Towards an Open Platform for Democratized Model-Based Design and Engineering of Cyber-Physical Systems

Mohamad Omar Nachawati[1] Gianmaria Bullegas[1] Andrey Vasilyev[1] Joe Gregory[1] Adrian Pop[2]
Maged Elaasar[3] Adeel Asghar[4]

[1]Perpetual Labs Ltd., UK, {omar, gian, andrey}@perpetuallabs.io
[2]Linköping University, Sweden, adrian.pop@liu.se
[3]NASA Jet Propulsion Laboratory, USA, maged.e.elaasar@jpl.nasa.gov
[4]Open Source Modelica Consortium, Sweden, adeel.asghar@liu.se

Abstract

This paper reports on the development of GitWorks, an open platform for democratizated Model-Based Design of cyber-physical systems (CPS). The GitWorks platform is currently under development by Perpetual Labs Ltd in collaboration with the Open Source Modelica Consortium (OSMC)[1] and the OpenCAESAR project[2]. In this paper, we present our vision for the platform, its system architecture and a prototype implementation. We also present a case study that demonstrates the use of the proposed platform for enabling the seamless integration of Modelica models into a broader range of systems engineering processes for complex product development. In the long-term, the platform also aims to enable the integration of Modelica tools with advanced systems engineering processes that rely on other domain specific languages (e.g. SysML v2, BPMN, etc.).
Keywords: MBD, MBSE, Modeling, Simulation, Interoperability, Cyber-Physical Systems, Semantic Twin, Realtime Collaboration

1 Introduction

The Modelica language (Elmqvist, Mattsson, and Otter 1998; Fritzson and Engelson 1998) has a growing user community that produces a large and constantly-increasing code base of models. However, there is a lack of tools to address a number of advanced model-management use cases, such as semantic search, analysis, cross-referencing, checking, component selection automation, for a large body of models (Johansson, Pop, and Fritzson 2005). Despite recent developments (Sirin et al. 2015; Isasi, Noguerón, and Wijnands 2015; Hussain et al. 2022), tool support for the integration of Modelica models into advanced Model-Based Systems Engineering (MBSE) practices remains limited (Larsen et al. 2016). This hinders the reuse of models within the Modelica community, and particularly in an industrial context, can greatly limit the potential for adoption of Modelica tools within integrated Model-Based Design (MBD) and product development processes.

There are multiple engineering processes that precede modeling and simulation within a complex product development lifecycle. The information generated by these processes defines the structure, configuration, and input parameter data used by the executable system models. For example:

1. The definition of operational scenarios and associated system requirements. These define the critical behaviors that the system must achieve, the circumstances under which they must be achieved, and other non-functional properties of the product.

2. The definition of the high-level architecture of the system. This includes the system's hierarchical division into different subsystems, their components, parameter values, and interconnections. Alternative design solutions can be evaluated against the criteria defined in (1) via simulation.

Many tools and formalisms can be used in these phases to capture the system information as part of an MBSE framework, such as UML, SysML, AADL, FMDesign, BDPM, and OPM (J. Ma et al. 2022; Basnet et al. 2022). Integration of system simulation and analysis with such MBSE models is difficult to achieve, particularly during the early phases of the system lifecycle, because domain-specific models often lack a common notation (Madni and Sievers 2018). Collecting, aggregating and exchanging information at the system level is complex and often error-prone, which hampers system-wide visibility in a multidisciplinary concurrent design setting (McDermott et al. 2020). This limits the ability to analyze system-level requirements (such as performance and dependability) in the early design phases, causing the postponement of design decisions to later phases. This, in turn, reduces possible opportunities to study alternative solutions and validation of fitness for purpose. It also increases costs, in terms of time and skill, of design refinements if irreversible consecutive design decisions are made in the early stages of the development (Stirgwolt, Mazzuchi, and Sarkani 2022).

[1]https://openmodelica.org/home/consortium
[2]https://www.opencaesar.io/

 10.3384/ECP21186102 DOI

The limitations of current MBSE frameworks and their poor integration with system simulation environments, such as Modelica-based tools, contribute to an increased barrier-to-entry for the adoption of MBD. These issues are especially acute for Small and Medium Enterprises (SMEs) that typically do not have the resources to link Commercial-Off-The-Shelf (COTS) tools into integrated tool chains and lack the in-house expertise to develop custom models from scratch.

GitWorks aims to democratize access to MBD for SMEs, independent developers and academia. GitWorks has been designed as a turn-key solution for model management and integration provided with the convenience of a Software as a Service (SaaS) product. It includes the GitWorks Commons which provides a searchable repository of models, tools and services with a try-before-you-buy business model. It also includes a Web application for integrated data and knowledge management, as well as a web-based collaborative Modelica editor.

This paper describes the early design and implementation of GitWorks. Specifically, the contributions of this paper are three-fold: First, we propose a system architecture for the platform to support model-based design and engineering of cyber-physical systems. Second, we develop a prototype implementation of the GitWorks platform that is focused on enabling the seamless integration of Modelica models into a broader range of MBSE activities. Third, we conduct a preliminary case study to demonstrate the use of the proposed platform for the federated design and engineering of an aircraft passenger air conditioner (PACK) system.

The rest of this paper is organized as follows. Section 2 provides an overview of our vision for the GitWorks platform, describing its design goals, conceptual system architecture, and user interfaces. Section 3 describes the prototype implementation of GitWorks and tooling for enabling the use of Modelica in the larger MBSE process. Section 4 presents a preliminary case study to demonstrate the use of the GitWorks for the federated design and engineering of a PACK system. Finally, Section 5 concludes the paper and provides some brief remarks on directions for future work.

2 GitWorks Platform Overview

This section provides an overview of the GitWorks platform, describing its design principles and conceptual system architecture. GitWorks aims to overcome several of the limitations of current systems engineering practices (Elaasar et al. 2019) by introducing three key concepts and their related functionalities:

DEMOps: DevOps for Digital Engineering and Manufacturing. Poor configuration management (CM) practices exacerbate trust issues in current MBSE practices. DEMOps introduces the notion of a Git-like history of changes made across inter-related model fragments. Enables *traceability* of information provenance and design decisions. This refers to the ability to trace from an authority to its design decisions and constraints, and from the latter to their rationales. Without this capability, a system description becomes a disorganized collection of information artifacts. Enables *repeatability*. This refers to the ability to encapsulate the analysis of the system description, including its dependencies, such that it becomes repeatable. This is important to maintain confidence in the analysis over time and use it to assert desirable properties. Enables *durability*. This refers to the ability to version control the information that describes or analyzes a system in such a way that versions become immutable. Without this, it is impossible to perform audits and repeat analyses. Enables *efficiency*. This refers to the ability to automate processes using CI/CD practices that would otherwise be manual and tedious (Elaasar et al. 2019). Without this, such processes become expensive and error-prone. In GitWorks, these functionalities are fulfilled by the Projects environment described in Section 2.1.

Semantic Twin. System information is captured using a precise language that is rooted in mathematics and formal logic. System descriptions are specified using common vocabularies consisting of concepts and their properties and relationships all expressed in a formal language. This enables *digital continuity*, which refers to the interoperability of system information contained in different information artifacts that are produced by different parties during different phases of the product lifecycle. It also enables the augmentation or, in some cases, the replacement of typically human-led processes (such as reporting, model transformation, and validation & verification of system information) through the use of powerful *automations* such as logical reasoning, machine learning, data mining, etc. In GitWorks, these functionalities are fulfilled by the OML language and the versioned triple store and associated reasoner (see Section 2.1).

Digital Prototype. In the context of DEMOps, *Digital Prototype* refers to the usage of digital environments to facilitate the co-simulation of engineering models, connection with HardWare-In-the-Loop (HWIL) frameworks and real-time system data to support *Virtual system Integration*, Validation and Verification from the early stages of the product lifecycle. In GitWorks, these functionalities are fulfilled by the Modelica Studio environment (see Section 3). In its current version, Modelica Studio only supports editing and simulation of Modelica-based models. In the future, we aim to introduce advanced functionalities and analyses, such as co-simulation of FMUs, HWIL, model surrogatization, optimization, uncertainty quantification, etc. The Digital Prototype must be supported by a scalable, cloud-based computational infrastructure to enable the more computationally-demanding workflows, and must also be integrated with the CI/CD pipeline to enable automation. In GitWorks, this is achieved via the integration of a standard CI/CD pipeline with a scalable HPC environment (see Section 2.1).

It should be noted that the concepts of the Digital Proto-

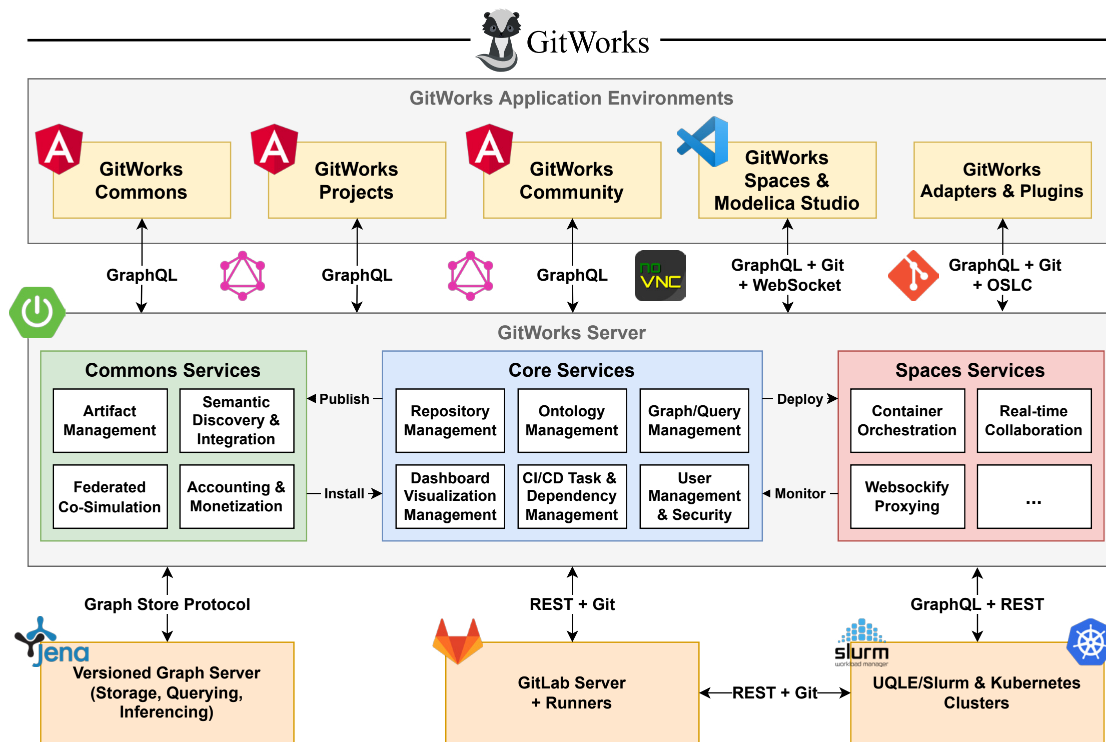


Figure 1. Conceptual software architecture of the GitWorks Platform[3]

type and the Semantic Twin are tightly linked and that the integration of the two enables the realization of new and powerful workflows. The Digital Prototype acts as a backbone to organize and facilitate access to system-related information scattered through the different engineering artifacts. An example application of this idea is provided in the PACK case study in Section 4. The system architecture information contained in a SysML model is used to automatically generate the high-level structure of the Modelica model. Also, queries against the GitWorks Commons, enabled by the OML language representation of Modelica, can be used to find suitable, port-compatible components to complete the model.

2.1 Conceptual System Architecture

The high-level software architecture of the GitWorks platform is shown in Figure 1. From the perspective its users, GitWorks provides multiple application environments for interacting with the platform, to include the *GitWorks Commons* for the publishing and reuse of digital engineering artifacts, the *GitWorks Projects* environment for Semantic Twin-powered DevOps for digital engineering, analysis, reporting and management of the digital thread. In addition to these environments, through Git, OSLC and REST-based APIs, GitWorks is also designed to integrate with third-party adapters and plugins for authoring and reporting.

Supporting these application environments, *GitWorks Server* is designed as a middleware of essential services, as well as a nexus for accessing data across organizations and third-party tools and systems. The GitWorks Server is implemented as a Spring Boot application and depends on a GitLab server to provide Git repository hosting and CI/CD capabilities. GitLab[4] is an open-source DevOps platform based on the popular Git version control system. GitLab provides a REST API for programmatic access and manipulation of resources, such as repositories, artifacts and users. This API is used to implement much of the Git-centric capabilities provided by GitWorks through GitLab4J[5].

Unlike traditional version control systems, such as CVS, where changes are managed at the file level, Git manages changes at the repository level so that for any particular commit one can recover the precise state of entire repository at that point. To facilitate the management of artifacts developed and owned by different stakeholders, repositories can be organized into groups that capture the hierarchical relationships between systems and their components or the relationships among the different entities participating in a supply chain.

Leveraging Git's repository level change management mechanism, from the artifacts contained in a GitWorks project repository, GitWorks constructs a versioned Semantic Twin that captures the interdependencies and trace-

[3] All trademarks, logos and brand names are the property of their respective owners.

[4] https://www.gitlab.com

[5] https://github.com/gitlab4j/gitlab4j-api

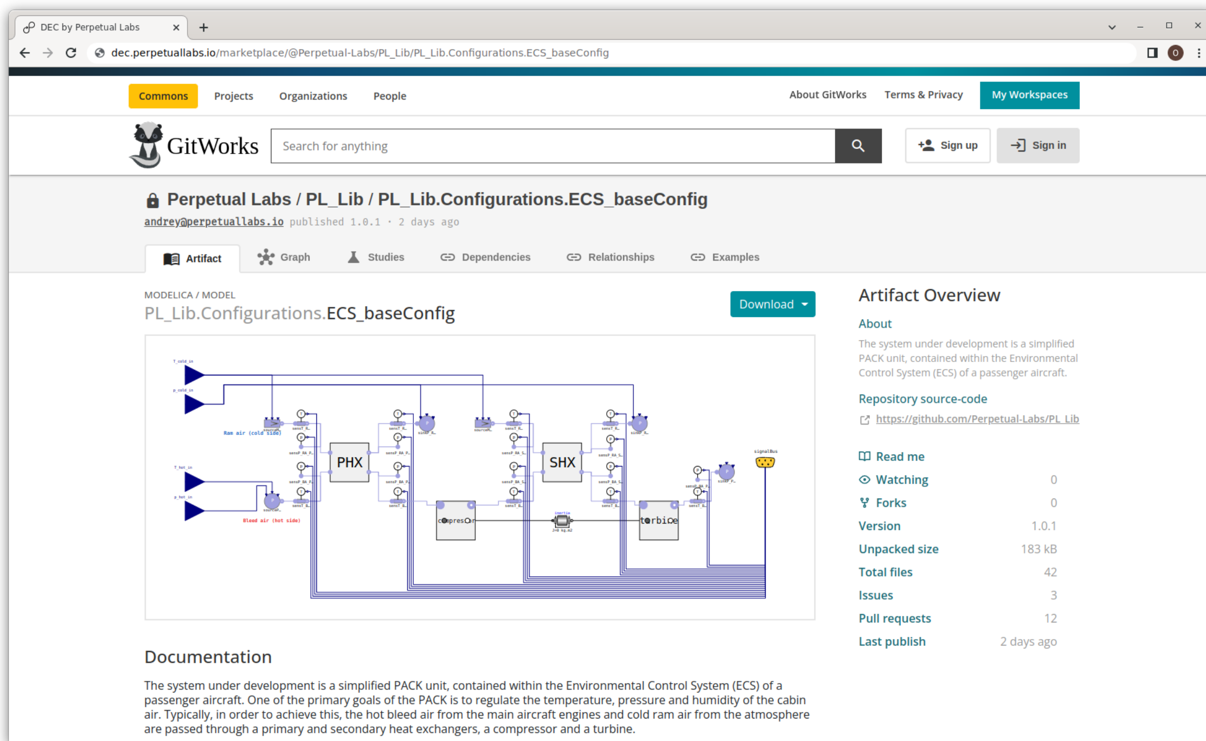


Figure 2. The GitWorks Commons UI for Modelica and other digital engineering artifacts

ability links that relate heterogeneous artifacts through the product lifecycle. The Semantic Twin forms a federated knowledge graph that is stored as RDF triples in a custom versioned triple store, which is developed by Perpetual Labs and is based on Jena (Carroll et al. 2004) TDB2. By creating trace links between heterogeneous elements, such as requirements, simulation models, simulation results, and test results, GitWorks provides traceability of design changes and branches at the system level and throughout the product lifecycle, while also allowing each stakeholder the necessary flexibility in managing their own datasets and internal development cycles.

GitWorks provides multiple adapters for automatically enriching the Semantic Twin from a project repository. While the focus of this paper is on the Modelica adapter (see Section 3.3), Perpetual Labs is actively working on adapters for other representations, including SysML and CAD. These adapters read artifact files from the Git repository, for example a Modelica library, and extract a semantic representation in the Ontology Modeling Language (OML).

OML serves as the core ontological language for the GitWorks platform and was originally developed by the Jet Propulsion Laboratory as part of the CAESAR project (Wagner et al. 2020). One of the goals of CAESAR has been to provide a set of OML vocabularies that capture some of the common concepts and relations used in systems engineering (Bayer et al. 2021). OML provides a foundation with well-defined semantics that can be used to model different types of engineering artifacts in a semantically consistent and interoperable fashion. OML extends OWL 2 DL (Web Ontology Language 2 - Description Logic) in such a way that retains the benefits of OWL 2 DL while addressing some of its limitations (Wagner et al. 2020).

GitWorks Commons. The GitWorks Commons enables the seamless reuse of design and engineering artifacts across tool, domain and organization boundaries. Within the GitWorks platform, an artifact is considered as the basic unit of reuse, where its coarsity depends on the tool and domain vocabulary. As shown in Figure 2, a single Modelica file, for example, may contain multiple models, each of which would be considered an artifact in the Commons. When the artifacts from a project repository are ready for release, these artifacts along with their dependencies are bundled together as package and then uploaded to the Commons as a versioned artifact. The GitWorks leverages GitLab Package Registries to support multiple package managers, including Maven and npm. The Commons also supports semantic discovery and integration of artifacts through a SPARQL endpoint that can be used to query the metadata extracted from artifacts via specific adapters (see Section 3.3).

The GitWorks Commons also provides vendors with multiple publication and deployment possibilities. Vendors can choose whether to allow users to download sources and/or binaries, or only provide cloud-based access. For example, a Modelica model can be published

as: (1) source code (i.e. a model library), (2) compiled FMU (Model-Exchange or Co-Simulation), and/or (3) as a REST service that can be used in a federated co-simulation. We are investigating different monetization strategies for the GitWorks similar to those proposed for the Digital Manufacturing Commons (DMC) (Beckmann et al. 2016), including (1) payment for each download or execution, (2) payment based on computational usage, and (3) freemium software as a service. To support federated collaboration for enhanced data security and intellectual property protection, we are also actively working on the integration of co-simulation engines, specifically Maestro2 by the INTO-CPS project(Larsen et al. 2016).

GitWorks Projects. The Projects environment is a Semantic Twin-powered Web application for integrated data and knowledge management, and exploration and visualization of the digital thread across multiple disciplines, organizations and product lifecycle stages. It enables the exploration, querying, and modification of OML-based knowledge base using a Web-based GUI, similar to WebProtégé (Tudorache, Vendetti, and Noy 2008) for OWL2, and provides customizable OML vocabularies for different cyber-physical system lifecycle activities, such as requirements analysis, system modeling, verification and maintenance. At its core, a project corresponds to a Git repository of OML vocabularies and descriptions, which can be cloned and edited using an OML IDE, such as Rosetta[6] or Luxor[7].

The GitWorks platform leverages GitLab to provide DevOps capabilities for project repositories, to include Git-based version control, issue management, and CI/CD. To enable Semantic Twin-powered authoring and reporting, every project repository on GitWorks is backed by both a Git repository and a corresponding RDF triple store. The versioned RDF triple store serves as a cache to accelerate semantic queries against the repository and can be reconstructed directly from the files in the Git repository. A project can import artifacts from the GitWorks Commons as dependencies, forming a federated knowledge graph that enables all the stakeholders of a complex engineering system to make specific system information and data available to other project participants independently of the specific tools that they are using (i.e. a Semantic Twin).

An HPC CI environment based on GitLab Runner and the Slurm Workload Manager[8] is under active development, and enables computational expensive analyses such as simulation-based requirements verification, uncertainty quantification and optimization to be seamlessly integrated into the DEMOps pipeline. We have already tested different analysis toolkits using our HPC CI environment, to include UncertainPy (Tennøe, Halnes, and Einevoll 2018) and Dakota (Adams et al. 2020). We are working to provide seamless support for surrogate-assisted methods to accelerate computationally expensive analyses using methods such as pre-trained surrogate models for accelerated simulation, as done by JuliaSim (Rackauckas et al. 2021), and dynamically generated surrogates, as done by GreyOpt (Nachawati and Brodsky 2021), for enhanced optimization.

GitWorks Community. Finally, the Community environment enables users and organizations on the platform to connect with one another in a kind of social network for Digital Engineering. Each user is provided with a profile page that contains a public bio with an activity stream and links to associated published artifacts, project workspaces, and organizations.

3 Modelica Tooling for the GitWorks

This section describes the prototype implementation of the GitWorks tooling for enabling the use of Modelica in the larger MBSE process. Specifically, we report on the progress of our development of: (1) Modelica Studio, a Semantic Twin-powered Modelica text and diagram editor for VSCode for Web, (2) OMFrontend.js[9], a reusable and open-source AGPLv3-licensed library for the parsing and analysis of Modelica source code, which serves as the foundation of Modelica Studio, and (3) the ModelicaOML adapter, also based on OMFrontend.js, for automatically enriching the Semantic Twin from Modelica artifact repositories.

3.1 Modelica Studio

We have developed Modelica Studio as a VSCode for Web extension that serves as a Semantic Twin-powered authoring environment for Modelica. Modelica Studio, shown in Figure 3, is designed to support three levels of collaboration: (1) federated, in-the-large collaboration enabled by the GitWorks Commons, (2) Git-based collaboration, using branches and pull requests, and (3) real-time collaboration, using the VSCode LiveShare extension[10].

While several attempts have been made towards the development of a Web-based Modelica editor, significant limitations preclude their use as a collaborative Modelica development environment for the GitWorks:

Modelon Impact (Elmqvist, Malmheden, and Andreasson 2019) is a closed-source, cloud-based platform that provides a Modelica diagram and code editor that runs in a Web browser. While it runs in the browser, the implementation appears to require an independent server-side session for each editor instance, where the Optimica Compiler Toolkit (OCT) is used to construct and maintain a semantic model that mirrors what is opened in the editor. This approach simplifies the logic on the client-side, however, the critical dependency on continuous server-side processing for each editor instance can quickly add up

[6] https://github.com/opencaesar/oml-rosetta/

[7] https://github.com/opencaesar/oml-luxor

[8] https://slurm.schedmd.com/documentation.html

[9] https://github.com/OpenModelica/OMFrontend.js

[10] https://visualstudio.microsoft.com/services/live-share/

 10.3384/ECP21186102 DOI

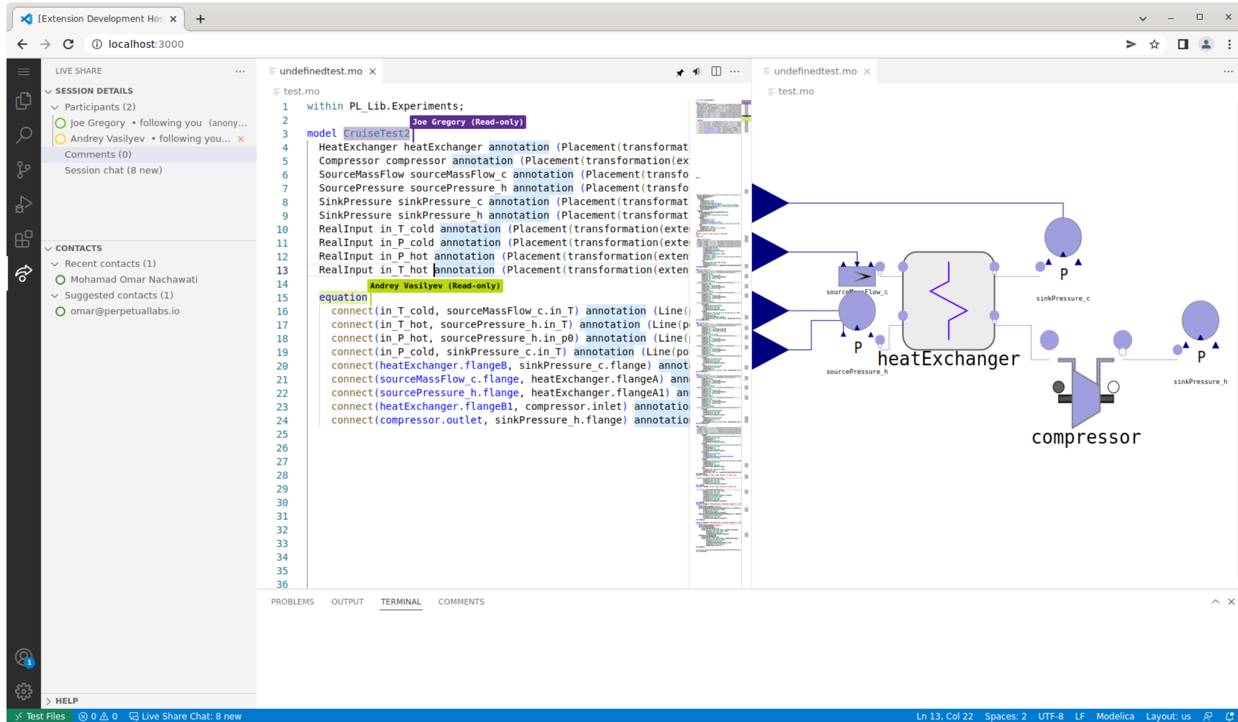


Figure 3. Modelica Studio, a Semantic Twin-powered Modelica editor extension for VSCode for Web

to extensive computational resource requirements, especially for open DevOps and collaborative platforms, such as GitHub. Although the Modelon community provides an open-source JavaScript client[11], it is tightly-coupled to the Modelon Impact platform.

WebMWorks (Wan et al. 2013) is also a closed-source, cloud-based platform that provides a Web-based Modelica diagram and code editor. WebMWorks follows a similar implementation approach to Modelon Impact, but uses the OpenModelica Compiler (OMC) (Fritzson, Pop, et al. 2020) instead of OCT to maintain the server-side, semantic model that mirrors what is opened in the editor. TongYuan, the developers of WebMWorks, do not appear to currently offer access to WebMWorks.

WebGME-DSS (Kecskes et al. 2019) is an open-source, cloud-based modeling environment that provides a Web-based Modelica diagram-only editor. The implementation is based on GME, where a translator is used to convert the interface of a Modelica model into an instance of a GME-based meta-model representing a subset of the Modelica language. Although WebGME-DSS is open-source (MIT-licensed), WebGME-DSS only supports a small subset of the Modelica language and does not appear to be actively maintained.

OMWeb (Torabzadeh-Tari et al. 2011) is an open-source platform for editing and simulating Modelica models in a Web browser. Although OMWeb provides the ability to edit Modelica code and visualize simulation results, it does not provide a Web-based Modelica diagram editor. Furthermore, OMWeb appears to be in maintenance mode.

The diagram editor of Modelica Studio provides a user experience similar to that of OMEdit (Fritzson, Pop, et al. 2020), supporting the composition of new Modelica models by dragging and dropping Modelica model components onto the canvas. Changes made in the diagram editor are immediately propagated to the text editor, and vice versa. Noteably, the component palette in Modelica Studio is also designed to integrate with the GitWorks Commons to provide seamless dependency management. Also, unlike the other previously mentioned Web-based Modelica editors, Modelica Studio is largely serverless and the rendering and editing of Modelica models is done on the client without requiring a heavy-weight remote process to maintain a corresponding semantic representation of the contents of the editor. This design decision was made to significantly improve the scalability of the platform and to help realize the goal of making the GitWorks an open platform for digital engineering.

As shown in Figure 4, Modelica Studio largely depends on the OMFrontend.js (see Section 3.2) for the implementation of the Modelica Language Server to provide text and diagram editing support for VSCode for Web. Modelica Studio is designed to integrate with the OMWebService[12] REST API for the simulation of Modelica models in the browser. OMWebService is developed largely as a

[11] `https://github.com/modelon-community/impact-client-js`

[12] `https://github.com/OpenModelica/OMWebService`

wrapper around the OMC and OMSimulator for simulating Modelica models, Functional Mock-up Units (FMU), and System Structure and Parameterization (SSP) models. In response to a request to run a simulation on a model, OMFrontend.js sends the flattened simulation model to OMWebService. The results of the simulation are then returned as a CSV file, which can then be plotted in Modelica Studio or a third-party application. OMWebService is developed in Python and uses the OMPython interface to communicate with the OMC.

3.2 OMFrontend.js

To facilitate the development of Modelica Studio and other Modelica language tools, we have developed the OMFrontend.js library that provides an API for analyzing and manipulating Modelica text documents in both Node.js and Web browser environments. This library handles Modelica parsing, instantiation, flattening, expression evaluation as well as diagram and SVG icon rendering. It simplifies the implementation of language service features in Modelica Studio (see Section 3.1), such as diagnostics, hovers, links, completion, folding, and formatting. We also use OMFrontend.js to implement the ModelicaOML adapter (see Section 3.3) for enriching the Semantic Twin automatically from Modelica source code.

OMFrontend.js provides a context object that manages the collection of opened documents and Modelica libraries and serves as the mechanism for handling references to Modelica classes defined in different files. The context object seamlessly resolves Modelica files stored on the local file system, virtual Web browser file system, and via HTTP, depending on whether it is running in a Node.js or Web browser environment.

While the OpenModelica compiler uses a parser that is generated from an ANTLR3 grammar, to support browser-based editing we found the need to develop a new parser for OMFrontend.js. The new Modelica parser is built using the tree-sitter[13] parser generator. The tree-sitter-modelica[14] project contains the Modelica grammar for the tree-sitter parser generator. OMFrontend.js then constructs an abstract syntax tree from the tree-sitter concrete syntax tree. This abstract syntax tree is akin to the class tree described in the Modelica Language Specification (MLS). Unlike the tree-sitter concrete syntax tree which is incrementally reparsed, the abstract syntax tree needs to be reconstructed every time the underlying text changes. To reduce latency, this is done in a lazy fashion inspired by the red/green trees[15].

3.3 ModelicaOML Adapter

The purpose of the ModelicaOML adapter is to handle the conversion of Modelica source code and OML conforming to the ModelicaOML vocabulary presented in Figure 5. This enables the automatic enrichment of the Semantic Twin directly from the Modelica artifact repositories based on the representation provided by OMFrontend.js.

The ModelicaOML vocabulary[16] defines concepts such as `Class`, `Block`, `Model`, `Package`, `Function`, `Record`, `Type` to model the Modelica class restrictions and `Component` to model the components. The component and class prefixes are also modeled as scalar enumerations: `Prefix` and `ClassPrefix`. There are also relations that bind these concepts together such as `hasType`, `hasPrefix`, `hasClassPrefix`, `contains`, `extendsClass`, etc.

4 PACK Case Study

This case study aims to demonstrate how the GitWorks platform enables collaborative and federated design and development throughout the systems engineering process.

The system under development is a simplified PACK unit, which is itself a subsystem of the Environmental Control System of a passenger aircraft. One of the primary goals of the PACK is to regulate the temperature, pressure and humidity of the cabin air (Jennions et al. 2020).

4.1 Federated Development of the PACK System

The PACK project comprises seven tasks with a focus on the systems engineering process: Define project; Define system requirements; Define system architecture; Define subsystem behavior; Perform analysis; Verify system requirements; Generate Reports. These tasks are to be performed by specialists with various roles: Project Manager, System Architect, Lead Systems Engineer and a team of Design Engineers.

The tasks are performed using either third-party or GitWorks-hosted tools (e.g. Modelica Studio), and each task yields one or more artifacts with a particular filetype (e.g. a SysML or a Modelica model). The GitWorks platform supports this use case in the manner summarized in Figure 6. This diagram describes how OML adapters are employed to consolidate the knowledge contained in different modeling artifacts by creating a unified, RDF-based Semantic Twin.

In this case, the Project Manager creates an OML Git repository (corresponding to the project) using the GitWorks Workbench environment. The knowledge regarding the participating organizations, teams and people is captured by importing the relevant OML vocabularies and populating an OML description file. In the Workbench environment, the Project Manager defines the relevant project tasks and assigns responsibility to members of the project. A small section of the resulting OML description is presented in Listing 1, in which five roles have been defined and tasks assigned.

[13] `https://tree-sitter.github.io/tree-sitter/`
[14] `https://github.com/OpenModelica/tree-sitter-modelica`
[15] `https://ericlippert.com/2012/06/08/red-green-trees/`
[16] `https://github.com/OpenModelica/ModelicaOML`

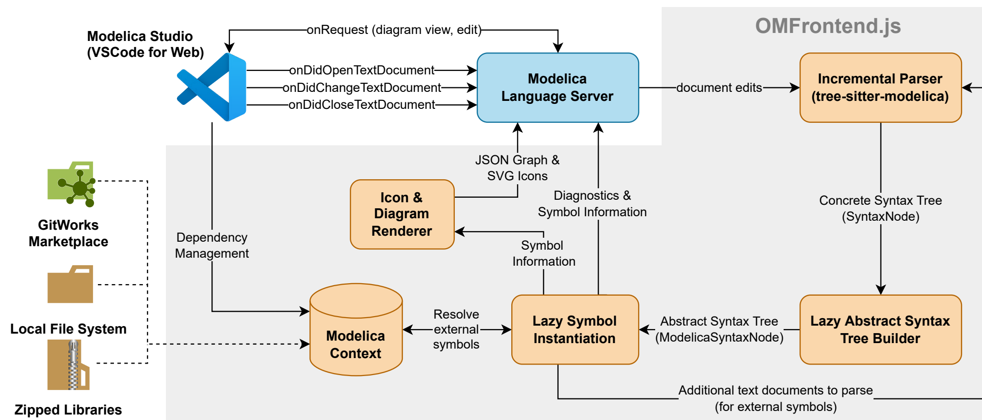


Figure 4. Modelica Studio and OMFrontend.js flow diagram

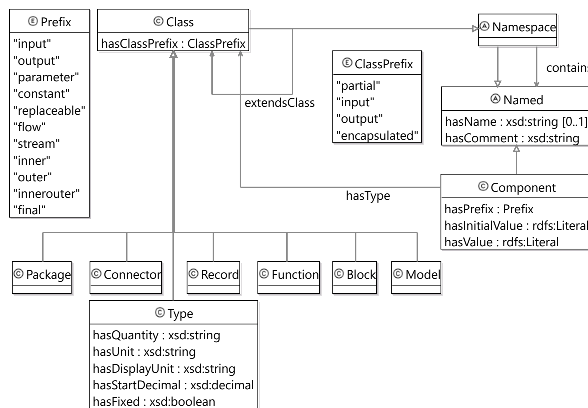


Figure 5. The Modelica Ontology (ModelicaOML Vocabulary)

Listing 1. OML representation of project role definitions

```
ci ProjectManager : project:Role [
    project:hasAssignment DefineProject]
ci SystemEngineer : project:Role [
  project:hasAssignment DefinePACKRequirements]
ci SystemArchitect : project:Role [
  project:hasAssignment DefinePACKArchitecture]
ci DesignEngineer1 : project:Role [
    project:hasAssignment
        DefineSubsystemBehavior]
ci DesignEngineer2 : project:Role [
    project:hasAssignment PerformAnalysis
    project:hasAssignment VerifyPACKRequirements
    project:hasAssignment GenerateReports]
```

Once the Project Manager creates the first commit, the CI/CD pipeline runs the build scripts which interprets the OML code into RDF triples and ingests them into the triple store, effectively integrating the information into the Semantic Twin. The Project Manager can then save the resulting project structure as a template and publish it as an OML artifact in the GitWorks Commons for future reuse and sharing. Once the project repository has been created, the other participants can review the tasks that have been assigned to them, and begin contributing to the project.

The lead Systems Engineer defines the overall systems requirements using a SysML tool. The three requirements for this system are defined as follows:

1. The mass of the PACK shall be no greater than 100kg.
2. The PACK shall produce Conditioned Air with a temperature to ±1°K of 293 °K.
3. The PACK shall produce Conditioned Air with a pressure to ±3kPa of 101 kPa.

The Systems Architect then defines the PACK architecture, also using a SysML tool. For the purposes of this case study, a simplified architecture for the PACK, comprising only the primary heat exchanger and the compressor, is developed. Figure 7 represents this composition of the PACK and the air flows into, out of, and within the system. It is also assumed that the three system requirements apply to the 'Cruise' scenario, during which the temperature and pressure of the input air flows are assumed to be fixed and known.

By specifying the SysML repositories as dependencies within the overall project, the SysML artifacts created by the actors (in this case, requirements and architecture) are translated into OML and populate the Semantic Twin in accordance with the corresponding OML vocabularies. This translation is performed by the dedicated SysMLOML adapter within the GitWorks platform. An example of the resulting OML description (translated from SysML) is presented in Listing 2. In this listing, only a portion of the PACK architecture definition is presented - the flows between the interfaces are also captured by the OML description but are not shown.

Listing 2. OML representation of PACK architecture

```
ci PACK : mission:Component [
    base:contains HeatExchanger
    base:contains Compressor
    PL_Mech:hasMass PACKMass]
ci HeatExchanger : mission:Component [
    mission:presents p1
    mission:presents p2
```

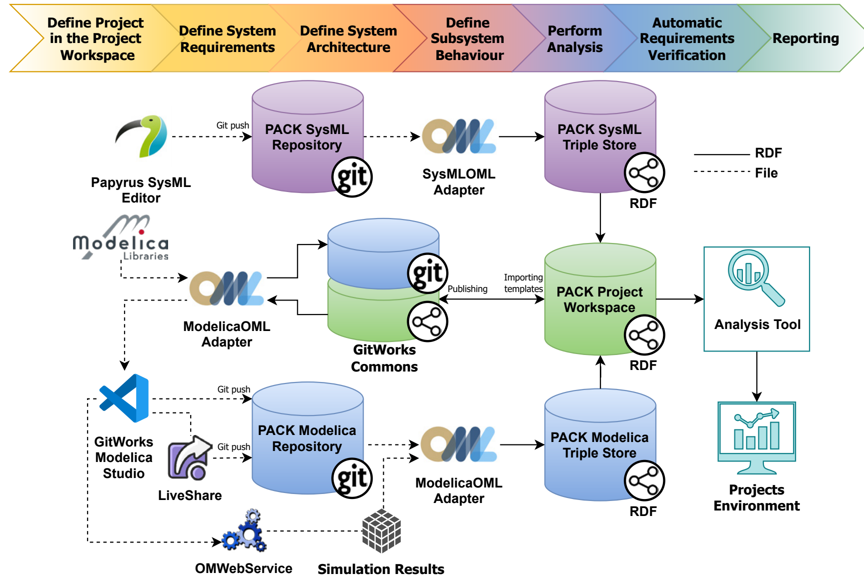


Figure 6. Data flow within the GitWorks platform for the PACK case study.

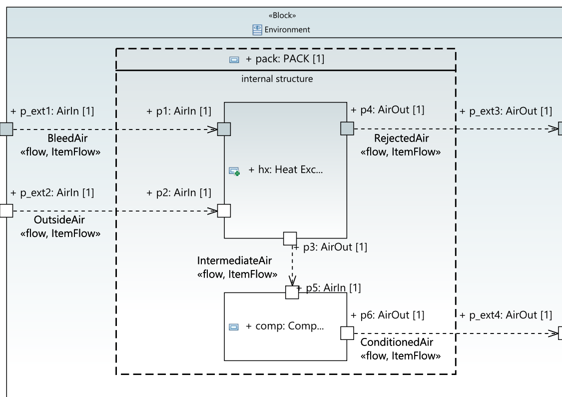
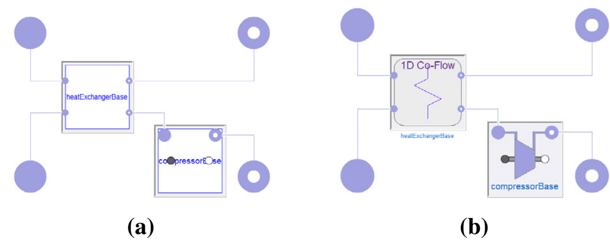


Figure 7. SysML representation of the PACK architecture (hx: Heat Exchanger; comp: Compressor)

```
    mission:presents p3
    mission:presents p4
    PL_Mech:hasMass HXMass]
ci Compressor : mission:Component [
    mission:presents p5
    mission:presents p6
    PL_Mech:hasMass CompMass]
ci PACKMass : PL_Mech:ComponentMass
ci HXMass : PL_Mech:ComponentMass
ci CompMass : PL_Mech:ComponentMass
```

This simple example illustrates how the Semantic Twin can be constructed either directly via the GitWorks Workbench environment (e.g. definition of roles and tasks), or via translation from another artifact (e.g. translation of requirements and architecture from SysML to OML). The Semantic Twin can then be used to efficiently query information about the project and the system (including time-travel queries to investigate evolution of system design) or to automatically generate new modeling artifacts as demonstrated in the next section.



(a) (b)

Figure 8. Stages of PACK Modelica model (a) Partial model, (b) Completed model

4.2 Semantic Twin-Powered Authoring

Direct integration of the centralized RDF databases into GitWorks authoring tools allows the users to translate the architectural and requirements knowledge originally expressed as SysML artifacts into corresponding Modelica partial models through the ModelicaOML adapter on the fly. As such, Figure 8a depicts a partial model generated from the OML representation of the PACK architecture.

The generated partial model can be used as a starting point for the Design Engineers to define the behavior of the PACK's components: the heat exchanger and the compressor. Depending on the desired model fidelity, there are multiple ways of implementing the behavior. For example, the total heat transfer rate of the heat exchanger can be obtained from:

$$Q = E_e C_a (T_{bleed} - T_{ram}) \quad (1)$$

where E_e is the effectiveness of the heat exchanger, $C_{a_{min}}$ is the heat capacity of the air stream and T_{bleed} and T_{ram} are

the temperatures of bleed and ram air respectively (Poudel 2019). The pressure drop in the heat exchanger can be calculated via the following equation:

$$\Delta P = \frac{f L_{tube} \rho v^2}{2 D_h} \quad (2)$$

where f is the friction factor, L_{tube} is the total length of the heat exchanger, ρ is the air density as a function of temperature, v is the mean stream velocity and D_h is the hydraulic diameter (Poudel 2019). Traditionally, Equation 1 and Equation 2 can be implemented manually by engineers in Modelica language. This can be a time-consuming, error-prone and costly process. Alternatively, the engineers can sift through many third-party Modelica repositories available online in the hope of finding the component model they need (Hussain et al. 2022). Existing Modelica tools such as OpenModelica and Dymola (Dempsey 2006) can assist with this task by performing a basic keyword search. However, this comes with a significant limitation of only searching within classes currently loaded into the workspace. Furthermore, such a search is incapable of analyzing the inherent structural and semantic knowledge embedded in the models.

In order to alleviate such modeling bottlenecks and boost model exchange, the GitWorks Commons offers users a convenient interface to query an RDF database of published models and libraries to find required components and blocks. This approach is based on a physics-based simulation ontology currently being developed in OML, and employs the SPARQL query language to offer a set of advanced search techniques such as aggregation, extensible value testing, subqueries, and negation (Hussain et al. 2022; Kollia, Glimm, and Horrocks 2011). For example, a SPARQL query can be designed to find another Modelica component with two compatible connector ports. The connector compatibility is established by ensuring that the connector variables have the same name, prefixes and type. Listing 3 shows an excerpt of such a query which outputs all compatible Modelica models containing two connector ports carrying the thermofluidic variables `m_flow, p, h_outflow, Xi_outflow, C_outflow`. We are searching for a model that has two connectors that contain these variables and the names, types and prefixes of the variables match. For example, Modelica.Fluid (Casella, Otter, et al. 2006) and ThermoPower (Casella and Leva 2005) libraries use distinct but compatible connectors which can only be identified manually or through a SPARQL query. As a result, the users are presented with compatible components from all libraries that use the same connector definition. The result of running the query in Listing 3 is given in Listing 4, and shows the three compressor models from the `PL_Lib` library that are matching. One can note that most Modelica tools support `choicesAllMatching` annotation which can help populate the dialogs with a list of matching models - this feature is limited to loaded libraries only and extending it would require tool changes.

Having the Modelica models expressed as individuals using an OML-based vocabulary and searching these using SPARQL queries against the GitWorks Commons populated with all the available libraries on the GitWorks platform is a paradigm-changing feature.

The GitWorks Commons displays a list of models and libraries obtained as a result of the search query and enables the user to inspect the documentation and the code. As highlighted in (Hussain et al. 2022), it is rare that a component model can be reused without any modifications. Therefore, if a suitable component is found, the engineer can import the selected model into the Modelica Studio workspace and invite the rest of the team to use the real-time collaborative functionality of the VSCode Liveshare extension to modify, complete and check the full model definition synchronously. The resulting model definition is shown in Figure 8b. Optionally, component models can also be seamlessly published in the GitWorks Commons with dependencies tracked through Modelica's `uses` annotation.

Development of Modelica models, therefore, is greatly aided by the proposed Semantic Twin technology through automatic generation of model architecture and facilitating model reuse and exchange within the community.

4.3 Simulation-based Requirements Verification and Reporting

Automating the requirement verification allows engineers to accelerate the iterative systems engineering process. In order to enable this capability, the requirements formally captured by the Semantic Twin in Section 4.1 can be expressed alongside a behavioral model developed in Modelica.

Several Modelica libraries for simulation-based requirements verification exist, such as Modelica_Requirements, ReqSysPro discussed in (Bouskela et al. 2022), and vVDR outlined in (Mengist, Buffoni, and Pop 2021). In this case study, the requirements and scenario are captured in a vVDR-style verification scenario model through the ModelicaOML adapter. The resulting model is shown in Figure 9. The model contains the design block containing the PACK system defined in the previous section. It receives the inputs defined in the scenario block and outputs the calculated values of system mass and conditioned air temperature and pressure. These values are then used as inputs in the three requirements blocks to calculate the verification status of the corresponding requirements.

The simulations are performed using the OMWebService (defined in Section 3.1), and the results can be committed to the relevant Git repository at the same time to preserve the traceability of the results. At the same time, the ModelicaOML adapter is invoked to pass the verification status of requirements to the Semantic Twin. Table 1 shows the verification status of each of the requirements defined in Section 4.1.

As a result, all the systems engineering knowledge

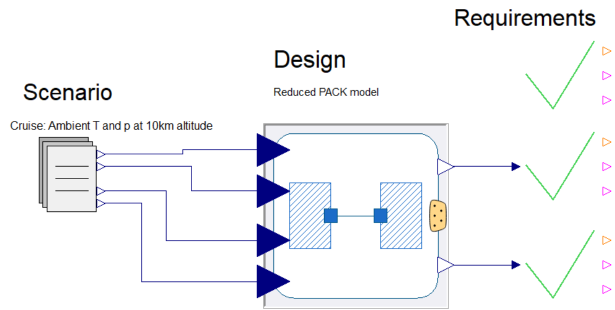


Figure 9. PACK system design model as a part of a verification scenario model.

Table 1. Requirements verification status

	Target	Calculated	Status
Req. 1	100 kg	97 kg	Verified
Req. 2	293 ± 1 K	293.15 K	Verified
Req. 3	101 ± 3 kPA	101.4 kPa	Verified

generated throughout the project is unified and stored in the RDF-based Semantic Twin. This enables automated and intelligent reporting of necessary decision-making information via querying and reasoning across the whole dataset. For example, the Lead Systems Engineer may wish to take a closer look into the project management and requirement verification aspects of the project by constructing a SPARQL query corresponding to the following natural language expression:

- Return all system-level requirements (with their proposed verification test cases) that have not been verified and the person responsible for performing the verification test case.

As another example, the Project Manager may wish to find out more about the model development and trace its metadata information:

- Return the artifact that defines a heat exchanger component which 'presents' a particular interface and the original author of that artifact.

Such queries can be expressed using SPARQL within the Workbench environment, and the outputs can be displayed as tables or graphs. This allows the users to gain insight that would normally be difficult to attain through other means.

The PACK case study presented in this section has demonstrated how the GitWorks platform can be used to effectively integrate Modelica modeling and simulation environments into the systems engineering process to achieve simulation-based system verification from the early stages of the product lifecycle. Artifacts can be automatically generated from other model types (e.g. SysML to Modelica) via the OML adapters. Modelica models can be further defined by searching the GitWorks Commons for relevant and compatible components. Modelica simulation results can be automatically translated into RDF to automatically verify requirements. The resulting integrated Semantic Twin can then be queried. In this way, Modelica models become an invaluable link in the systems engineering chain by providing added value across multiple domains, all while maximizing automation and reducing the effort required.

5 Conclusions and Future Work

We have presented our vision for the GitWorks platform to enable the democratized model-based design and engineering of cyber-physical systems. We have proposed a system architecture for GitWorks and have developed a prototype implementation focused around enabling the use of Modelica in the larger MBSE process. We have conducted a preliminary case study that has demonstrated the use of GitWorks for the federated design and engineering of a passenger air conditioner system.

Plans for future work include further development of OML vocabularies and OML adaptors to increase the number of different modeling paradigms and COTS tools supported by the platform, increase the maturity of the user interface for the web applications, and demonstrate the application to other use cases including satellite systems and composite structures design and fabrication.

Acknowledgments

The work presented here is supported by the Engineering and Physical Sciences Research Council (UK) under the InnovateUK project 97404 and partially supported by the HUBCAP Innovation Action funded by the European Commission's Horizon 2020 Programme under Grant Agreement 872698.

References

Adams, Brian et al. (2020). *Dakota, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 6.13 User's Manual.* Tech. rep. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

Basnet, Sunil et al. (2022). "A decision-making framework for selecting an MBSE language–A case study to ship pilotage". In: *Expert Systems with Applications*, p. 116451.

Bayer, Todd et al. (2021). "Europa Clipper: MBSE Proving Ground". In: *2021 IEEE Aerospace Conference*. IEEE.

Beckmann, B et al. (2016). "Developing the digital manufacturing commons: a national initiative for US manufacturing innovation". In: *Procedia Manufacturing* 5, pp. 182–194.

Bouskela, Daniel et al. (2022-03). "Formal requirements modeling for cyber-physical systems engineering: an integrated solution based on FORM-L and Modelica". en. In: *Requirements Engineering* 27.1, pp. 1–30. ISSN: 0947-3602, 1432-010X. DOI: 10.1007/s00766-021-00359-z. URL: https://link.springer.com/10.1007/s00766-021-00359-z (visited on 2022-05-22).

Carroll, Jeremy J. et al. (2004). “Jena: Implementing the Semantic Web Recommendations”. In: *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters*. WWW Alt. ’04. New York, NY, USA: Association for Computing Machinery, pp. 74–83. ISBN: 1581139128. DOI: 10.1145/1013367.1013381. URL: https://doi.org/10.1145/1013367.1013381.

Casella, Francesco and Alberto Leva (2005). “Object-oriented modelling & simulation of power plants with modelica”. In: *Proceedings of the 44th IEEE Conference on Decision and Control*. IEEE, pp. 7597–7602.

Casella, Francesco, Martin Otter, et al. (2006). “The Modelica Fluid and Media library for modeling of incompressible and compressible thermo-fluid pipe networks”. In: *Proceedings of the 5th international modelica conference*, pp. 631–640.

Dempsey, Mike (2006). “Dymola for multi-engineering modelling and simulation”. In: *2006 IEEE Vehicle Power and Propulsion Conference*. IEEE, pp. 1–6.

Elaasar, Maged et al. (2019). “The case for integrated model centric engineering”. In: *Proceedings of the 10th model-based enterprise summit (MBE 2019). National Institute of Standards and Technology, Gaithersburg, MD*, pp. 9–16.

Elmqvist, Hilding, Martin Malmheden, and Johan Andreasson (2019). “A Web Architecture for Modeling and Simulation”. In: *Proceedings of the 2nd Japanese Modelica Conference, Tokyo, Japan, May 17-18, 2018*. 148. Linköping University Electronic Press, pp. 255–260.

Elmqvist, Hilding, Sven Erik Mattsson, and Martin Otter (1998). “Modelica: The new object-oriented modeling language”. In: *12th European Simulation Multiconference, Manchester, UK*. Vol. 5.

Fritzson, Peter and Vadim Engelson (1998). “Modelica—A unified object-oriented language for system modeling and simulation”. In: *European Conference on Object-Oriented Programming*. Springer, pp. 67–90.

Fritzson, Peter, Adrian Pop, et al. (2020). “The OpenModelica integrated environment for modeling, simulation, and model-based development”. In: *Modeling, Identification and Control* 41.4, pp. 241–295.

Hussain, Mohammad et al. (2022). “Approaches for Simulation Model Reuse in Systems Design—A Review”. In: *SAE Technical Paper* 2022-01-0355. DOI: 10.4271/2022-01-0355.

Isasi, Yago, Ramón Noguerón, and Quirien Wijnands (2015). “Simulation Model Reference Library: A new tool to promote simulation models reusability”. In: *Workshop on Simulation for European Space Programmes (SESP)*. Vol. 24, p. 26.

Jennions, Ian et al. (2020). “Simulation of an aircraft environmental control system”. In: *Applied Thermal Engineering* 172, p. 114925.

Johansson, Olof, Adrian Pop, and Peter Fritzson (2005). “ModelicaDB - A Tool for Searching, Analysing, Crossreferencing and Checking of Modelica Libraries”. In: *Proceedings fo the 4th International Modelica Conference, March 7-8, Hamburg University of Technology, Hamburg-Harburg, Germany, Volume 2 :* Linköping University, The Institute of Technology, pp. 445–454. URL: http://www.modelica.org/events/Conference2005.

Kecskes, Tamas et al. (2019). “Modelica on the Web”. In: *Proceedings of The American Modelica Conference 2018, October 9-10, Somberg Conference Center, Cambridge MA, USA*. 154. Linköping University Electronic Press, pp. 220–226.

Kollia, Ilianna, Birte Glimm, and Ian Horrocks (2011). “SPARQL query answering over OWL ontologies”. In: *Extended Semantic Web Conference*. Springer, pp. 382–396.

Larsen, Peter Gorm et al. (2016). “Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project”. In: *2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS (CPS Data)*. IEEE, pp. 1–6.

Ma, Junda et al. (2022-03). “Systematic Literature Review of MBSE Tool-Chains”. en. In: *Applied Sciences* 12.7, p. 3431. ISSN: 2076-3417. DOI: 10.3390/app12073431. URL: https://www.mdpi.com/2076-3417/12/7/3431 (visited on 2022-05-23).

Madni, Azad and Michael Sievers (2018). “Model-based systems engineering: Motivation, current status, and research opportunities”. In: *Systems Engineering* 21.3, pp. 172–190.

McDermott, Thomas et al. (2020). “Benchmarking the Benefits and Current Maturity of Model-Based Systems Engineering across the Enterprise: Results of the MBSE Maturity Survey”. In: *Technical Report SERC-2020-SR-001*. Systems Engineering Research Center.

Mengist, Alachew, Lena Buffoni, and Adrian Pop (2021). “An Integrated Framework for Traceability and Impact Analysis in Requirements Verification of Cyber–Physical Systems”. In: *Electronics* 10.8, p. 983.

Nachawati, Mohamad Omar and Alexander Brodsky (2021). “Mixed-Integer Constrained Grey-Box Optimization based on Dynamic Surrogate Models and Approximated Interval Analysis”. In: *Proceedings of the 10th International Conference on Operations Research and Enterprise Systems, ICORES 2021, Online Streaming, February 4-6, 2021*. SCITEPRESS, pp. 99–112.

Poudel, Sabin (2019). *Modelling of a Generic Aircraft Environmental Control System in Modelica*.

Rackauckas, Chris et al. (2021). “Composing Modeling and Simulation with Machine Learning in Julia”. In: *Modelica Conferences*, pp. 97–107.

Sirin, Göknur et al. (2015). “A model identity card to support simulation model development process in a collaborative multidisciplinary design environment”. In: *IEEE Systems Journal* 9.4, pp. 1151–1162.

Stirgwolt, Benjamin W, Thomas A Mazzuchi, and Shahram Sarkani (2022). “A model-based systems engineering approach for developing modular system architectures”. In: *Journal of Engineering Design* 33.2, pp. 95–119.

Tennøe, Simen, Geir Halnes, and Gaute T Einevoll (2018). “Uncertainpy: a Python toolbox for uncertainty quantification and sensitivity analysis in computational neuroscience”. In: *Frontiers in neuroinformatics*, p. 49.

Torabzadeh-Tari, Mohsen et al. (2011). “Omweb–virtual web-based remote laboratory for modelica in engineering courses”. In: *Proceedings 8th Modelica Conference, Dresden, Germany*. Vol. 3. Citeseer.

Tudorache, Tania, Jennifer Vendetti, and Natalya Fridman Noy (2008). “Web-Protege: A Lightweight OWL Ontology Editor for the Web.” In: *OWLED*. Vol. 432, p. 2009.

Wagner, David et al. (2020). “CAESAR Model-Based Approach to Harness Design”. In: *2020 IEEE Aerospace Conference*. IEEE, pp. 1–13.

Wan, Li et al. (2013). “A modelica-based modeling, simulation and knowledge sharing web platform”. In: *20th ISPE International Conference on Concurrent Engineering*. IOS Press, pp. 517–525.

A Example SPARQL Query

Listing 3. SPARQL query to find a compatible compressor model

```
PREFIX m:   <http://openmodelica.org/
    openmodelica/modelica#>
PREFIX msl:   <http://examples/
    ModelicaStandardLibrary#>

SELECT DISTINCT ?m ?comp1 ?comp2

WHERE {
  ?m a m:Model .
  ?m m:contains ?comp1 .
  ?m m:contains ?comp2 .

  ?comp1 a m:Component ;
         m:hasType ?con1 .
  ?comp2 a m:Component ;
         m:hasType ?con2 .
  ?con1 a m:Connector . # FlangeA
  ?con1 m:contains [
    a m:Component ;
    m:hasName "m_flow" ;
    m:hasType msl:ThermoPower.Gas.Flange.Medium.
        MassFlowRate
  ] .
  ?con1 m:contains [
    a m:Component ;
    m:hasName "p" ;
    m:hasType msl:ThermoPower.Gas.Flange.Medium.
        AbsolutePressure
  ] .
  ?con1 m:contains [
    a m:Component ;
    m:hasName "h_outflow" ;
    m:hasPrefix "stream"
    m:hasType msl:ThermoPower.Gas.Flange.Medium.
        SpecificEnthalpy
  ] .
  ?con1 m:contains [
    a m:Component ;
    m:hasName "Xi_outflow" ;
    m:hasPrefix "stream"
    m:hasType msl:ThermoPower.Gas.Flange.Medium.
        MassFraction
  ] .
  ?con1 m:contains [
    a m:Component ;
    m:hasName "C_outflow" ;
    m:hasPrefix "stream"
    m:hasType msl:ThermoPower.Gas.Flange.Medium.
        ExtraProperty
  ] .
  ?con2 a m:Connector . # FlangeB
    ...
}
```

Listing 4. The result of running the SPARQL query above

```
{ "head": {
    "vars": [ "m" , "comp1" , "comp2" ] } ,
  "results": {
    "bindings": [
      {
        "m": { "type": "uri" , "value": "http://
            examples/PL_Lib#PL_Lib.Interfaces.
            CompressorBase"},
        "comp1": { "type": "uri" , "value": "
            http://examples/PL_Lib#PL_Lib.
            Interfaces.CompressorBase.inlet"},
        "comp2": { "type": "uri" , "value": "
            http://examples/PL_Lib#PL_Lib.
            Interfaces.CompressorBase.outlet"}},
      {
        "m": { "type": "uri" , "value": "http://
            examples/PL_Lib#PL_Lib.Components.
            Compressor_noMaps"},
        "comp1": { "type": "uri" , "value": "
            http://examples/PL_Lib#PL_Lib.
            Components.Compressor_noMaps.inlet"
            },
        "comp2": { "type": "uri" , "value": "
            http://examples/PL_Lib#PL_Lib.
            Components.Compressor_noMaps.outlet"
            }},
      {
        "m": { "type": "uri" , "value": "http://
            examples/PL_Lib#PL_Lib.Components.
            Compressor_noMaps_mass"},
        "comp1": { "type": "uri" , "value": "
            http://examples/PL_Lib#PL_Lib.
            Components.Compressor_noMaps_mass.
            inlet"},
        "comp2": { "type": "uri" , "value": "
            http://examples/PL_Lib#PL_Lib.
            Components.Compressor_noMaps_mass.
            outlet"}},
    ]
  }
}
```