs.

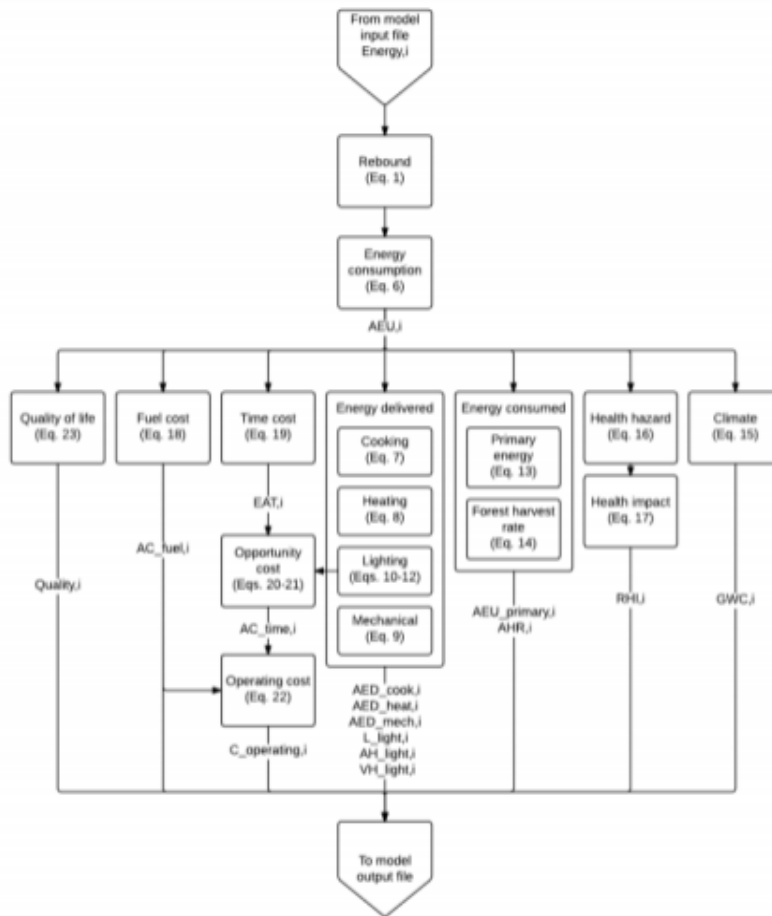


Figure 7.1: System model diagram of the energy needs subsystem of a third world village (MacCarty, 2015).

Consider the "energy need" system model diagram in Figure 7.1, where MacCarty (2015) developed this subsystem as part of a model used to evaluate the impact and practicality of replacing traditional third-world energy devices, such as three stone cookstoves, with more efficient devices that were deemed less harmful to the villagers and their environment. The complexity of the analysis is that while modern devices are more efficient than traditional energy sources, the increase in expense for fuel, the practicality of their reuse and the time spent learning to use the new devices could make their adoption less than ideal for the people actually using them in their everyday life (MacCarty, 2015). This design of the system model begins by

loading data into tables that represent the various energy services or tasks to run comparative analysis on the various devices and their related properties, such as efficiency, used to perform the applicable tasks, and the fuels used to operate the devices. Based on a user selection of tasks to compare, each subsystem model within the system model consumes an aggregation of valid combinations of tasks, devices, and fuels. Each model then calculates its output for each combination of task, device, and fuel to assemble into the resulting energy need table. This table is then passed on to the next system; the technology needs model. The migration of this system to a microservices architecture presents a couple of challenges. Firstly, from the description above, it can be seen that the design of each of the modeled subsystem implementations assumed the existence of a table that could be iterated over to acquire data specific to the task, device, and fuel. While not an incorrect assumption, in a microservices architecture, this is against the principle of scale as well as isolation (Newman, 2015). If for some reason, the tables were not accessible within the memory space of the system, all of the models would be unusable. Secondly, without the tables, the models are not reusable. Consider a system model used in a climate change study in which the designer would like to use the climate and health hazard models of the energy needs system (MacCarty, 2015). Their use in a microservice architecture without modifications would require that the new application also provide the model's access to the input tables. Similarly, the introduction of new fuels and or devices into the tables forces all models that accessed these tables, wherever accessed, to update to account for this change accordingly.

Lastly, the current implementation of the models assumes that the combination of tasks, devices, and fuels all fit within the address space of a single process. Although the assumption is sufficient for the analysis in the work of MacCarty (2015), consider the case of this model incorporated into a system model where data is collected from various villages around the world, and the number of tasks to be analyzed is increased by an order of magnitude. The current design of the models would suffer from CPU bottlenecks iterating over the tables.

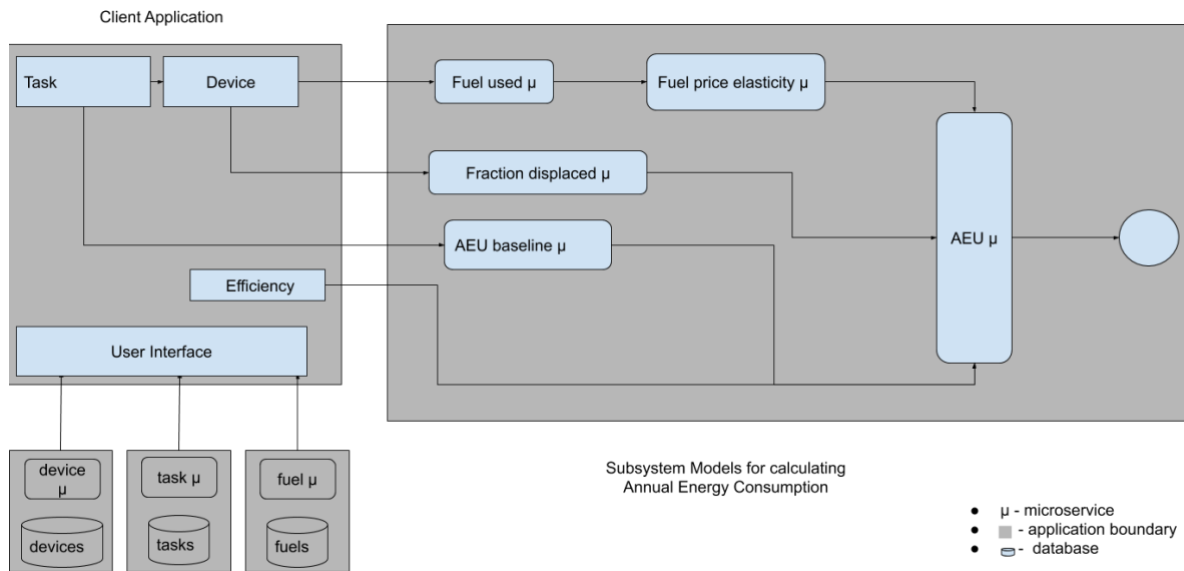


Figure 7.2: Microservices architecture for subsystem models calculating Annual Energy Consumption portion of the energy sub need system model for a single chosen task, device and fuel combination.

For this work, the energy needs subsystem was analyzed and redesigned using a microservices architecture. The main algorithms were the same except, instead of each model calculating their output for all the combinations of tasks, devices, and fuels, each model's input was changed to calculate a single combination of task, device, and fuel so that the entire system would scale based on the number of individual combinations. As part of this redesign, it became apparent that the tables are specific to the application, i.e., village energy, and as such were

extracted and provided from their microservice models to allow them to be accessed independently from the energy needs subsystem models.

As an example of rearchitecting the village energy models following microservices architecture principals, Figure 7.2 depicts a simplified application architecture in which there are three components: the client application which provides a user interface to drive the application; a set of microservices providing access to the data for the tables of devices, tasks, and fuel; and the design environment responsible for orchestrating the execution of the modeled subsystem. Figure 7.2 only shows a portion of the energy needs subsystem to highlight the separation of the data from the individual microservices. It also reveals that in this migration of the models to a distributed cloud architecture, new supporting microservices may need to be developed, such as the “fuel used” microservice.

The models in the original work of MacCarty (2015) defined implicit relationships with the tables encapsulated within each model. Each model previously had access to all the tables; calculating the “fuel used” based on the device type was simply a lookup of the device table cross-referenced with the fuel table.

Moving the selection of the task and the device out of the models to make them stateless means selection is handled outside of the models, and the result is passed into any downstream model requiring it as input. Creating a separate model that takes as its input a “device” and performing the lookup of the fuel used is a simple model that satisfies this requirement, providing a clean separation between the device selection and the fuel used. As of this work, the information available for the fuel used is simply a lookup; however, if a more sophisticated implementation is required, a model or combination of models that adhere to the input of “device” output of “fuel used,” can be substituted in place of the original “fuel used”

microservice. This new model would need to be registered as outlined in the previous chapter, and once validated, it would appear in the library of the design environment as an option for inclusion in the design.

7.3 High Level Architecture of a Decentralized Design Environment

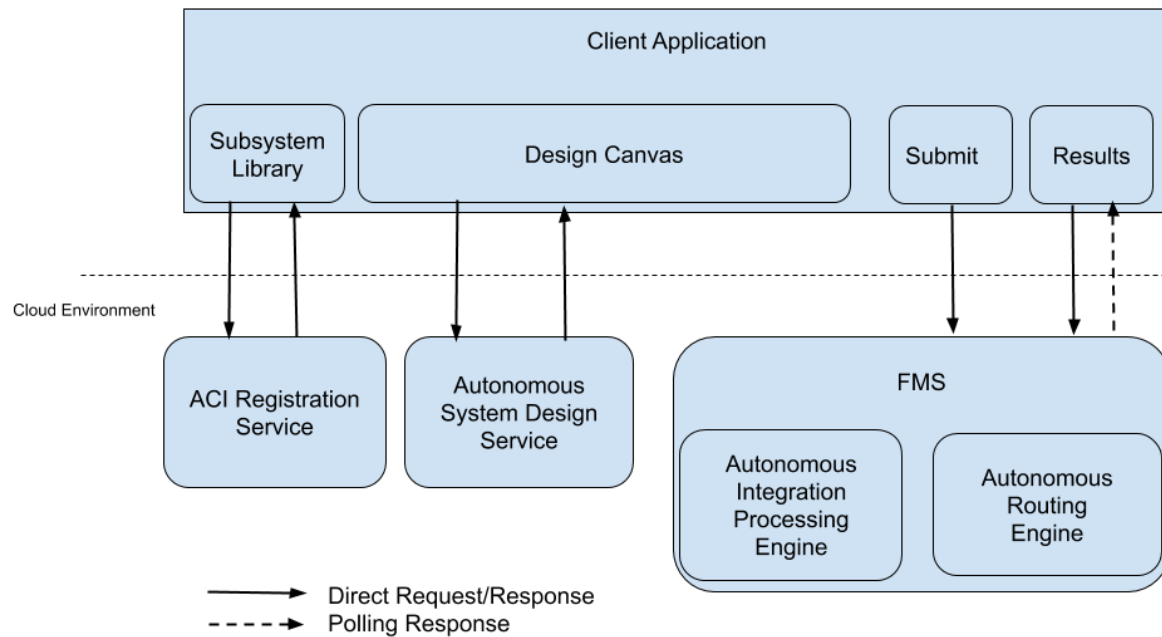


Figure 7.3: Architecture of a decentralized design environment, that leverages microservices defined in this research for autonomous system integration.

Given a set of ACIs representing subsystems available to use as components in a system model design, a distributed application can be developed to allow a user to design and execute the digital representation of the system. Although the application design and development were not the focus of this research, the implementation aids in demonstrating the practical contributions of the research.

Figure 7.3 depicts a high-level architecture of a prototypical system modeling and design environment. Due to the nature of the microservices developed in this research being web-enabled, there is flexibility in the architecture requirements of the client application, with the

only real restriction being proper access to the cloud hosting environment and services providing the functionality defined in this research. The high-level architecture depicts four main client components interacting with the services outlined in this work, either directly or indirectly, as follows:

- *Subsystem library*—A component that provides access to the available registered subsystem models and metadata information about them provided by their ACI by querying the ACI Registration Service. The subsystem representation of this component is left up to the developer of the client application, but the ACI has enough information available to give flexibility in these decisions.
- *Design Canvas*—A component that is used to design system models by composing subsystem models that are available as determined by the Subsystem library. The Design Canvas makes requests to the Autonomous System Design Service to create strong couplings between subsystems as dictated by their ACIs. Overall system validation can be incorporated into the Design Canvas as this information is available from the Autonomous System Design Service for any particular system model. The choices of canvas interactions and the representations of the system and subsystem models are left up to the developer of the client application, but the ACI and the Autonomous System Design Service provide enough information to give flexibility in these decisions.
- *Submission*—A component that is responsible for submitting system model designs to the FMS for execution. Due to the asynchronous design of the FMS (Suram et al. 2018), handling of the results is the responsibility of a different component. Here, the submission request will respond with a UUID to be used to look up the results when they are made available. Overall system validation can be incorporated into the Submission, as

this information is available from the Autonomous System Design Service for any particular system model.

- *Results Viewer*—A component that is used to visualize the results of a system model execution. Due to the asynchronous nature of the FMS (Suram et al., 2018), the client application should poll the FMS for a result of a submission using the UUID from the response of the corresponding submission request. The results will be in a form similar to that as defined by Suram et al. (2018) in that it will contain the workflow execution table (solver list) and the resulting parameters of the system execution.

The remainder of this chapter will describe an implementation of the components of a web application and how the work presented in this research, mainly the ACI and the enhanced messaging system of the FMS were leveraged to create an initial implementation of the cyber-physical system design environment inspired by the ideas of Bryden (2014).

7.4 Web Client Application

The screenshot shows a web application interface. At the top left, there is a 'Model Library' header with a circular icon. At the top right, there is an 'Inputs' header with a circular icon. The main content area is a light gray rectangle. In the center of this area is a white modal box. The modal box has the title 'Enter the URL of an ACI Repository'. Below the title is a text input field labeled 'ACI Repository URL' with a cursor inside. To the right of the input field is a blue button with the text 'GET MODELS' in white. At the bottom of the modal box, there is a small, faint 'CALCULATE' label.

Figure 7.4: Specifying the location of the repository of registered ACIs.

Usage of the application is initialized by visiting the hosting web site and entering the location of the registered subsystems that the user would like to use for designing a system. As of

this work, there is only one valid location and a single set of subsystems available within the environment, as depicted in Figure 7.4; however, it is conceivable that the interface presented here could provide the user a choice of “categories” that could be selected from to initialize the catalog of the design environment. These types of user experience considerations are left for future research. Upon entering the design environment, the user is presented with a visual representation of curated ACIs representing the subsystems available for designing cyber-physical systems. The ACIs are made available using the ACI Registration Service defined earlier in this work. As depicted in Figure 7.5, the library of subsystems is a simple grid-like display, in which the “name” field of each subsystem’s ACI is displayed along with a button that allows the user to see more of the detailed metadata as registered in the ACI. Information about each subsystem is available and accessible from within the application using the metadata information of the corresponding ACI.

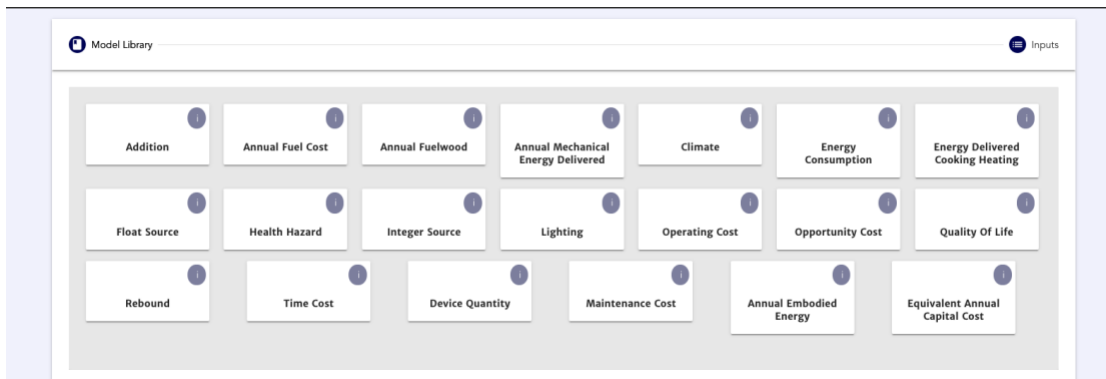


Figure 7.5: The library of available subsystems accessed by querying the ACI Repository Microservice.

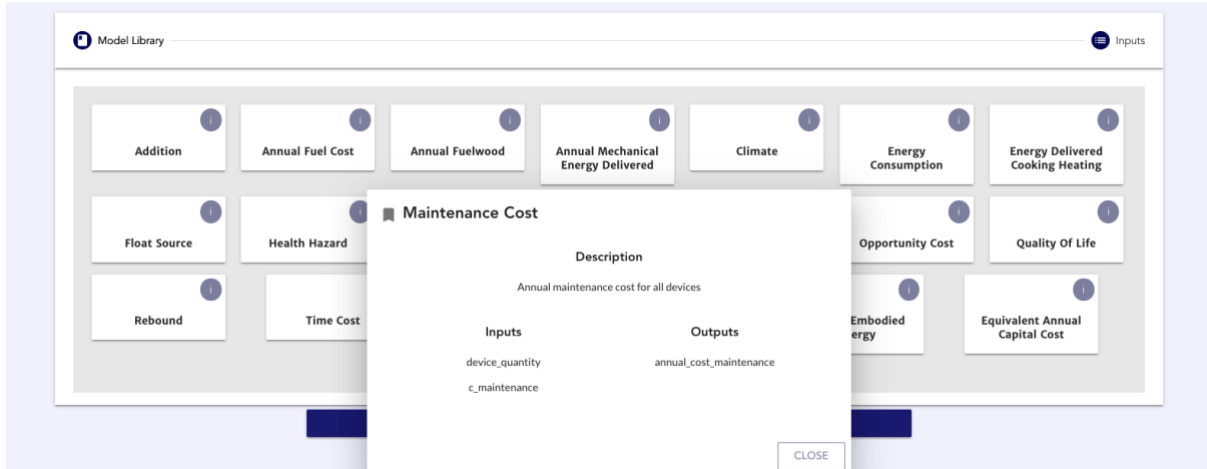


Figure 7.7: Information of subsystems available to the user for composing systems by querying the ACI Registration Microservice.

The most interactive component of the prototype application is the modeling/design canvas. The modeling/design canvas is the main work area of the application where the user assembles subsystems into an overall larger system model by defining connections between the models, “linking” them together. The application accesses the ACI and the supporting services defined earlier in this work to guide the user throughout the design process in several ways.

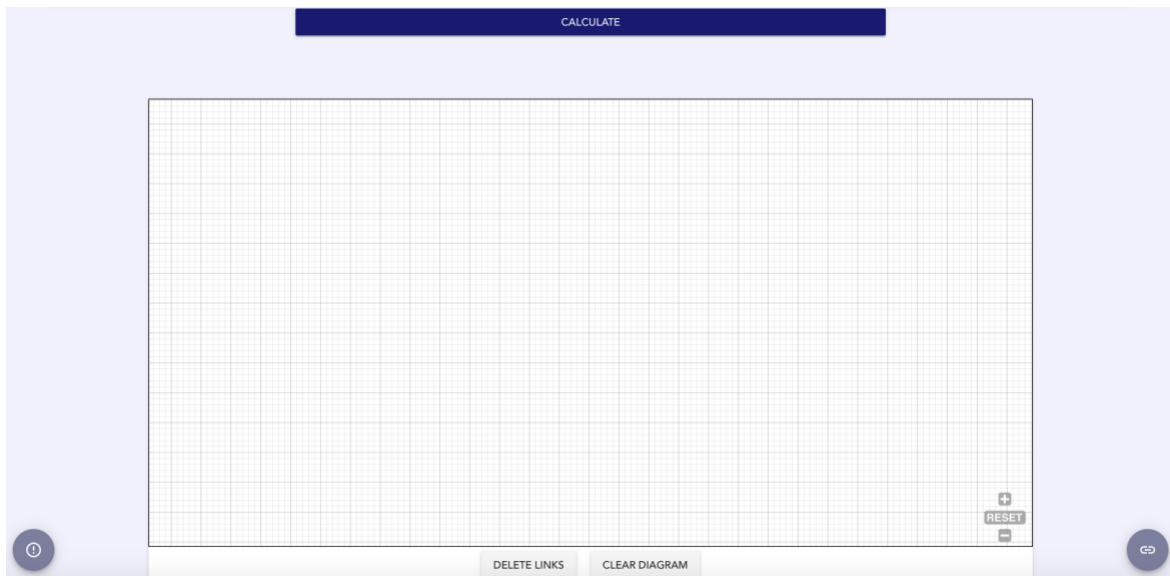


Figure 7.6: The system modeling/design canvas.

Firstly, the system can leverage the information provided by the ACIs of the subsystems to preventing incompatible linkages between models. A parameter validation algorithm can quickly evaluate the semantics of the parameters involved in the linking, providing immediate feedback on the validity of the connections. The microservice takes as input a pair of parameters and evaluates if they are compatible based on the “typeName” field defined and a comparison of the “unit” and “quantity” fields defined by the ACIs. The sophistication of the validation algorithm implementation is not the focus of this research but by incorporating an implementation of the algorithm within a microservice, the validation can easily be enhanced in its sophistication by updating the existing implementation or creating an entirely new microservice and deploying it alongside the existing implementation, updating the main application at the discretion of the developers of the application.

Secondly, in the event of an incompatible coupling, the environment can also leverage the parameter information declared in the ACIs of the subsystems to recommend a more suitable option for linking between subsystems based on models registered in the environment and the state of those models already added to the system. As with the validation service, the recommendation service developed in this work is a microservice that can easily be improved upon by providing a new implementation and updating the application to access it when ready.

Lastly, as discussed earlier, the integration and orchestration execution of these subcomponents of the model system is completely handled by the design environment. As a designer is iteratively exploring a system design, the “connectivity” changes of the subsystems correspond to changes in the integration of the underlying subsystem models. The Autonomous System Design Service will validate these changes by inspecting the ACIs of the subsystems, ensuring a viable execution graph of the subsystems can be derived. This graph can then be

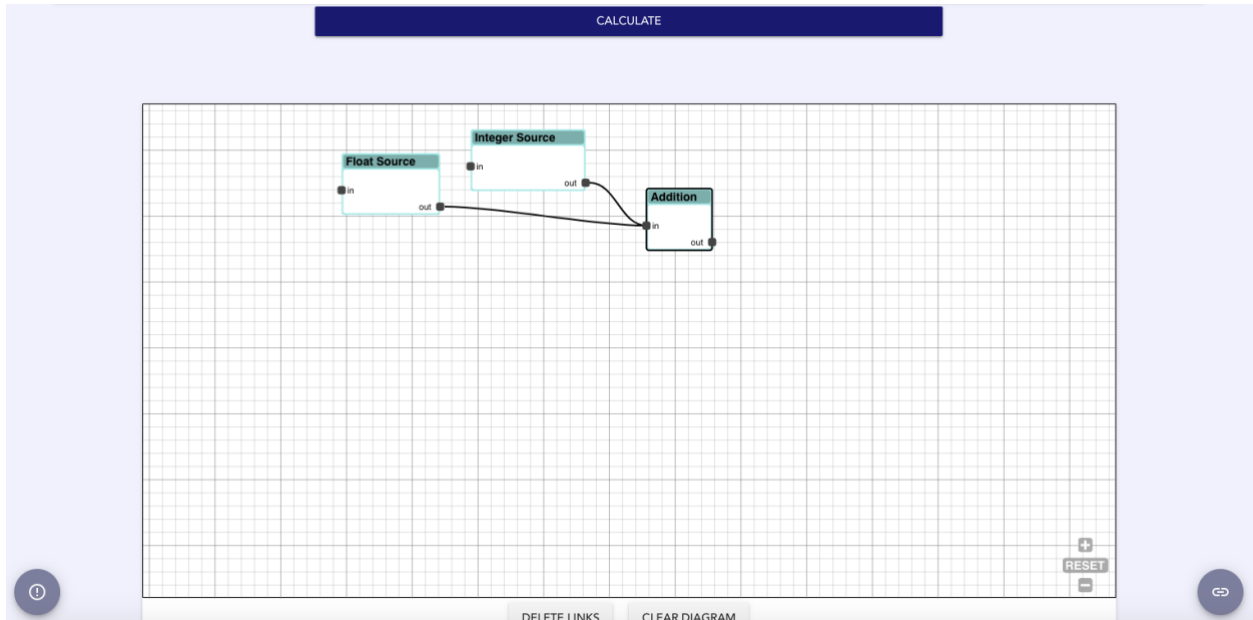


Figure 7.6: A system model composed of three models: a) Float Source a subsystem that generates a float value; b) Integer Source, a subsystem that generates an integer value; and c) Addition, a subsystem that computes the sum total of its inputs.

interpreted as a system integration of the involved subsystems. Combined with the incorporation of enterprise integration patterns to the messaging system of the FMS, the design environment can handle arbitrarily complex system designs without intervention from the user.

Figure 7.6 depicts a simple system modeled on the Design Canvas of the web client. The model is composed of a model that provides a random decimal number, a model that provides a random whole number and a model that adds its two inputs together.

7.5 Desktop Client Application

As stated earlier, interfacing with the design environment and the new services for autonomous system model design is not restricted to web applications. A desktop application with similar functionality can interact with the autonomous service and the FMS. The following images are of an analogous desktop client developed using the same services as the web client described in the previous section.

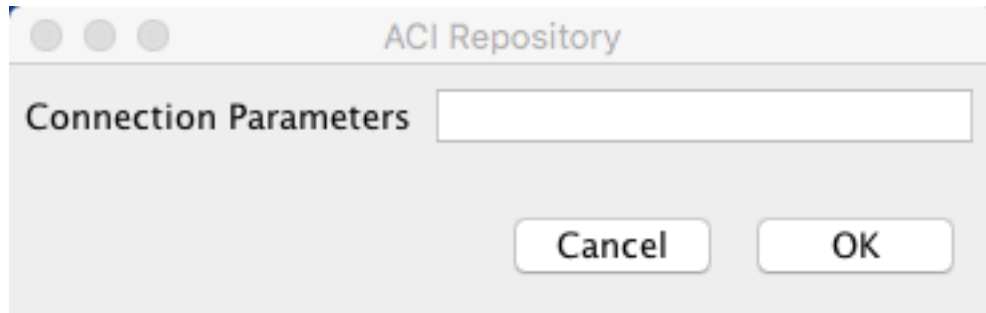


Figure 7.8: Desktop client application component allowing the user to specify the location of the ACI Repository.

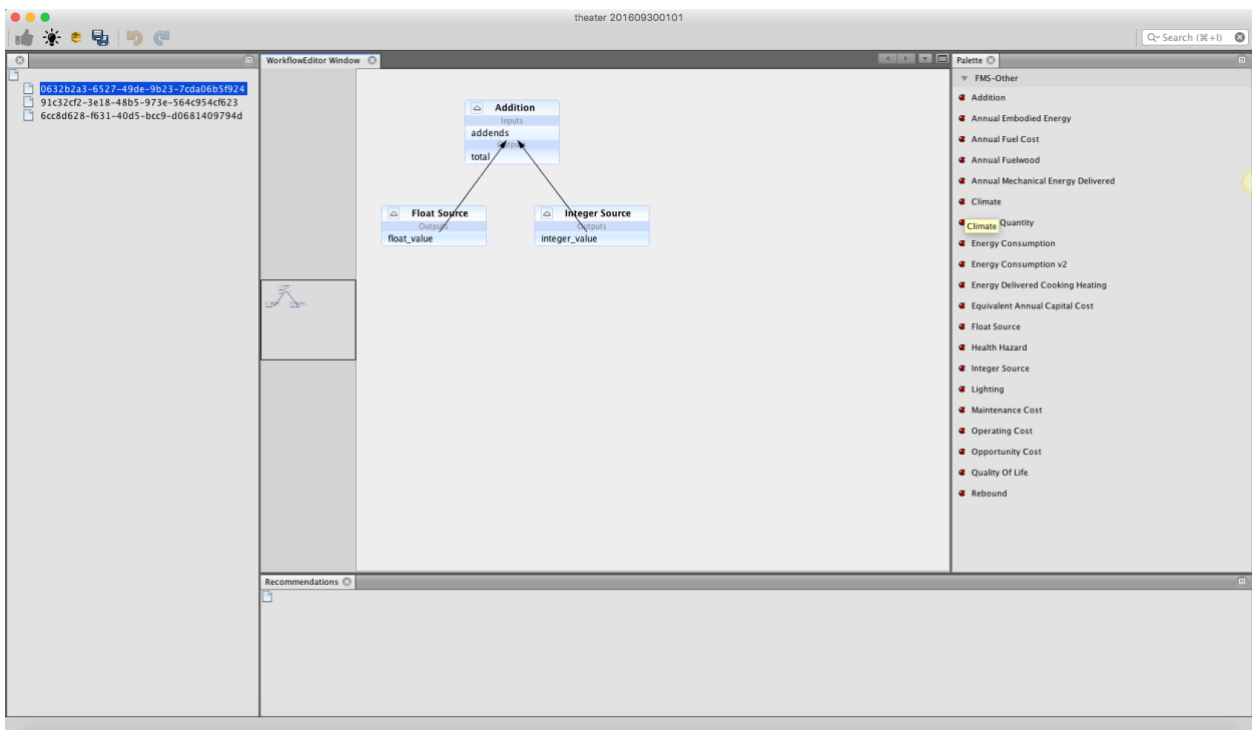


Figure 7.9: Desktop version of the modeling/design environment with the same “Addition” model as described in Figure 6.7.

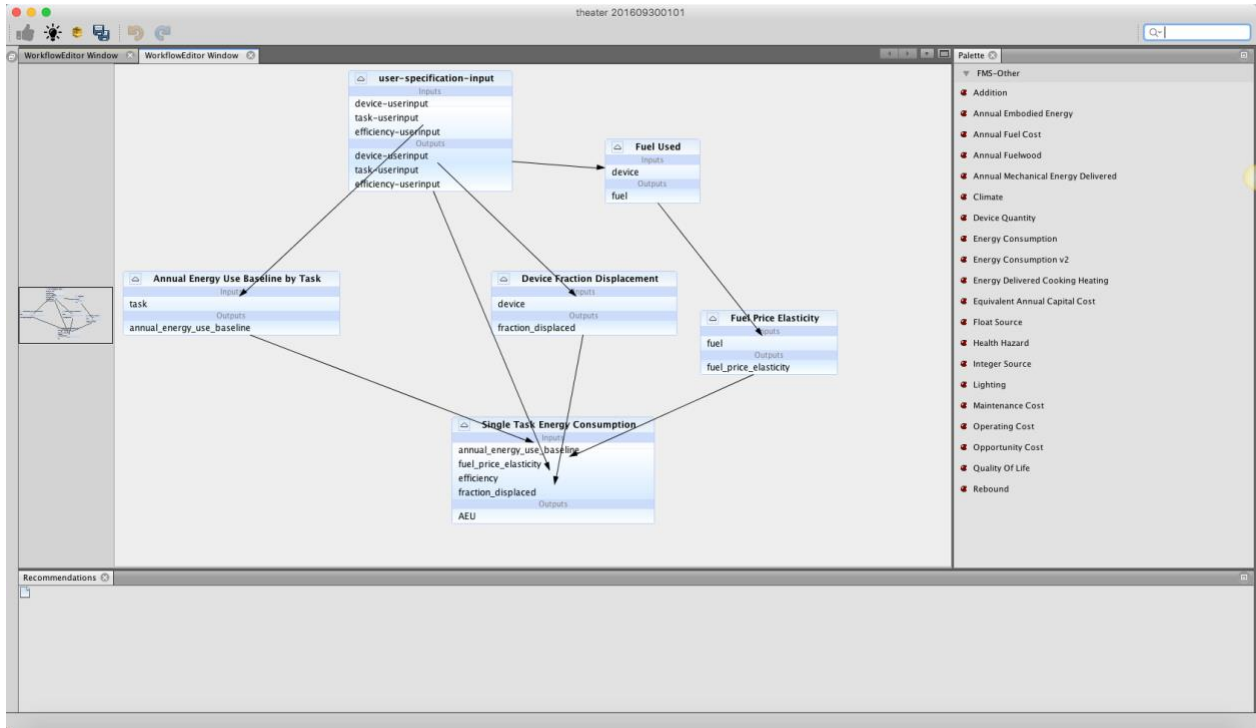


Figure 7.10: Annual energy consumption model of Figure 6.2, modeled and designed in the desktop client.

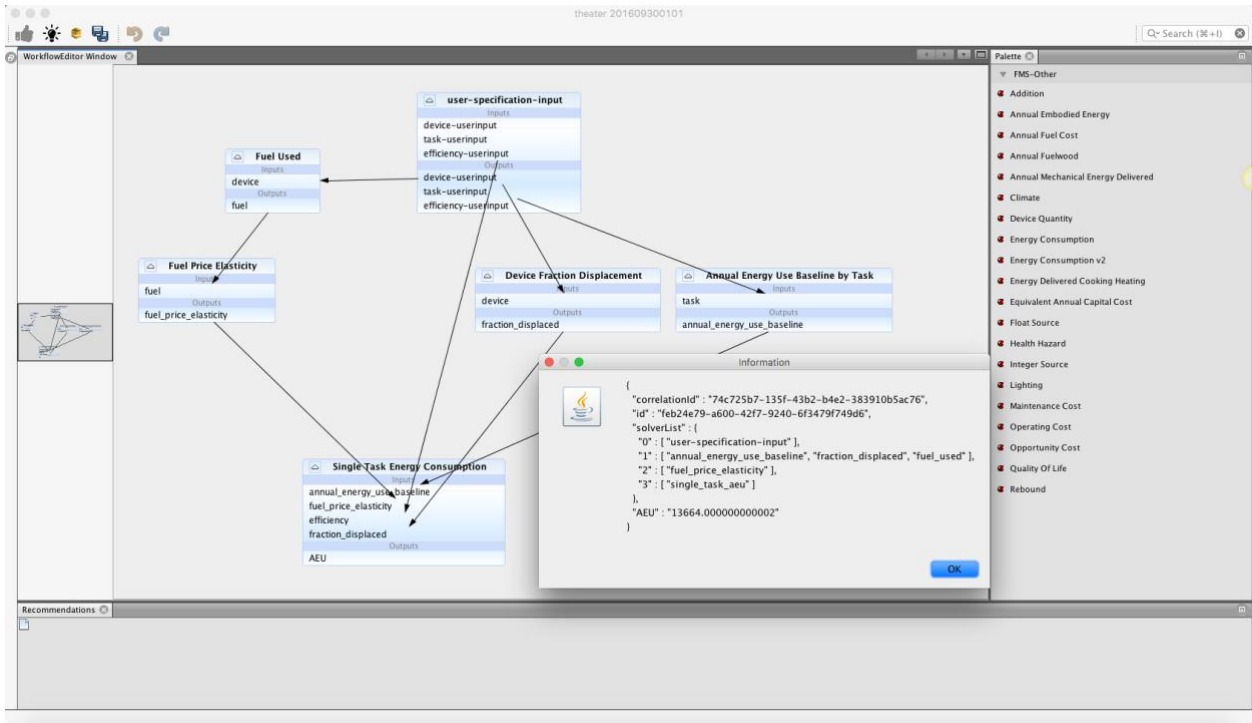


Figure 7.11: Results retrieved from within a desktop client application interfacing with the FMS and the autonomous system modeling/design services.

CHAPTER 8. CONCLUSIONS AND FUTURE WORK

8.1 Overview

This chapter summarizes the research presented in this dissertation and the impacts of the work. The concluding section is a discussion of future work to consider for further improvements and analysis.

8.2 Conclusion

The work presented in this dissertation is aimed at furthering the progress of modernizing the engineering modeling/design process. Prior research in this area demonstrated that the overall modeling/design process could benefit from leveraging cloud technologies but not without introducing new challenges, particularly in the area of system integration. This research details how system integration is a common concern in traditional distributed applications but presents a unique challenge to an engineering design application due to the exploratory nature of the design process. System integration needs to be dynamic in this case and handled by the application, i.e., autonomous, rather than the developer of the application. This research provides a solution by expanding on the work of Bryden (2014) and Suram et al. (2018) by developing the following components:

- Definition of an application coupling interface (ACI), an abstraction of the execution and protocols of distributed applications that provide the basis for automation of integration distributed systems. By creating ACIs for microservices representing individual disparate subsystems, the subsystems can be intuitively composed together to constitute a complex system model in a modern engineering design environment.
- Subsystem execution order can be derived using the information provided by the ACI and a specified connection between them to construct a “directed graph like” execution of the

subsystems: traversing the graph structure breadth first, adjacent nodes in the graph corresponding to a level in the graph and, achieving asynchronous execution of the subsystems.

- Incorporation of enterprise integration patterns that leverage the protocol and protocol details of the ACI to provide dynamic routing between distributed subsystem models and asynchronous execution paradigms necessary to orchestrate the execution of complex system models.
- Demonstration of authoring tools and contributing web-enabled models to a repository accessible to the design environment.
- Demonstration of iterative design and execution of a system composed of decentralized web-enabled subsystem models.

The schema of the ACI was developed independently of any particular design environment. In this work, it was shown to provide a means for the developer of the subsystem model to contribute their work in a manner that is composable with other subsystems within the modeling/design environment. The ACI also provides the intelligence about the subsystems to the environment of their composition and execution this is sufficient for automation of the validation and orchestration of the subsystems, decreasing the burden associated with system integration for a system modeler/designer.

8.3 Future Work

8.3.1 Self organization

One of the end goals in the vision of Bryden (2014) is the concept of self-organizing systems. The idea is that by collecting enough information about similar systems over time, a system could analyze the historical data and leverage it to accept and properly place a new component introduced into the system or reject it altogether without human intervention. While

an increase in the size and complexity of the digital representation of these cyber-physical systems ideally provides a higher quality of analysis and insight into the physical systems, the cost of maintenance and improving the system increases, so much so that the need for automatically organizing, editing, and assembling subcomponents into the system is required. The ACI already provides a minimal level of automation at the system integration level, but more investigation should be done to determine whether an ACI can be leveraged to facilitate some aspect of the self-organization of systems.

8.3.2 Breadth of systems model domains

The engineering modeling/design environment presented by Suram et al. (2018), allowed defining a system in which derived workflow execution could be run iteratively by defining a criterion for completion (Suram, 2016). While this work did not explicitly address this, the changes developed in the messaging system of the environment do not preclude the ability to execute iteratively. Further work could include the investigation of additional integration patterns and possible modifications of the completeness condition of the aggregator pattern to incorporate the concept of “iterations” within the levels of execution as well as for the entire workflow. Similarly, the work defined in this research does not exclude systems models that vary in the temporal or spatial domain; however, this research did not investigate systems with these requirements.

8.3.3 Additional protocols for endpoints and routing

In this work, a single web protocol, queue-based messaging, was implemented for extension of the environment; however, the definition of the ACI and its protocol details are flexible enough to accommodate other protocols. Chapter 3 described the preliminary steps for adding support for other protocols to the messaging system presented in this work, and further investigation should be done to provide implementations supporting popular microservice

protocols such as Representational State Transfer (REST). Once the protocol is supported, investigation in providing supporting tools such as model templates that ease cloud resource provisioning and integration to encourage contribution to the environment further.

8.3.4 Constraint verification

In this work, definitions for constraints on parameters of inputs to subsystems were outlined; however, implementations enforcing these constraints were minimally investigated, only enough to validate that the ACI definition accommodates the specification of constraints. Further investigation of constraint evaluation at runtime and incorporation as part of the validation during the registration process is needed.

8.3.5 Human computer interaction

In this work, the implementation of a microservice to simplify building user representations of systems has been developed. However, the intent was limited to the scope of this research. Leveraging the information of the ACI lessens the interactions and requirements of the user; however, the investigation of intuitive tools and the associated interactions for a user of a modern engineering modeling/design environment should be done. In traditional engineering modeling/design environment user interfaces, the designer interacts with a palette of models of which they can drag and drop onto a canvas, which constitutes the subsystems composing the model. Once on the canvas, the interaction is typically that of “linking” to define connectivity or data exchange between subsystems. This work implemented this type of interface; however, research in user interaction of engineering modeling/design applications should be performed. The information provided by the ACI could be leveraged to develop tools that aid in providing a more intuitive user experience, such as a recommender system.

8.3.6 Registration process

In this work, while the microservices were all developed as separate artifacts, these artifacts were deployed in a cloud hosting environment that allowed for unrestricted communication with the messaging system of the engineering design environment. Orchestration of the deployment and provisioning of the modeled subsystems as microservices was not the focus of this research and remains for future investigation. An ACI is akin to a “contract,” which may lead to interesting consumer-driven testing paradigms integrated into the submission process and validation, yielding automation to validate the execution of the protocols and protocol details upon submission of a new ACI.

8.3.7 Transient workflow execution information

In this work, this system did not persist data calculated during the construction and execution of the workflow, such as the directed graph, the “level to subsystem relational table” described in Chapter 4, or any intermediate subsystem execution values. The system sustained intermediate results during the life of the execution in a cache. However, in the context of spatial and temporal varying domains, this will be more critical. Ideally, these could be stored in the namespace, as described by Suram et al. (2018) for future reference and troubleshooting purposes.

8.3.8 Parameters

One of the key aspects of the ACI and its ability to provide autonomous linking of subsystems lies in the definition of its parameters. During this research, where the development and creation of ACIs for subsystems were relatively controlled, there was a high likelihood that parameters that were intended to be coupled would be successfully coupled; however, in an environment with less regulation and restrictions, values for parameter quantities, types, unit, etc. can be ambiguous. This work referred to enforcing parameters and constraint definitions via the

specification validation and editor tools; however, there are aspects of this that weren't investigated in this work. An example is that there isn't a formal system of valid values for "typeNames." In this work, "typeNames" were restricted to a known set; however, this may not be applicable or may be too flexible to reliably allow the system to determine if subsystems can be linked.

REFERENCES

1. “curl://.” Accessed 1 Oct. 2019. <https://curl.haxx.se/>.
2. “Introducing JSON.” Accessed 3 Sep. 2019. <https://www.json.org/>.
3. “Introduction–Resilience4j.” Accessed 30 Oct. 2019. <https://resilience4j.readme.io/docs>.
4. “Open Source Initiative.” Accessed 3 Sep. 2019. <https://opensource.org/>.
5. Alam, Kazi M., and Abdulmotaleb El Saddik. 2017. “C2PS: A Digital Twin Architecture Reference Model for the Cloud-Based Cyber-Physical Systems.” *IEEE Access* 5: 2050–62.
6. Apache Software Foundation. “Apache Camel”. Accessed 2 Oct. 2019. <https://camel.apache.org/>.
7. App Developer Magazine. “Record growth in microservices.” Posted 2 May. 2018. Accessed 8 Sep. 2019. <https://appdevelopermagazine.com/record-growth-in-microservices/>.
8. App Store on the App Store. Accessed 15 Sep. 2019. <https://apps.apple.com/us/app/apple-store/id375380948>.
9. AWS. “Amazon Elastic Container Registry” (ECR). Accessed 1 Oct. 2019. <https://aws.amazon.com/ecr/>.
10. AWS. “Amazon Elastic Container Service” (ECS). Accessed 20 Oct. 2019. <https://aws.amazon.com/ecs/>.
11. AWS. “Amazon Simple Queue Service” (SQS). Accessed 14 Sep. 2019. <https://aws.amazon.com/sqs/>.
12. AWS. “Amazon Simple Storage Service” (S3). Accessed 15 Sep. 2019. <https://aws.amazon.com/s3/>.
13. AWS. “Amazon SQS Dead-Letter Queues.” Accessed 15 Sep. 2019. <https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-dead-letter-queues.html>.
14. AWS. “AWS Identity and Access Management” (IAM). Accessed 17 Sep. 2019. <https://aws.amazon.com/iam/>.
15. AWS. “Message Queues.” Accessed 3 Sep. 2019. <https://aws.amazon.com/message-queue/>.

16. Balalaie, Armine, Abbas Heydarnoori, and Pooyan Jamshidi. 2016. “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture.” *IEEE Software* 33 (3): 42–52.
17. Balasubramanian, Krishnakumar, Douglas C. Schmidt, Zoltán Molnár, and Ákos Lédeczi. 2009. “System Integration Using Model-Driven Engineering.” In *Designing Software-Intensive Systems: Methods and Principles*, edited by Pierre F. Tiako, 474–504. IGI Global.
18. Bang-Jensen, Jørgen, and Gregory Z. Gutin. 2008. *Digraphs: Theory, Algorithms and Applications*. Springer Science & Business Media.
19. Bryden, Kenneth M. 2014. “A Proposed Approach to the Development of Federated Model Sets.” In *International Congress on Environmental Modelling and Software*. scholarsarchive.byu.edu. <https://scholarsarchive.byu.edu/iemssconference/2014/Stream-B/5/>.
20. Core definitions and terminology – JSON Schema. Accessed 14 Sep. 2019. <http://json-schema.org/draft-04/json-schema-core.html>.
21. Csikszentmihalyi, Mihaly, Sami Abuhamdeh, and Jeanne Nakamura. 2014. Flow. In *Flow and the Foundations of Positive Psychology: The Collected Works of Mihaly Csikszentmihalyi*, edited by Mihaly Csikszentmihalyi, 227–38. Dordrecht: Springer Netherlands.
22. Dastjerdi, A. V., and R. Buyya. 2016. “Fog Computing: Helping the Internet of Things Realize Its Potential.” *Computer* 49 (8): 112–16.
23. del Valle, Ignaciao. “brutusin/json-forms.” GitHub. Accessed 10 Nov. 2019. <https://github.com/brutusin/json-forms>.
24. Domo. “Data Integration: Connect everything, everywhere.” Accessed 8 Sep. 2019. <https://www.domo.com/platform/connect>.
25. El Saddik, A. 2018. “Digital Twins: The Convergence of Multimedia Technologies.” *IEEE Multimedia* 25 (2): 87–92.
26. Fielding, Roy T., and Richard N. Taylor. 2000. “Architectural Styles and the Design of Network-based Software Architectures.” (PhD diss., University of California, Irvine).
27. Fowler, Martin. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.
28. FreeCAD. <https://www.freecadweb.org/>. Accessed 7 Sep. 2019.
29. Gilbert, Seth, and Nancy Lynch. 2002. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services.” *ACM SIGACT News* 33 (2): 51–59.
30. Git. <https://git-scm.com/>. Accessed 1 Oct. 2019.

31. GitHub. <https://github.com/>. Accessed 15 Sep. 2019.
32. GitLab Docs. “Introduction to CI/CD with GitLab.” Accessed 1 Oct. 2019. <https://docs.gitlab.com/ee/ci/introduction/>.
33. GitLab. <https://gitlab.com/>. Accessed 1 Oct. 2019.
34. Google Cloud. “What are Containers and their benefits.” Accessed 1 Oct. 2019. <https://cloud.google.com/containers/>.
35. Google Developers. “Blockly”. Accessed 20 Oct. 2019. <https://developers.google.com/blockly>.
36. Groundan, Appu and Qingyan Chen. “Introducing Jib — build Java Docker images better.” | Google Cloud Platform. Posted 9 Jul. 2018. Accessed 1 Oct. 2019. <https://cloud.google.com/blog/products/gcp/introducing-jib-build-java-docker-images-better>.
37. Hohpe, Gregor and Bobby Woolf. “Enterprise Integration Patterns – Aggregator”. Accessed 3 Oct. 2019. <https://www.enterpriseintegrationpatterns.com/patterns/messaging/Aggregator.html>.
38. Hohpe, Gregor and Bobby Woolf. “Enterprise Integration Patterns – Splitter”. Accessed 3 Oct. 2019. <https://www.enterpriseintegrationpatterns.com/patterns/messaging/Sequencer.html>.
39. Hohpe, Gregor and Bobby Woolf. “Enterprise Integration Patterns–Message Router.” Accessed 3 Oct. 2019. <https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageRouter.html>.
40. Hohpe, Gregor and Bobby Woolf. “Enterprise Integration Patterns–Pipes and Filters.” Accessed 4 Oct. 2019. <https://www.enterpriseintegrationpatterns.com/patterns/messaging/PipesAndFilters.html>.
41. Hohpe, Gregor and Bobby Woolf. “Enterprise Integration Patterns”. Accessed 3 Oct. 2019. <https://www.enterpriseintegrationpatterns.com/>.
42. Hohpe, Gregor, and Bobby Woolf. 2004. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional.
43. IDC. “Industrial Customers Are Ready for Cloud – Now”: an IDC Info Brief sponsored by AWS.” Accessed 30 Oct. 2019. https://d1.awsstatic.com/analyst-reports/AWS%20infobrief_final.pdf?trk=ar_card.
44. Jaramillo, David, Duy Van Nguyen, and Robert Smart. 2016. “Leveraging Microservices Architecture by Using Docker Technology.” *SoutheastCon 2016*, 1–5.

45. JSON Schema. “Implementations.” Accessed 15 Sep. 2019. <https://json-schema.org/implementations.html#web-ui-generation>.
46. JSON Schema. “Specification Links.” Accessed 14 Sep. 2019. <http://json-schema.org/specification-links.html>.
47. Kanitkar, Tejal, Rangan Banerjee, and T. Jayaraman. 2019. “An Integrated Modeling Framework for Energy Economy and Emissions Modeling: A Case for India.” *Energy* 167 (January): 670–79.
48. Klimaschewski, Udo. “EvalEx.” Accessed 16 Sep. 2019. <https://github.com/uklimaschewski/EvalEx>.
49. Koulamas, Christos, and Athanasios Kalogeras. 2018. “Cyber-Physical Systems and Digital Twins in the Industrial Internet of Things [Cyber-Physical Systems].” *Computer* 51 (11): 95–98.
50. Kreps, Jay, Neha Narkhede, Jun Rao, and Others. 2011. “Kafka: A Distributed Messaging System for Log Processing.” In *Proceedings of the NetDB*.
51. Kubernetes. “Production – Grade Container Orchestration.” Accessed 1 Oct. 2019. <https://kubernetes.io/>.
52. Lawrence Livermore National Laboratory. “LLNL Software Portal.” Accessed 8 Sep. 2019. <https://software.llnl.gov/>.
53. Lewis, James and Martin Fowler. “Microservices.” Accessed 3 Sep. 2019. <https://martinfowler.com/articles/microservices.html>.
54. Lloyd, W., O. David, J. C. Ascough, K. W. Rojas, J. R. Carlson, G. H. Leavesley, P. Krause, T. R. Green, and L. R. Ahuja. 2011. “Environmental Modeling Framework Invasiveness: Analysis and Implications.” *Environmental Modelling & Software* 26 (10): 1240–50.
55. MacCarty, Nordica Ann. 2015. “Development and Use of an Integrated Systems Model to Design Technology Strategies for Energy Services in Rural Developing Communities.” (PhD Diss., Iowa State University)
56. Mauro, Tony. “Adopting Microservices at Netflix: Lessons for Architectural Design” Nginx. Posted 19 Feb. 2015. Accessed 8 Sep. 2019. <https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/>.
57. Maven. “Plugin Developers Centre.” Accessed 1 Oct. 2019. <https://maven.apache.org/plugin-developers/>.
58. Maven. “Welcome to Apache Maven.” Accessed 1 Oct. 2019. <https://maven.apache.org/>.

59. MDN web docs Mozilla. "Arithmetic operators." Accessed 17 Sep. 2019, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Arithmetic_Operators#Addition.
60. MDN web docs Mozilla. "HTTP request methods." Accessed 14 Sep. 2019. <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>.
61. Microsoft Azure. "Intelligent edge innovation across data, IoT, and mixed reality." Posted 2 May. 2019. Accessed 29 Oct. 2019. <https://azure.microsoft.com/en-us/blog/intelligent-edge-innovation-across-data-iot-and-mixed-reality/>
62. Microsoft. "Visual Studio Code – Contribution Points." Accessed 16 Sep. 2019. <https://code.visualstudio.com/api/references/contribution-points>.
63. MongoDB. "The database for modern applications." Accessed 16 Sep. 2019. <https://www.mongodb.com/>.
64. Murdock, Paul, Louay Bassbous, Martin Bauer, Mahdi Ben Alaya, Rajdeep Bhowmik, Patricia Brett, Rabi Ndra Chakraborty, et al. 2016. "Semantic Interoperability for the Web of Things."
65. Muth, David. J., and Kenneth M. Bryden. 2013. "An Integrated Model for Assessment of Sustainable Agricultural Residue Removal Limits for Bioenergy Systems." *Environmental Modelling & Software* 39 (January): 50–69.
66. NetBeans Platform. "File Type Integration Tutorial." Accessed 15 Sep. 2019. <https://platform.netbeans.org/tutorials/nbm-filetype.html>.
67. Newman, Sam. 2015. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media.
68. Nilsson, J., and F. Sandin. 2018. "Semantic Interoperability in Industry 4.0: Survey of Recent Developments and Outlook." In *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, 127–32. ieeexplore.ieee.org.
69. NREL. "NWTC Information Portal." Accessed 8 Sep. 2019. <https://nwtc.nrel.gov/Software>.
70. NT Graham, EGR Davies, MI Hejazi, et al., (2018) "Water sector assumptions for the shared socioeconomic pathways in an integrated modeling framework," *Water Resources Research*, pp 6423-6440.
71. OnShape. "The State of Product Development & Hardware Design 2019." Accessed 30 Oct. 2019. <https://www.onshape.com/resources/ebooks/state-of-product-development-hardware-design-2019>.
72. Oracle. "Java + You, Download Today!" Accessed 17 Sep. 2019. <https://www.java.com/>.

73. OSF. <https://osf.io/>. Accessed 7 Sep. 2019.
74. Padula, S.L., Gillian, R.E., 2006. Multidisciplinary environments: a history of engineering framework development. In: Proceedings of the 11th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference. AIAA Report No. 2006- 7083. Portsmouth, VA.
75. Pasternak, Erik, Rachel. Fenichel, and Andrew N. Marshall. 2017. “Tips for Creating a Block Language with Blockly” In *2017 IEEE Blocks and Beyond Workshop*, 21–24. ieeexplore.ieee.org.
76. POETS. “Our Research: Overview.” Accessed 8 Sep. 2019. https://poets-erc.org/our-research/overview_.
77. RabbitMQ. “Understanding RabbitMQ.” Accessed 14 Sep. 2019. <https://www.rabbitmq.com/>.
78. Resnick, Mitchel, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, et al. 2009. “Scratch: Programming for All.” *Communications of the ACM* 52 (11): 60–67.
79. Rizzoli, A. E., G. Leavesley, I. I. J. C. Ascough, R. M. Argent, I. N. Athanasiadis, V. Brillhante, F. H. A. Claeys, et al. 2008. “Integrated Modelling Frameworks for Environmental Assessment and Decision Support,” USGS Staff -- Published Research, <https://digitalcommons.unl.edu/usgsstaffpub/196/>.
80. Rossum, G. van. 1995. “Python Tutorial”, Technical Report CS-R9526.” *Centrum Voor Wiskunde En Informatica (CWI), Amsterdam*.
81. Roy, Audry. “CookieCutter: Better Project Templates.” Accessed 1 Oct. 2019. <https://cookiecutter.readthedocs.io/en/latest/>.
82. Sage. “SageMath.” Accessed 7 Sep. 2019. <http://www.sagemath.org/>.
83. SimScale. “Simulation software reinvented for the web.” Accessed 8 Sep. 2019. <https://www.simscale.com/>.
84. Soanes, Catherine, and Angus Stevenson. 2004. *Concise Oxford English Dictionary*. Vol. 11. Oxford University Press Oxford.
85. Sprint by Pivotal. “Spring Boot.” Accessed 17 Sep. 2019. <https://spring.io/projects/spring-boot>.
86. Sprint by Pivotal. “Spring Data REST.” Accessed 17 Sep. 2019. <https://spring.io/projects/spring-data-rest>.
87. Sprint by Pivotal. “Spring Framework.” Accessed 2 Oct. 2019. <https://spring.io/>.

88. Sprint by Pivotal. “Spring Integration.” Accessed 2 Oct. 2019. <https://spring.io/projects/spring-integration>.
89. Suram, Sunil, Nordica A. MacCarty, and Kenneth M. Bryden. 2018a. Engineering Design Analysis Utilizing a Cloud Platform. *Advances in Engineering Software* 115 (January): 374–85.
90. Suram, Sunil. 2016. “Strategies for including Cloud-computing into an Engineering Modeling Workflow” (PhD Diss., Iowa State University).
91. Swagger. “OpenAPI Specification” Sponsored by Brown Bear. Accessed 10 Sep. 2019. <https://swagger.io/specification/>.
92. Texas Advanced Computing Center (TAAC). “Software packages and libraries on tacc systems.” Accessed 8 Sep. 2019. <https://www.tacc.utexas.edu/systems/software>.
93. The OpenFOAM Foundation. “OpenFOAM.” Accessed 7 Sep. 2019. <https://openfoam.org/>.
94. The OpenScience Project. Posted 22 Mar. 2017. Accessed 7 Sep. 2019. <http://openscience.org/>.
95. U.S. General Services Administration (GSA). “Code.gov — Sharing America’s Code.” Accessed 8 Sep. 2019. <https://code.gov/>.
96. Uber Engineering. “The Uber Engineering Tech Stack, Part I: The Foundation.” Posted 19 Jul. 2016. Accessed 8 Sep. 2019. <https://eng.uber.com/tech-stack-part-one/>.
97. Uber Engineering. “The Uber Engineering Tech Stack, Part II: The Edge and Beyond.” 21 Jul. 2016. Accessed 8 Sep. 2019. <https://eng.uber.com/tech-stack-part-two/>.
98. VTK. Accessed 7 Sep. 2019. <https://vtk.org/>.
99. W Yan, Y Lian, C. Yu, (2018), “An integrated-process-property modeling framework for additive manufacturing,” *Computer Methods in Applied Mechanics and Engineering*, 339 pp 184-204.
100. Weerawarana, Sanjiva, Chathura Ekanayake, Srinath Perera, and Frank Leymann. 2018. “Bringing Middleware to Everyday Programmers with Ballerina.” In *Business Process Management*, edited by Mathias Weske, Marco Montali, Ingo Weber and Jan vom Brocke, 12–27. Springer International.
101. Wikipedia. “Create, read, update and delete.” Accessed 1 Oct. 2019. https://en.wikipedia.org/wiki/Create,_read,_update_and_delete.
102. Wikipedia. “DevOps.” Accessed 1 Oct. 2019. <https://en.wikipedia.org/wiki/DevOps>.

103. Wikipedia. "Message broker." Accessed 2 Oct. 2019.
https://en.wikipedia.org/wiki/Message_broker.
104. Wikipedia. "Message-oriented middleware." Accessed 2 Oct. 2019.
https://en.wikipedia.org/wiki/Message-oriented_middleware.
105. Wikipedia. "Microservices." Accessed 24 Aug. 2019.
<https://en.wikipedia.org/wiki/Microservices>.
106. Wikipedia. "Middleware." Accessed 8 Sep. 2019.
<https://en.wikipedia.org/wiki/Middleware>.
107. Wikipedia. "NoSQL." Accessed 30 Sep. 2019. <https://en.wikipedia.org/wiki/NoSQL>.
108. Wikipedia. "Queue (abstract data type)." Accessed 3 Sep. 2019.
[https://en.wikipedia.org/wiki/Queue_\(abstract_data_type\)](https://en.wikipedia.org/wiki/Queue_(abstract_data_type)).
109. Wikipedia. "Remote procedure call." Accessed 3 Sep. 2019.
https://en.wikipedia.org/wiki/Remote_procedure_call.
110. Wikipedia. "System integration." Accessed 7 Sep. 2019.
https://en.wikipedia.org/wiki/System_integration.
111. Wikipedia. "Unit of measurement." Accessed 15 Sep. 2019.
https://en.wikipedia.org/wiki/Unit_of_measurement.
112. Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing." In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2–2. NSDI'12. Berkeley, CA, USA: USENIX Association.

APPENDIX A. CONSTRAINTS MICROSERVICE API

Endpoints

BasicErrorController

errorUsingDELETE

DELETE /error

error

Description

Parameters

Return Type

[???](#)

Content Type

• /

Responses

http response codes

Code	Message	Datatype
200	OK	Map[???]
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

Samples

errorUsingGET

GET /error

error

Description

Parameters

Return Type

[???](#)

Content Type

- /

Responses

http response codes

Code	Message	Datatype
200	OK	Map[???
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

Samples

errorUsingHEAD

HEAD /error

error

Description

Parameters

Return Type

[???](#)

Content Type

- /

Responses

http response codes

Code	Message	Datatype
200	OK	Map[???
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

Samples

errorUsingOPTIONS

OPTIONS /error

error

Description

Parameters

Return Type

[???](#)

Content Type

• /

Responses

http response codes

Code	Message	Datatype
200	OK	Map[???]
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

Samples

errorUsingPATCH

PATCH /error

error

Description

Parameters

Return Type

[???](#)

Content Type

• /

Responses

http response codes

Code	Message	Datatype
200	OK	Map[???]
204	No Content	<<>>
401	Unauthorized	<<>>

403	Forbidden	<<>>
------------	------------------	------

Samples

errorUsingPOST

POST /error

error

Description

Parameters

Return Type

[???](#)

Content Type

• /

Responses

http response codes

Code	Message	Datatype
200	OK	Map[???]
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

Samples

errorUsingPUT

PUT /error

error

Description

Parameters

Return Type

[???](#)

Content Type

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	Map[???
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples***ConstraintController***validateUsingPOST*

POST /constraints/validate

validate

*Description**Parameters**Body Parameter*

Name	Description	Required	Default	Pattern
request	request ConstraintValidationRequest	X		

Return Type

[ConstraintValidationResponse](#)*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	_ConstraintValidationResponse
201	Created	<<>>
401	Unauthorized	<<>>

403	Forbidden	<<>>
404	Not Found	<<>>

Samples

verifyExpressionUsingPOST

POST /constraints/verifyExpression

verifyExpression

Description

Parameters

Body Parameter

Name	Description	Required	Default	Pattern
request	request ConstraintValidationRequest	X		

Return Type

[ExpressionValidationResponse](#)

Content Type

• /

Responses

http response codes

Code	Message	Datatype
200	OK	ExpressionValidationResponse
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

Samples

Models

ConstraintValidationRequest **ConstraintValidationRequest**

Field Name	Required	Type	Description	Format
expression		String		_
variableBindings		List of ConstraintVariable		_

ConstraintValidationResponse ConstraintValidationResponse

Field Name	Required	Type	Description	Format
evaluation		String		Enum: _ VALID, VIOLATION, _
expression		String		_
result		String		_

ConstraintVariable ConstraintVariable

Field Name	Required	Type	Description	Format
expressionVariableName		String		_
parameterType		String		_
valueAsString		String		_

ExpressionValidationResponse ExpressionValidationResponse

Field Name	Required	Type	Description	Format
evaluation		String		Enum: _ VALID, INVALID, _
validationMessage		String		_

ModelAndView ModelAndView

Field Name	Required	Type	Description	Format
empty		Boolean		_
model		Object		_
modelMap		Map of ???		_
reference		Boolean		_
status		String		Enum: _ 100 CONTINUE, 101 SWITCHING_PROTOCOLS, 102 PROCESSING, 103 CHECKPOINT, 200 OK, 201 CREATED, 202 ACCEPTED, 203 NON_AUTHORITATIVE_INFORMATION, 204 NO_CONTENT, 205 RESET_CONTENT, 206 PARTIAL_CONTENT, 207 MULTI_STATUS, 208 ALREADY_REPORTED, 226 IM_USED, 300 MULTIPLE_CHOICES, 301 MOVED_PERMANENTLY, 302 FOUND, 302 MOVED_TEMPORARILY, 303 SEE_OTHER, 304 NOT_MODIFIED, 305 USE_PROXY, 307

				TEMPORARY_REDIRECT, 308 PERMANENT_REDIRECT, 400 BAD_REQUEST, 401 UNAUTHORIZED, 402 PAYMENT_REQUIRED, 403 FORBIDDEN, 404 NOT_FOUND, 405 METHOD_NOT_ALLOWED, 406 NOT_ACCEPTABLE, 407 PROXY_AUTHENTICATION_REQUIRED , 408 REQUEST_TIMEOUT, 409 CONFLICT, 410 GONE, 411 LENGTH_REQUIRED, 412 PRECONDITION_FAILED, 413 PAYLOAD_TOO_LARGE, 413 REQUEST_ENTITY_TOO_LARGE, 414 URI_TOO_LONG, 414 REQUEST_URI_TOO_LONG, 415 UNSUPPORTED_MEDIA_TYPE, 416 REQUESTED_RANGE_NOT_SATISFIABL E, 417 EXPECTATION_FAILED, 418 I_AM_A_TEAPOT, 419 INSUFFICIENT_SPACE_ON_RESOURCE, 420 METHOD_FAILURE, 421 DESTINATION_LOCKED, 422 UNPROCESSABLE_ENTITY, 423 LOCKED, 424 FAILED_DEPENDENCY, 426 UPGRADE_REQUIRED, 428 PRECONDITION_REQUIRED, 429 TOO_MANY_REQUESTS, 431 REQUEST_HEADER_FIELDS_TOO_LAR GE, 451 UNAVAILABLE_FOR_LEGAL_REASONS , 500 INTERNAL_SERVER_ERROR, 501 NOT_IMPLEMENTED, 502 BAD_GATEWAY, 503 SERVICE_UNAVAILABLE, 504 GATEWAY_TIMEOUT, 505 HTTP_VERSION_NOT_SUPPORTED, 506 VARIANT_ALSO_NEGOTIATES, 507 INSUFFICIENT_STORAGE, 508 LOOP_DETECTED, 509 BANDWIDTH_LIMIT_EXCEEDED, 510 NOT_EXTENDED, 511 NETWORK_AUTHENTICATION_REQUI RED, _
view		View		_

viewName		String		–
-----------------	--	---------------	--	---

View View

Field Name	Required	Type	Description	Format
contentType		String		–

APPENDIX B. PROTOCOLS MICROSERVICE API DOCUMENTATION

Endpoints

BasicErrorController

errorHtmlUsingDELETE

DELETE /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

Responses

http response codes

Code	Message	Datatype
200	OK	ModelAndView
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

Samples

errorHtmlUsingGET

GET /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

*Responses**http response codes*

Code	Message	Datatype
200	OK	ModelAndView
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples****errorHtmlUsingHEAD***

HEAD /error

errorHtml

*Description**Parameters**Return Type*[ModelAndView](#)*Content Type*

- text/html

*Responses**http response codes*

Code	Message	Datatype
200	OK	ModelAndView
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

*Samples****errorHtmlUsingOPTIONS***

OPTIONS /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

Responses

http response codes

Code	Message	Datatype
200	OK	ModelAndView
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

Samples

errorHtmlUsingPATCH

PATCH /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

Responses

http response codes

Code	Message	Datatype
200	OK	ModelAndView
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

Samples

errorHtmlUsingPOST

POST /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

Responses

http response codes

Code	Message	Datatype
200	OK	ModelAndView
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

Samples

errorHtmlUsingPUT

PUT /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

*Responses**http response codes*

Code	Message	Datatype
200	OK	ModelAndView
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples***ProtocolController*****availableProtocolsUsingGET***

GET /protocols

availableProtocols

*Description**Parameters**Return Type*array[[Protocol](#)]*Content Type*

• /

*Responses**http response codes*

Code	Message	Datatype
200	OK	List[Protocol]
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples****verifyProtocolUsingPOST***

POST /protocols/verify

verifyProtocol

*Description**Parameters**Body Parameter*

Name	Description	Required	Default	Pattern
request	request Protocol	X		

Return Type

[ProtocolVerifyResponse](#)*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	ProtocolVerifyResponse
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples***Models*****ModelAndView ModelAndView***

Field Name	Required	Type	Description
empty		Boolean	–
model		Object	–
modelMap		Map of ???	–
reference		Boolean	–
status		String	Enum: _ 100 CONTINUE, 101 SWITCHING_PROTOCOLS, 102 PROCESSING, 103 CHECKPOINT, 200 OK, 201 CREATED, 202 ACCEPTED, 203 NON_AUTHORITATIVE_INFORMATION, 204 NO_CONTENT, 205 RESET_CONTENT, 206 PARTIAL_CONTENT, 207

MULTI_STATUS, 208
ALREADY_REPORTED, 226 IM_USED, 300
MULTIPLE_CHOICES, 301
MOVED_PERMANENTLY, 302 FOUND,
302 MOVED_TEMPORARILY, 303
SEE_OTHER, 304 NOT_MODIFIED, 305
USE_PROXY, 307
TEMPORARY_REDIRECT, 308
PERMANENT_REDIRECT, 400
BAD_REQUEST, 401 UNAUTHORIZED,
402 PAYMENT_REQUIRED, 403
FORBIDDEN, 404 NOT_FOUND, 405
METHOD_NOT_ALLOWED, 406
NOT_ACCEPTABLE, 407
PROXY_AUTHENTICATION_REQUIRED,
408 REQUEST_TIMEOUT, 409 CONFLICT,
410 GONE, 411 LENGTH_REQUIRED, 412
PRECONDITION_FAILED, 413
PAYLOAD_TOO_LARGE, 413
REQUEST_ENTITY_TOO_LARGE, 414
URI_TOO_LONG, 414
REQUEST_URI_TOO_LONG, 415
UNSUPPORTED_MEDIA_TYPE, 416
REQUESTED_RANGE_NOT_SATISFIABLE
, 417 EXPECTATION_FAILED, 418
I_AM_A_TEAPOT, 419
INSUFFICIENT_SPACE_ON_RESOURCE,
420 METHOD_FAILURE, 421
DESTINATION_LOCKED, 422
UNPROCESSABLE_ENTITY, 423 LOCKED,
424 FAILED_DEPENDENCY, 426
UPGRADE_REQUIRED, 428
PRECONDITION_REQUIRED, 429
TOO_MANY_REQUESTS, 431
REQUEST_HEADER_FIELDS_TOO_LARGE,
451
UNAVAILABLE_FOR_LEGAL_REASONS,
500 INTERNAL_SERVER_ERROR, 501
NOT_IMPLEMENTED, 502
BAD_GATEWAY, 503
SERVICE_UNAVAILABLE, 504
GATEWAY_TIMEOUT, 505
HTTP_VERSION_NOT_SUPPORTED, 506
VARIANT_ALSO_NEGOTIATES, 507
INSUFFICIENT_STORAGE, 508
LOOP_DETECTED, 509

BANDWIDTH_LIMIT_EXCEEDED, 510
 NOT_EXTENDED, 511
 NETWORK_AUTHENTICATION_REQUIRE
 D, _

view View
 viewName String
 e

Protocol Protocol

Field Name	Required	Type	Description	Format
name		String		–
protocolDetails		Map of ???		–
protocolKeys		List of ???		–

ProtocolVerifyResponse ProtocolVerifyResponse

Field Name	Required	Type	Description	Format
message		String		–
validation		String		Enum: _ VALID, INVALID, _

View View

Field Name	Required	Type	Description	Format
contentType		String		–

APPENDIX C. WORKFLOW SERVICE API DOCUMENTATION

Endpoints

BasicErrorController

errorHtmlUsingDELETE

DELETE /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

Responses

http response codes

Code	Message	Datatype
200	OK	ModelAndView
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

Samples

errorHtmlUsingGET

GET /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

*Responses**http response codes*

Code	Message	Datatype
200	OK	ModelAndView
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples****errorHtmlUsingHEAD***

HEAD /error

errorHtml

*Description**Parameters**Return Type*[ModelAndView](#)*Content Type*

- text/html

*Responses**http response codes*

Code	Message	Datatype
200	OK	ModelAndView
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

*Samples****errorHtmlUsingOPTIONS***

OPTIONS /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

Responses

http response codes

Code	Message	Datatype
200	OK	ModelAndView
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

Samples

errorHtmlUsingPATCH

PATCH /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

Responses

http response codes

Code	Message	Datatype
200	OK	ModelAndView
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

Samples

errorHtmlUsingPOST

POST /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

Responses

http response codes

Code	Message	Datatype
200	OK	ModelAndView
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

Samples

errorHtmlUsingPUT

PUT /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

*Responses**http response codes*

Code	Message	Datatype
200	OK	ModelAndView
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples***ConnectionsController*****createWorkflowConnectionParameterBindingUsingPOST***

POST /connections/parameter/binding

createWorkflowConnectionParameterBinding

*Description**Parameters**Body Parameter*

Name	Description	Required	Default	Pattern
request	request CreateConnectionParameterBindingRequest	X		

Return Type[WorkflowParameterBinding](#)*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	WorkflowParameterBinding
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples****createWorkflowConnectionUsingPOST***

POST /connections

createWorkflowConnection

*Description**Parameters**Body Parameter*

Name	Description	Required	Default	Pattern
request	request CreateConnectionRequest	X		

Return Type

[WorkflowMCIconnection](#)*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	WorkflowMCIconnection
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples****getWorkflowConnectionsUsingGET***

GET /connections/{workflowId}

getWorkflowConnections

*Description**Parameters**Path Parameters*

Name	Description	Required	Default	Pattern
workflowId	workflowId	X	null	

*Return Type*array[[WorkflowMCConnection](#)]*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	List[WorkflowMCConnection]
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples****removeWorkflowConnectionParameterBindingUsingDELETE***

DELETE /connections/parameter/binding

removeWorkflowConnectionParameterBinding

*Description**Parameters**Body Parameter*

Name	Description	Required	Default	Pattern
request	request RemoveConnectionParameterBindingRequest	X		

Return Type[WorkflowParameterBinding](#)*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	WorkflowParameterBinding
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

*Samples****removeWorkflowConnectionUsingDELETE***

DELETE /connections

removeWorkflowConnection

*Description**Parameters**Body Parameter*

Name	Description	Required	Default	Pattern
request	request RemoveConnectionRequest	X		
	<i>Return Type</i>			

[WorkflowMCIConnection](#)*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	WorkflowMCIConnection
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

*Samples***RecommenderController*****recommendAllConnectionsUsingGET***

GET

/recommendations/{workflowId}/connections/{integrationSpecificationId}

recommendAllConnections

Description

Parameters

Path Parameters

Name	Description	Required	Default	Pattern
integrationSpecificationId	integrationSpecificationId	X	null	
workflowId	workflowId	X	null	

Return Type

[RecommendConnectionsResponse](#)

Content Type

- /

Responses

http response codes

Code	Message	Datatype
200	OK	RecommendConnectionsResponse
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

Samples

recommendConnectionParameterBindingsUsingGET

GET /recommendations/{workflowId}/{connectionId}

recommendConnectionParameterBindings

Description

Parameters

Path Parameters

Name	Description	Required	Default	Pattern
connectionId	connectionId	X	null	
workflowId	workflowId	X	null	

Return Type

[RecommendParametersResponse](#)

Content Type

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	RecommendParametersResponse
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples***UserController*****addParameterToUserIntegrationSpecificationUsingPOST***

POST /userinput/parameter

addParameterToUserIntegrationSpecification

*Description**Parameters**Body Parameter*

Name	Description	Required	Default	Pattern
request	request AddUserParameterRequest	X		

Return Type[WorkflowMCI](#)*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	WorkflowMCI
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples****removeParameterFromUserIntegrationSpecificationUsingDELETE***

DELETE /userinput/parameter

removeParameterFromUserIntegrationSpecification

*Description**Parameters**Body Parameter*

Name	Description	Required	Default	Pattern
request	request RemoveUserParameterRequest	X		

Return Type

[WorkflowMCI](#)*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	WorkflowMCI
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

*Samples****setUserInputParameterUsingPOST***

POST /userinput/parameter/value

setUserInputParameter

*Description**Parameters**Body Parameter*

Name	Description	Required	Default	Pattern
request	request SetUserParameterValueRequest	X		

Return Type

[ParameterBinding](#)

Content Type

- /

Responses

http response codes

Code	Message	Datatype
200	OK	ParameterBinding
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

Samples

WorkflowController

addModelCouplingInterfaceToWorkflowUsingPOST

POST /workflows/integration/specification

addModelCouplingInterfaceToWorkflow

Description

Parameters

Body Parameter

Name	Description	Required	Default	Pattern
request	request AddSpecificationRequest	X		

Return Type

[WorkflowMCI](#)

Content Type

- /

Responses

http response codes

Code	Message	Datatype
200	OK	WorkflowMCI

201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

Samples

createFlowUsingPOST

POST /workflows

createFlow

Description

Parameters

Return Type

[Flow](#)

Content Type

• /

Responses

http response codes

Code	Message	Datatype
200	OK	Flow
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

Samples

createUserIntegrationSpecificationForWorkflowUsingPOST

POST /workflows/{workflowId}/userinput

createUserIntegrationSpecificationForWorkflow

Description

Parameters

Path Parameters

Name	Description	Required	Default	Pattern
workflowId	workflowId	X	null	

Return Type[WorkflowMCI](#)*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	WorkflowMCI
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples****deleteModelCouplingInterfaceFromWorkflowUsingDELETE***

DELETE /workflows/integration/specification

deleteModelCouplingInterfaceFromWorkflow

*Description**Parameters**Body Parameter*

Name	Description	Required	Default	Pattern
request	request RemoveSpecificationRequest	X		

Return Type[WorkflowMCI](#)*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	WorkflowMCI
204	No Content	<<>>
401	Unauthorized	<<>>

403 Forbidden <<>>
Samples

getFlowUsingGET

GET /workflows/{workflowId}

getFlow

Description

Parameters

Path Parameters

Name	Description	Required	Default	Pattern
workflowId	workflowId	X	null	

Return Type

[Flow](#)

Content Type

• /

Responses

http response codes

Code	Message	Datatype
200	OK	Flow
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

Samples

getFlowsUsingGET

GET /workflows

getFlows

Description

Parameters

Return Type

[???](#)

Content Type

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	List[???]
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples****getWorkflowIntegrationSpecificationsUsingGET***

GET /workflows/{workflowId}/integration/specifications

getWorkflowIntegrationSpecifications

*Description**Parameters**Path Parameters*

Name	Description	Required	Default	Pattern
workflowId	workflowId	X	null	

*Return Type*array[[WorkflowMCI](#)]*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	List[WorkflowMCI]
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples****persistWorkflowUsingPOST***

POST /workflows/{workflowId}/persist

persistWorkflow

*Description**Parameters**Path Parameters*

Name	Description	Required	Default	Pattern
workflowId	workflowId	X	null	

Return Type

[PersistWorkflowResponse](#)*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	PersistWorkflowResponse
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples****validateWorkflowUsingGET***

GET /workflows/{workflowId}/validate

validateWorkflow

*Description**Parameters**Path Parameters*

Name	Description	Required	Default	Pattern
workflowId	workflowId	X	null	

Return Type[WorkflowValidationResponse](#)*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	WorkflowValidationResponse
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples***Models*****AddSpecificationRequest AddSpecificationRequest***

Field Name	Required	Type	Description	Format
integrationSpecification		ModelCouplingInterface		-
workflowId		UUID		uuid _

AddUserParameterRequest AddUserParameterRequest

Field Name	Required	Type	Description	Format
connectionId		UUID		uuid _
socketMCI		UUID		uuid _
socketParameterId		UUID		uuid _
userMCIAsPlug		UUID		uuid _
workflowId		UUID		uuid _

Constraint Constraint

Field Name	Required	Type	Description	Format
expression		String		-
expressionVariable		String		-
id		String		-
parameterId		UUID		uuid _
uid		UUID		uuid _

CreateConnectionParameterBindingRequest**CreateConnectionParameterBindingRequest**

Field Name	Required	Type	Description	Format
connectionId		UUID		uuid _

plugId	UUID	uuid _
socketId	UUID	uuid _
workflowId	UUID	uuid _

CreateConnectionRequest CreateConnectionRequest

Field Name	Required	Type	Description	Format
plugId		UUID		uuid _
socketId		UUID		uuid _
workflowId		UUID		uuid _

Flow Flow

Field Name	Required	Type	Description	Format
description		String		-
id		String		-
links		List of FlowLinkage		-
name		String		-
tags		List of ???		-
uid		UUID		uuid _

FlowLinkage FlowLinkage

Field Name	Required	Type	Description	Format
id		String		-
plug		ParameterBinding		-
socket		ParameterBinding		-
uid		UUID		uuid _

InvalidIntegrationSpecificationInWorkflow**InvalidIntegrationSpecificationInWorkflow**

Field Name	Required	Type	Description	Format
invalidInputs		List of WorkflowUnboundedParameter		-
workflowMCIId		UUID		uuid _

ModelAndView ModelAndView

Field Name	Required	Type	Description	Format
empty		Boolean		-
model		Object		-
modelMap		Map of ???		-
reference		Boolean		-
status		String	Enum: _ 100 CONTINUE, 101 SWITCHING_PROTOCOLS, 102 PROCESSING, 103 CHECKPOINT, 200 OK,	

201 CREATED, 202 ACCEPTED, 203
NON_AUTHORITATIVE_INFORMATION,
204 NO_CONTENT, 205 RESET_CONTENT,
206 PARTIAL_CONTENT, 207
MULTI_STATUS, 208
ALREADY_REPORTED, 226 IM_USED, 300
MULTIPLE_CHOICES, 301
MOVED_PERMANENTLY, 302 FOUND,
302 MOVED_TEMPORARILY, 303
SEE_OTHER, 304 NOT_MODIFIED, 305
USE_PROXY, 307
TEMPORARY_REDIRECT, 308
PERMANENT_REDIRECT, 400
BAD_REQUEST, 401 UNAUTHORIZED,
402 PAYMENT_REQUIRED, 403
FORBIDDEN, 404 NOT_FOUND, 405
METHOD_NOT_ALLOWED, 406
NOT_ACCEPTABLE, 407
PROXY_AUTHENTICATION_REQUIRED,
408 REQUEST_TIMEOUT, 409 CONFLICT,
410 GONE, 411 LENGTH_REQUIRED, 412
PRECONDITION_FAILED, 413
PAYLOAD_TOO_LARGE, 413
REQUEST_ENTITY_TOO_LARGE, 414
URI_TOO_LONG, 414
REQUEST_URI_TOO_LONG, 415
UNSUPPORTED_MEDIA_TYPE, 416
REQUESTED_RANGE_NOT_SATISFIABLE
, 417 EXPECTATION_FAILED, 418
I_AM_A_TEAPOT, 419
INSUFFICIENT_SPACE_ON_RESOURCE,
420 METHOD_FAILURE, 421
DESTINATION_LOCKED, 422
UNPROCESSABLE_ENTITY, 423 LOCKED,
424 FAILED_DEPENDENCY, 426
UPGRADE_REQUIRED, 428
PRECONDITION_REQUIRED, 429
TOO_MANY_REQUESTS, 431
REQUEST_HEADER_FIELDS_TOO_LARGE,
451
UNAVAILABLE_FOR_LEGAL_REASONS,
500 INTERNAL_SERVER_ERROR, 501
NOT_IMPLEMENTED, 502
BAD_GATEWAY, 503
SERVICE_UNAVAILABLE, 504
GATEWAY_TIMEOUT, 505

HTTP_VERSION_NOT_SUPPORTED, 506
 VARIANT_ALSO_NEGOTIATES, 507
 INSUFFICIENT_STORAGE, 508
 LOOP_DETECTED, 509
 BANDWIDTH_LIMIT_EXCEEDED, 510
 NOT_EXTENDED, 511
 NETWORK_AUTHENTICATION_REQUIRE
 D, _

view View
 viewName String
 e

ModelCouplingInterface ModelCouplingInterface

Field Name	Required	Type	Description	Format
constraints		List of Constraint		–
description		String		–
id		String		–
messageType		String		–
name		String		–
protocol		Protocol		–
protocolDetails		Map of ???		–
results		List of Parameter		–
status		String		Enum: _ SUBMITTED, ACCEPTED, REJECTED, _
tags		List of ???		–
uid		UUID		uuid _
variables		List of Parameter		–
version		String		–

Parameter Parameter

Field Name	Required	Type	Description	Format
id		String		–
name		String		–
quantity		String		–
typeName		String		–
uid		UUID		uuid _
unit		String		–
uri		String		–

ParameterBinding ParameterBinding

Field Name	Required	Type	Description	Format
bindingParameterId		UUID		uuid _
contract		ModelCouplingInterface		–

id		String		–
mciFlowView		UUID		uuid _
uid		UUID		uuid _
value		Object		–

PersistWorkflowResponse PersistWorkflowResponse

Field Name	Required	Type	Description	Format
persistedFlow		String		–
workflowId		UUID		uuid _

Protocol Protocol

Field Name	Required	Type	Description	Format
protocolKeys		List of ???		–

RecommendConnectionsResponse RecommendConnectionsResponse

Field Name	Required	Type	Description	Format
asPlugRecommendations		List of WorkflowMCICConnection		–
asSocketRecommendations		List of WorkflowMCICConnection		–
workflowId		UUID		uuid _
workflowMCIId		UUID		uuid _

RecommendParametersResponse RecommendParametersResponse

Field Name	Required	Type	Description	Format
connectionId		UUID		uuid _
recommendations		List of WorkflowParameterBinding		–
unboundParameters		List of WorkflowUnboundedParameter		–
workflowId		UUID		uuid _

RemoveConnectionParameterBindingRequest

RemoveConnectionParameterBindingRequest

Field Name	Required	Type	Description	Format
connectionId		UUID		uuid _
plugId		UUID		uuid _
socketId		UUID		uuid _
workflowId		UUID		uuid _

RemoveConnectionRequest RemoveConnectionRequest

Field Name	Required	Type	Description	Format
connectionId		UUID		uuid _
workflowId		UUID		uuid _

***RemoveSpecificationRequest* RemoveSpecificationRequest**

Field Name	Required	Type	Description	Format
integrationSpecificationId		UUID		uuid _
workflowId		UUID		uuid _

***RemoveUserParameterRequest* RemoveUserParameterRequest**

Field Name	Required	Type	Description	Format
parameterId		UUID		uuid _
userMCIId		UUID		uuid _
workflowId		UUID		uuid _

***SetUserParameterValueRequest* SetUserParameterValueRequest**

Field Name	Required	Type	Description	Format
parameterId		UUID		uuid _
userMCIId		UUID		uuid _
value		Object		_
workflowId		UUID		uuid _

***View* View**

Field Name	Required	Type	Description	Format
contentType		String		_

***WorkflowMCI* WorkflowMCI**

Field Name	Required	Type	Description	Format
instanceId		UUID		uuid _
mci		ModelCouplingInterface		_

***WorkflowMCIConnection* WorkflowMCIConnection**

Field Name	Required	Type	Description	Format
bindings		List of WorkflowParameterBinding		_
plug		WorkflowMCI		_
socket		WorkflowMCI		_
uid		UUID		uuid _
workflowId		UUID		uuid _

***WorkflowParameterBinding* WorkflowParameterBinding**

Field Name	Required	Type	Description	Format
connectionId		UUID		uuid _
plugParameter		Parameter		_
socketParameter		Parameter		_

***WorkflowUnboundedParameter* WorkflowUnboundedParameter**

Field Name	Required	Type	Description	Format
unboundParameterId		UUID		uuid _

workflowId	UUID	uuid_
workflowMCI	UUID	uuid_

WorkflowValidationResponse WorkflowValidationResponse

Field Name	Required	Type	Description	Format
invalidMCIs		List of InvalidIntegrationSpecificationInWorkflow		-
workflowId		UUID		uuid_

Example usage

```
import requests
import json
import argparse
import os

parser = argparse.ArgumentParser(description="Create a new workflow")
parser.add_argument("url", help="Endpoint of workflow-service")

def create_flow(workflow_service):
    r = requests.post(workflow_service + "/workflows/")
    return r.json()

def create_user_input(workflow_service, workflow_id):
    r = requests.post(workflow_service + "/workflows/" + workflow_id +
"/userinput")
    return r.json()

def add_aci_to_workflow(workflow_service, workflow_id, aci):
    aci_add_request = {
        "integrationSpecification": aci["integrationSpecification"],
        "workflowId": workflow_id
    }
    r = requests.post(workflow_service +
"/workflows/integration/specification", json=aci_add_request)
    return r.json()

def add_connection_to_workflow(workflow_service, workflow_id, plug_id,
socket_id):
    add_connection_request = {
        "plugId": plug_id,
        "socketId": socket_id,
        "workflowId": workflow_id
    }
    r = requests.post(workflow_service + "/connections",
json=add_connection_request)
    return r.json()
```

```

def create_user_parameter_binding(workflow_service, workflow_id,
connection_id, socket_id,
                                socket_parameter_to_add_to_user,
plug_id):
    add_user_parameter = {
        "connectionId": connection_id,
        "socketMCI": socket_id,
        "socketParameterId": socket_parameter_to_add_to_user,
        "userMCIAsPlug": plug_id,
        "workflowId": workflow_id
    }
    r = requests.post(workflow_service + "/userinput/parameter",
json=add_user_parameter)
    return r.json()

def set_user_parameter_value(workflow_service, workflow_id,
user_input_mci_id,
                            parameter_id, value):
    set_user_parameter = {
        "parameterId": parameter_id,
        "userMCIId": user_input_mci_id,
        "value": value,
        "workflowId": workflow_id
    }
    r = requests.post(workflow_service + "/userinput/parameter/value",
json=set_user_parameter)
    return r.json()

def connect_parameter_connection_between_acis(workflow_service,
workflow_id, connection_id, plug_id, socket_id):
    create_parameter_connection = {
        "connectionId": connection_id,
        "plugId": plug_id,
        "socketId": socket_id,
        "workflowId": workflow_id
    }
    r = requests.post(workflow_service + "/connections/parameter/binding",
json=create_parameter_connection)
    return r.json()

def recommend_parameter_bindings(workflow_service, workflow_id,
connection_id):
    r = requests.get(workflow_service + "/recommendations/" + workflow_id
+ "/" + connection_id)
    return r.json()

def validate_workflow(workflow_service, workflow_id):
    r = requests.get(workflow_service + "/workflows/" + workflow_id +
"/validate")

```

```

return r.json()

def persist_workflow(workflow_service, workflow_id):
    r = requests.post(workflow_service + "/workflows/" + workflow_id +
"/persist")
    return r.json()

# def
add_parameter_to_user_input(workflow_service, workflow_id, connection_id,)
def load_aci_from_directory(directory, aci_file):
    aci_file_location = os.path.join(directory, aci_file)

    with open(aci_file_location, 'r') as aci_from_file:
        data = json.load(aci_from_file)

    return data

# simulates what a user working through the UI would do to create a
workflow and the underlying APIS
# in workflow-service that are used
def main():
    args = parser.parse_args()
    print("Using Workflow Service endpoint:", args.url)
    workflow_service = args.url
    # create new flow
    workflow = create_flow(workflow_service)
    print("Created new workflow:", json.dumps(workflow, indent=4))
    # add specification (ACI)
    addition_aci = load_aci_from_directory("./acis",
"addition.workflow.mci")
    print("Adding ACIs to workflow...")
    addition_aci_instance = add_aci_to_workflow(workflow_service,
workflow["uid"], addition_aci)
    print("Add Addition:", addition_aci_instance)
    # create user input
    user_input_aci = create_user_input(workflow_service, workflow["uid"])
    print("Created User input:", user_input_aci)

    # create connection between the two
    print("Creating connection between [{}] --> [{}]
...".format(user_input_aci["mci"]["name"],
addition_aci_instance["mci"]["name"]))

    connection = add_connection_to_workflow(workflow_service,
workflow["uid"], user_input_aci["instanceId"],
addition_aci_instance["instanceId"])
    print("Connection created... ", connection)

    # validate workflow ...should be invalid at this point

```

```

print("Validating workflow")
validation = validate_workflow(workflow_service, workflow["uid"])
is_valid = 1 if len(validation["invalidMCIs"]) == 0 else 0
if not is_valid:
    print("Validation failed:", validation)
    # it failed because there is no connection to the additon aci
    # lets create a user input parameter for it
    # this is the list of invalid parameters for the addition aci
    number_of_unbound_parameters_to_create_user_input_for = 1
    for i in range(0,
number_of_unbound_parameters_to_create_user_input_for):
        invalid_input =
validation["invalidMCIs"][0]["invalidInputs"][0] # theres only one
available, addition
        user_input_binding =
create_user_parameter_binding(workflow_service,
invalid_input["workflowId"],

connection["uid"],

invalid_input["workflowMCI"],

invalid_input["unboundParameterId"],

user_input_aci["instanceId"])
        print("Updated User Input With Parameter...",
user_input_binding)
        # set the value for the user input...here its a float
value...The UI should use ACI info to validate
        # this input before setting it on the parameter binding
        parameter_binding_with_value =
set_user_parameter_value(workflow_service, workflow["uid"],

user_input_binding["instanceId"],

user_input_binding["mci"]["variables"][0]["uid"],

15.3))
        print("Set User input value:", parameter_binding_with_value)

    # now see if there are recommendations for parameter bindings, ie map
user input result to addition variables
    recommendations = recommend_parameter_bindings(workflow_service,
workflow["uid"], connection["uid"])
    print("Recommendations...", recommendations)

    # use the recommendations ... these could be presented to the user to
apply or not
    # there could be multiple recommendations but here we know there is
only one
    print("Applying recommendations...")
    for recommendation in recommendations["recommendations"]:

```

```

        parameter_connection =
connect_parameter_connection_between_acis(workflow_service,
workflow["uid"],

recommendation["connectionId"],

recommendation["plugParameter"]["uid"],

recommendation["socketParameter"]["uid"])
        print("Created parameter connection...", parameter_connection)

        # validate workflow ...should be invalid at this point
        print("Validating workflow")
        validation = validate_workflow(workflow_service, workflow["uid"])
        is_valid = 1 if len(validation["invalidMCIs"]) == 0 else 0
        # should be valid now
        if not is_valid:
            print("Validation failed:", validation)
            raise RuntimeError(validation)

        # save the workflow to the repo
        persisted_workflow = persist_workflow(workflow_service,
workflow["uid"])
        print("Saved workflow...", json.dumps(persisted_workflow, indent=3))

        # todo submit this workflow which has user input
        # todo still need interpret them on the FMS side
        # right now submitting this would just stall out because
        # the FMS is expecting user input to represent a model executing some
code
        # rather than just plucking it from somewhere

if __name__ == "__main__":
    main()

```

APPENDIX D. REGISTRATION MICROSERVICE API DOCUMENTATION

Endpoints

BasicErrorController

errorHtmlUsingDELETE

DELETE /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

Responses

http response codes

Code	Message	Datatype
200	OK	ModelAndView
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

Samples

errorHtmlUsingGET

GET /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

*Responses**http response codes*

Code	Message	Datatype
200	OK	ModelAndView
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples****errorHtmlUsingHEAD***

HEAD /error

errorHtml

*Description**Parameters**Return Type*[ModelAndView](#)*Content Type*

- text/html

*Responses**http response codes*

Code	Message	Datatype
200	OK	ModelAndView
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

*Samples****errorHtmlUsingOPTIONS***

OPTIONS /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

Responses

http response codes

Code	Message	Datatype
200	OK	ModelAndView
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

Samples

errorHtmlUsingPATCH

PATCH /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

Responses

http response codes

Code	Message	Datatype
200	OK	ModelAndView
204	No Content	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>

Samples

errorHtmlUsingPOST

POST /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

Responses

http response codes

Code	Message	Datatype
200	OK	ModelAndView
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

Samples

errorHtmlUsingPUT

PUT /error

errorHtml

Description

Parameters

Return Type

[ModelAndView](#)

Content Type

- text/html

*Responses**http response codes*

Code	Message	Datatype
200	OK	ModelAndView
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples***RegistrationController*****registerIntegrationSpecificationUsingPOST***

POST /integration/specification/register

registerIntegrationSpecification

*Description**Parameters**Body Parameter*

Name	Description	Required	Default	Pattern
mci	mci ModelCouplingInterface	X		

Return Type[RegistrationResponse](#)*Content Type*

- /

*Responses**http response codes*

Code	Message	Datatype
200	OK	RegistrationResponse
201	Created	<<>>
401	Unauthorized	<<>>
403	Forbidden	<<>>
404	Not Found	<<>>

*Samples***Models*****Constraint Constraint***

Field Name	Required	Type	Description	Format
expression		String		–
expressionVariable		String		–
id		String		–
parameterId		UUID		uuid_
uid		UUID		uuid_

ModelAndView ModelAndView

Field Name	Required	Type	Description	Format
empty		Boolean		–
model		Object		–
modelMap		Map of ???		–
reference		Boolean		–
status		String	Enum: _ 100 CONTINUE, 101 SWITCHING_PROTOCOLS, 102 PROCESSING, 103 CHECKPOINT, 200 OK, 201 CREATED, 202 ACCEPTED, 203 NON_AUTHORITATIVE_INFORMATION, 204 NO_CONTENT, 205 RESET_CONTENT, 206 PARTIAL_CONTENT, 207 MULTI_STATUS, 208 ALREADY_REPORTED, 226 IM_USED, 300 MULTIPLE_CHOICES, 301 MOVED_PERMANENTLY, 302 FOUND, 302 MOVED_TEMPORARILY, 303 SEE_OTHER, 304 NOT_MODIFIED, 305 USE_PROXY, 307 TEMPORARY_REDIRECT, 308 PERMANENT_REDIRECT, 400 BAD_REQUEST, 401 UNAUTHORIZED, 402 PAYMENT_REQUIRED, 403 FORBIDDEN, 404 NOT_FOUND, 405 METHOD_NOT_ALLOWED, 406 NOT_ACCEPTABLE, 407 PROXY_AUTHENTICATION_REQUIRED, 408 REQUEST_TIMEOUT, 409 CONFLICT,	

410 GONE, 411 LENGTH_REQUIRED, 412
 PRECONDITION_FAILED, 413
 PAYLOAD_TOO_LARGE, 413
 REQUEST_ENTITY_TOO_LARGE, 414
 URI_TOO_LONG, 414
 REQUEST_URI_TOO_LONG, 415
 UNSUPPORTED_MEDIA_TYPE, 416
 REQUESTED_RANGE_NOT_SATISFIABLE
 , 417 EXPECTATION_FAILED, 418
 I_AM_A_TEAPOT, 419
 INSUFFICIENT_SPACE_ON_RESOURCE,
 420 METHOD_FAILURE, 421
 DESTINATION_LOCKED, 422
 UNPROCESSABLE_ENTITY, 423 LOCKED,
 424 FAILED_DEPENDENCY, 426
 UPGRADE_REQUIRED, 428
 PRECONDITION_REQUIRED, 429
 TOO_MANY_REQUESTS, 431
 REQUEST_HEADER_FIELDS_TOO_LARGE,
 451
 UNAVAILABLE_FOR_LEGAL_REASONS,
 500 INTERNAL_SERVER_ERROR, 501
 NOT_IMPLEMENTED, 502
 BAD_GATEWAY, 503
 SERVICE_UNAVAILABLE, 504
 GATEWAY_TIMEOUT, 505
 HTTP_VERSION_NOT_SUPPORTED, 506
 VARIANT_ALSO_NEGOTIATES, 507
 INSUFFICIENT_STORAGE, 508
 LOOP_DETECTED, 509
 BANDWIDTH_LIMIT_EXCEEDED, 510
 NOT_EXTENDED, 511
 NETWORK_AUTHENTICATION_REQUIRED, _

view View
 viewName String
 e -
 -

ModelCouplingInterface ModelCouplingInterface

Field Name	Required	Type	Description	Format
constraints		List of Constraint		-
description		String		-
id		String		-
messageType		String		-
name		String		-

protocol	Protocol	–
protocolDetails	Map of ???	–
results	List of Parameter	–
status	String	Enum: _ SUBMITTED, ACCEPTED, REJECTED, _
tags	List of ???	–
uid	UUID	uuid _
variables	List of Parameter	–
version	String	–

Parameter Parameter

Field Name	Required	Type	Description	Format
id		String		–
name		String		–
quantity		String		–
typeName		String		–
uid		UUID		uuid _
unit		String		–
uri		String		–

Protocol Protocol

Field Name	Required	Type	Description	Format
protocolKeys		List of ???		–

RegistrationResponse RegistrationResponse

Field Name	Required	Type	Description	Format
message		String		–
validation		String		Enum: _ VALID, INVALID, _

View View

Field Name	Required	Type	Description	Format
contentType		String		–

APPENDIX E. ATTRIBUTIONS

Font Awesome was used for some images in this work <https://fontawesome.com/license>