PROMETHEUS: BRIDGING ACCESSIBILITY AND FLEXIBILITY IN DEVS

Curtis Winstanley[a,b], Gabriel Wainer[a] and Iryna Borshchova [a,b]

[a] Dept. of Systems and Computer Eng., Carleton University, Ottawa, ON, Canada
[b] Flight Research Laboratory, National Research Council Canada, Ottawa, ON, Canada

ABSTRACT

The practice of Discrete Event System Specification (DEVS) modeling is widely used in the modelling and simulation (M&S) field to support event-driven system architectures. By defining states and events, DEVS enables an efficient and cost-effective analysis of systems, allowing practitioners to anticipate potential issues and respond to unexpected scenarios. However, this advantage is not without its challenges as there is an inherent complexity with this practice that can inhibit the adoption of this technology. This paper proposes an architecture implemented in a graphical modelling tool that can address challenges met by DEVS practitioners at varying skill levels. The design of our architecture is aimed at striking an optimal balance between accessibility for new practitioners and the flexibility demanded by DEVS experts. Through this approach, we ensure a comprehensive environment that lowers the entry barrier to DEVS, but also empowers seasoned practitioners to leverage extensive customization capabilities.

Keywords: PROMETHeUS, DEVS, GUI, architecture.

1 INTRODUCTION

The application of M&S is rapidly growing in various industries such as aerospace, business, healthcare, manufacturing, and many others. In 2023, the simulation software market was estimated to be valued at 13.1 billion USD and this value is projected to increase to 32 billion USD by 2032[1]. M&S facilitates the optimization of real-world processes and systems in a virtual environment before a prototype is created. This is especially relevant in defense or manufacturing industries as prototyping can become expensive and time-consuming. Since M&S offers invaluable insight to users which saves time, effort, and ultimately cost, it is unsurprising that the adoption and valuation of M&S software is increasing at a considerably high rate.

Among many methodologies in the M&S field, the DEVS formalism [2] stands out as a widely adopted approach for modeling and simulation of discrete-event systems. DEVS is a framework designed to provide users with a hierarchical, modular and scalable way to design systems and run simulations. It allows for the complexity of systems to be broken down into atomic models. The reverse is also possible where atomic models or even coupled models can be combined into more complex coupled components. This makes it a very powerful tool when used in the field of modeling and simulation.

However, this versatility comes with a level of complexity consistent with the broader challenges in M&S adoption. The authors in [3] do a comprehensive and quantitative literature review of the barriers when adopting M&S technologies in the construction industry. Out of 78 papers used in the analysis, they find that some of the most frequently reported reasons over the past 19 years are: complexity of simulation methodologies, special skills required to develop simulation models and sophisticated nature of simulation outputs. Also, [4] mentions that restraining factors that hinder the adoption of M&S technologies are that it requires expertise to understand modelling algorithms and operate tools effectively. The learning curve can deter practitioners without specialized training from adopting simulation methodologies.

To alleviate difficulties when using DEVS, there are graphical tools available that can help practitioners get started with simulation. These graphical tools offer a graphical user interface(s) (GUI) to assist in the

Proc. of the 2025 Annual Simulation Conference (ANNSIM'25),
May 26-29, 2025, Universidad Complutense de Madrid, Madrid, Spain
J. L. Risco-Martín, G. Rabadi, D. Cetinkaya, R. Cárdenas, S. Ferrero-Losada, and A. Bany Abdelnabi, eds.

process of DEVS modelling by allowing the user to interact with comprehensible diagrams and elements, ultimately making the learning curve easier to navigate. By automating coding in the background, it allows the user to focus more on the design and analysis of their simulations.

While a GUI for DEVS can make system design easier for the user by providing intuitive diagrams and clear workflows, it comes with the disadvantage through a loss of control. Since the GUI is generally developed by someone unknown to an end user, generally the user would adhere to any configurations dictated by the developer. This will hinder the level of customization that the user could otherwise have and potentially make a GUI meaningless depending on the application.

Considering these issues, we define an architecture that is designed to support a specific level of generality, ensuring sufficient flexibility to overcome the constraints outlined above, while striving to preserve usability. As part of this effort, we introduce a graphical tool titled PROMETHeUS (Platform for Realtime Operations, Modeling, Exploration, and Testing of Hierarchical Units and Systems). The purpose is to make DEVS accessible to beginners with an intermediate understanding of software, providing a user-friendly introduction to the field. Additionally, PROMETHeUS will be flexible enough for users to have control over configurations to allow experienced DEVS practitioners more freedom over their applications.

The following sections are organized as such: Section 2 explores other work related to the work done in this paper that may provide more insight into the methodology presented. Section 3 introduces the architecture that was designed for use in PROMETHeUS, describes each part and what role it plays to achieve our requirements. Section 4 provides a case study that demonstrates how an experienced DEVS practitioner could use PROMETHeUS to implement their own simulator through a series of defined configuration steps. The simulator chosen to be implemented for the case study is Cadmium2. Section 5 concludes the paper discussing the future scope of this research for PROMETHeUS.

2 RELATED WORKS

The authors in [5] establish a set of best practices for design systems. These best practices consider finding a balance between the coherence and responsiveness of a design system. Coherence ensures consistency across the system and focuses on maintaining a unified, coherent overall design. If there is a lack of a systematic design approach, inconsistencies can occur where custom solutions are applied to various parts of an application causing it to become fragmented. The adoption of a systematic design can be very powerful as there is a standardized approach to creating components of the system preventing these inconsistencies. On the other hand, responsiveness addresses the system's ability to adapt to users' needs which in many applications can evolve. This raises the concern that coherence and responsiveness are equal and opposite as one cannot be improved without compromising the other. This balance is particularly relevant when developing M&S tools for DEVS, as it is critical to consider structured yet adaptable solutions to accommodate a variety of users.

The work in [6] introduces a tool JDEVS that is a DEVS based modelling framework with a java-based simulation kernel intended for use for environmental modeling. The architecture of the framework allows the user to design DEVS models in a graphical setting, save them in a .xml format to a model library and visualize the simulation. When running a simulation, the DEVS semantics and model structure saved in the .xml files within the model library are converted to Java code and executed with the simulation kernel.

A similar architecture was demonstrated in [7] where they implemented a model builder tool titled *CD++ Builder* with Eclipse and uses the CD++ DEVS simulator. The architecture that was implemented allows the user to create models in Eclipse GUI and save them in a model library. The models can be compiled and simulations run with the CD++ simulator kernel. The simulations' output log files that are parsed, processed and used to deliver visualization to the user on their simulation results.

DEVSML Studio [8] is a DEVS modelling tool created using Eclipse with its architecture rooted in meta-modeling concepts [9]. The tool can import models by importing meta-models into the Eclipse framework. Its architecture is highly modular, and it uses plugins for customization. The base DEVSML Studio uses plugins for the simulator kernels allowing users a choice of what simulator their application uses. Additionally, visualization modules are available through plugins for enhanced user experience.

The intention of this research is to use PROMETHeUS to advance the work done on a Supervisory Controller for NRC's ASRA – Bell 412 [10]. The supervisory controller was developed using DEVS with the Cadmium [11] framework and plays a critical role in orchestrating the aircraft's autonomy. Its complexity has grown rapidly in recent years which has driven a desire to create a user friendly but flexible GUI for building DEVS models, validating, and visualizing them. This would allow users unfamiliar with DEVS to contribute to the development of the supervisory controller. Moreover, PROMETHeUS would support configuration management and code generation within Cadmium, not only for the supervisory controller but also for other simulators, ensuring extensibility for future projects.

We leverage the concept of coherence vs responsiveness from [5] as a basis for PROMETHeUS' architecture for supporting a specific level of generality. The architecture for the tools shown in [6] and [7] are useful when we implement the visualization aspects of PROMETHeUS' architecture, but we expand and allow the user to visualize the execution of the simulations in real time. We additionally take inspiration from [8] to allow experienced DEVS practitioners to add their own desired configurations tailored to their simulators. This enables our tool to support multiple DEVS frameworks implemented in any programming language, rather than restricting users to a specific language or simulator. However, unlike plugin-based approaches, PROMETHeUS is offered as a standalone application, eliminating the complexity of setting up plugins that can deter users new to DEVS.

3 ARCHITECTURE OVERVIEW OF PROMETHEUS

Drawing from the concept of coherence and responsiveness discussed in [5], we outline the requirements for the architecture of PROMETHeUS. In our case, the coherence of the systems' architecture is determined by the necessary level of customization for M&S applications. Requirements for the system's architecture related to responsiveness include the graphical visualization components and a well-defined workflow. We need to design the architecture with a balance such that there is a level of generality to achieve requirements but not too generic where it becomes too difficult for a user new to DEVS to navigate and use the tool.

In PROMETHeUS, the customization available for the user is based on dependency control, simulator selection, and adding configurations for custom simulators. Dependency control and simulator selection are requirements that will impact users new to DEVS. The assumption made is that new users initially will not be interested in adding any required configurations for their own simulator. Getting started with the available simulators in PROMETHeUS is a good starting point for users when learning DEVS. The addition of a custom simulator should be reserved for expert DEVS practitioners as the configuration steps are intricate as demonstrated in the case study later.

With regards to the usability of PROMETHeUS, we need graphical and visualization elements for the following tasks: model building, model testing, setting asynchronous inputs, and real time simulation visualization. For creating DEVS atomic models and coupled models in a graphical format, we use an atomic model builder and a coupled model builder. To validate the models created by the model builders, we use a graphical interface for the user to view their model, design test cases, and run simulations. For some real time simulation applications, the user needs to have the ability to configure asynchronous inputs to their model and we use a graphical interface that is used to achieve this. When running a simulation in real time, we have a window that the user can view each model through a tree hierarchy, and for each atomic model they can view the data received/sent by the ports as well as the current state of the model.

Given the requirements outlined above, we bring it together to define the architecture of PROMETHeUS which is shown in Figure 1. In Figure 1 the color scheme is as follows: gray blocks represent data caches, magenta blocks represent configuration elements, blue blocks represent interactive graphical/visualization component, and orange blocks correspond to validation and simulation functionalities. Additionally, any blocks with a diagonal pattern suggest that the block is optional/conditional depending on the context.

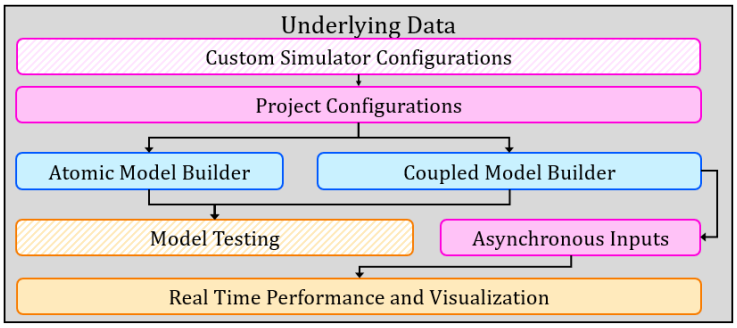


Figure 1: Block diagram of the architecture of PROMETHeUS.

Custom Simulator Configurations is designed for advanced users who want to add their own simulator to PROMETHeUS. These features provide flexibility and customization but are not intended for beginners. Assuming a user is new to DEVS, they would start with *Project Configurations* which is where the user can customize aspects of the project with regards to the software. From there the user can create DEVS models using either the *Atomic Model Builder* or the *Coupled Model Builder*. Either builder can then go to *Model Testing* for users to validate and simulate their models. For coupled models, the process can further extend to the *Asynchronous Inputs* module, which then leads to *Real Time Performance and Visualization*. All these blocks are on top of the *Underlying Data* block which serves as the foundation for each previously mentioned block. In section 3, we will explore the functionality in depth of each block of the architecture.

3.1 Architecture Component: Underlying Data

We need to share common information across all modules of the PROMETHeUS architecture. This will ensure that as a user operates PROMETHeUS, the information is consistent throughout the application. To do this, we implement a centralized system for managing the underlying data which comes either in the form of serialized metadata or code. The metadata records details such as project settings, model structures, or simulation parameters and the code is what PROMETHeUS generates for executing simulations. Figure 2 provides a diagram of the *Underlying Data* of the PROMETHeUS architecture.

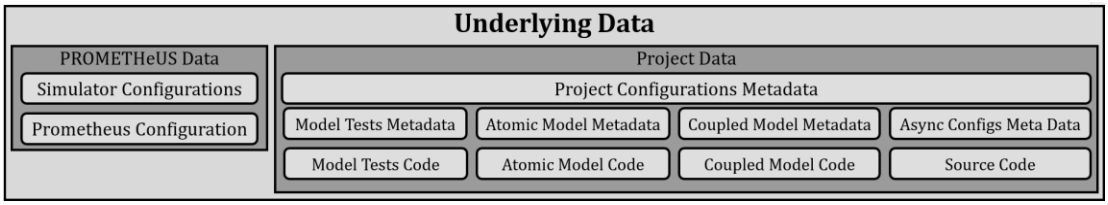


Figure 2: Structure of the Underlying Data block of the PROMETHeUS architecture.

Within the *Underlying Data* block there are two categories: *PROMETHeUS Data* and *Project Data*. *PROMETHeUS Data* refers to the information used for the PROMETHeUS application. This includes settings that control the behavior of the application itself, such as operational parameters, as well as configurations for any simulators available to the user. This data enables PROMETHeUS to determine how it should interact with a users' project(s) by preforming reading or writing operations to the *Project Data*. *Project Data* is specific to the project and managed by the application. This data encompasses all the resources necessary for the development and execution of the project. Throughout the development of a

project, the user creates atomic or coupled models, tests, and asynchronous inputs configurations using a GUI, which records the meta data and generating code derived from that metadata.

The GUI contributes to the global project configuration metadata, a repository that tracks information necessary for various modules within the architectural framework. This allows any module of the architecture accessibility to critical information in a centralized manner. Furthermore, it improves scalability and integration of new modules into the PROMETHeUS architecture for future development.

3.2 Architecture Component: Project Configurations

As previously mentioned at the beginning of Section 3, this part of the architecture would be the start of the workflow for a user new to M&S. This was considered when designing the *Project Configurations* block as we need it to be as simple as possible. A step-by-step workflow for *Project Configurations* is outlined in Figure 3.

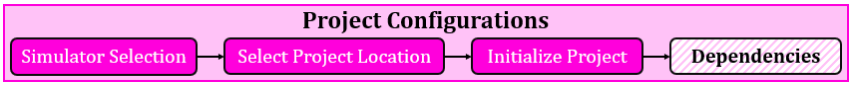


Figure 3: Structure of the Project Configurations block in the PROMETHeUS architecture.

A user would begin by selecting a simulator from a list of available simulators. What constitutes an available simulator is a set of simulator specific project configurations, scripts and settings for a specific simulator (Ex. Cadmium2, PyDEVS, MS4ME) available to the user. If a simulator is not available, the user can add these configurations for their own simulator which is discussed in more detail in Section 3.7.

Once a simulator is selected, the user must select where the project should be located. This was done so that users can do version control with their PROMETHeUS projects using software project management such as GitHub [12]. Additionally, it allows users to make any number of PROMETHeUS projects on their local machine and switching between them is simply switching the project location.

The project must be initialized before the user can proceed with the application. This initialization is a simple button press but, in the background, two things are happening. The first is that the project directory tree and any required files are populated in the project location the user selected. The second is that any required CLI (Command Line Interface) commands are executed and, in these commands, includes the download of the selected simulator itself into the project. This process is explained more in Section 3.7

The final and optional part of *Project Configurations* is dependency management. There may be cases where a user wants to include dependencies in their projects such as external software libraries for additional flexibility in their application. It should be noted the user can come back to this at any time, but the project must be first initialized.

3.3 Architecture Component: Model Builders

Model creation can be one of the challenging aspects for users learning DEVS as the user must understand the DEVS formalism and how the DEVS formalism semantics are transcribed into the software of their simulator. That can become overwhelming and can in turn repulse practitioners interested in M&S from utilizing it.

We provide interactive graphical creation of models in PROMETHeUS through an *Atomic Model Builder* and a *Coupled Model Builder*. As shown in Figure 1, the model builders are only accessible once the user has completed the *Project Configurations* block discussed in the previous section. The *Atomic Model Builder* is an interactive graphical interface that allows for the creation of atomic models through graphical elements such as states, external transitions, internal transitions, input ports, output ports, state variables and code boxes. Similarly, the *Coupled Model Builder* is an interactive graphical interface that allows both atomic and coupled models to be coupled together using graphical elements.

3.4 Architecture Component: Model Testing

DEVS is a framework designed to provide users with a hierarchical and modular way to design systems and run simulations [2] hence, DEVS simplifies the process of testing and validating models. In PROMETHeUS, when models are created using either atomic or coupled, users have the option to test, simulate, and validate their models. Figure 4 shows the workflow for testing models in PROMETHeUS.

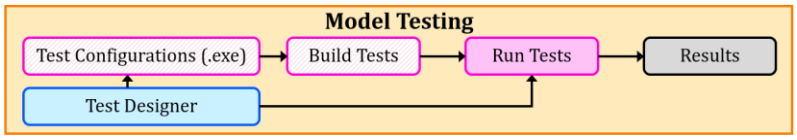


Figure 4: Structure of the Model Testing block in the PROMETHeUS architecture.

The first part of model testing is designing the tests the user wants to execute. We offer a graphical interface where the user can set up testing aspects such as inputs to generate, outputs to expect, and states transitions to expect. Additionally, the user can design any number of test cases before moving onto the next blocks. When the user is designing tests, the code for the tests is automatically generated in the background.

Once users have designed tests to their satisfaction and are prepared to proceed with simulation, there are two blocks we can proceed to depending on the context. If the language of the simulator used by a PROMETHeUS project is a mid-level coding language that requires compilation such as C++ or Java, we proceed to the *Test Configurations* block. Otherwise, if the simulator's native language is a high-level language that does not require compilation, we proceed to the *Run Tests* block.

Assuming we go to the *Test Configurations* block, the user needs to consider configurations related to compiling/building the code that was generated by the *Test Designer*. This block provides the user with the opportunity to add their tests to a build system to manage the executable of the tests. It also allows for defining compilation constraints, dependencies, or execution parameters. When the configurations are set, we move to the *Build Tests* block whereby the click of a button, build commands are executed to produce the executable. Once built we can run the software by clicking a button which will run commands to execute the binary that was compiled.

For simulators where their native language does not require code compilation for running tests, we would simply run the code by clicking a button hence skipping any compilation steps. Either way, when the tests are run the log files are visible for users to see and validate the models' behavior.

3.5 Architecture Component: Asynchronous Inputs

Asynchronous inputs are critical in real time applications as they allow for a system to dynamically react to inputs or external events that are non-deterministic and arrive at unpredictable times. This is particularly crucial when you may have a system with sensors or components that propagate data over a network for consumption or processing. In the context of DEVS real time applications where the external data is often integral to real-time simulations, the need for asynchronous inputs becomes even more pronounced. PROMETHeUS lets users configure asynchronous inputs into their projects' coupled models if they choose to reap the benefits described above. Figure 5 shows the block diagram for the workflow of the *Asynchronous Inputs* block in the PROMETHeUS architecture.

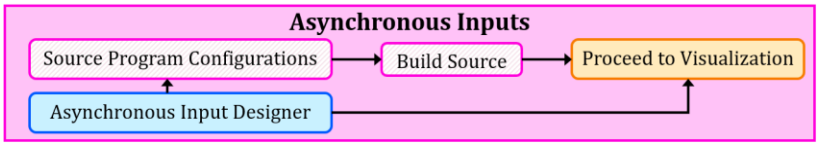


Figure 5: Structure of the Asynchronous Input block in the PROMETHeUS architecture.

The overall structure of the *Asynchronous Inputs* block is like the *Model Testing* block in Section 3.4. The primary difference is that we consider *Asynchronous Inputs* as a configuration block and not a validation block, hence defined magenta color. This is because this step is required before moving on to the *Real Time Simulation Visualization* block of the PROMETHeUS architecture which itself is considered as the validation block for real-time simulation.

A user would initially begin by using the graphical interface for configuration of the asynchronous inputs. As the user is interacting with the *Asynchronous Input Designer*, code is automatically generated in the background. Like the *Model Testing* block, if the language of the simulator is one that requires compilation, the user would have to set the source code configurations and build the code generated by the *Asynchronous Input Designer*. Otherwise, if the native simulator language is one that does not require compilation, the user can advance to real time simulation visualization for validating their application.

3.6 Architecture Component: Real Time Performance and Visualization

Being able to visualize the real time state of the real time applications significantly simplifies the validation process. It provides the ability to observe their application's behavior throughout different tasks and most importantly to identify when it breaks or deviate from its intended behavior. Traditionally, if there were faults in an application, the application itself would have log files describing its internal state which would be processed after execution in a visualization application. A user would then have to play back the data through the application to gain some insight into where the fault is sourced. PROMETHeUS offers users the ability to view the state of their DEVS applications in real time. The structure of the Real Time Performance and Visualization block from Figure 1 is shown in Figure 6.

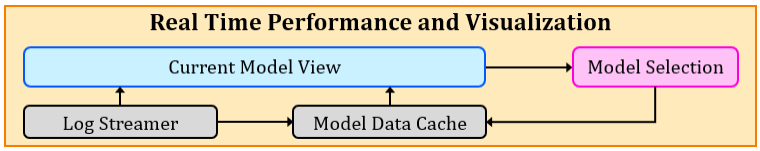


Figure 6: Structure of Real Time Performance and Visualization in the PROMETHeUS architecture.

When we run an application in real time, we have a graphical interface that defaults to the view of the top-level or otherwise tree root model of the application. The concept behind the *Current Model View* block is that in the coupled model case, the user can select any child model within the model that is already displayed. In the atomic model case, the user will be able to visually see the state of that model and any data going in and out of the model. For both atomic and coupled models, the user can navigate up the model hierarchy to the view of the parent model, enabling the ability for full exploration of the simulations' model tree.

For tree exploration of the application to be successful, we construct a *Model Data Cache* before the application begins. The *Model Data Cache* itself is a tree data structure that mimics the structure of a applications' model tree. Each model data structure in the *Model Data Cache* has information about the model type, name, ports, child models, model state, and input/output information. Static data like model type, name, and ports are populated before the simulation begins. Dynamic data such as the state of the model or input/output information are populated at runtime.

For populating the dynamic data in the *Model Data Cache*, we use a *Log Streamer* that streams data throughout the execution of the application into that *Model Data Cache*. The *Log Streamer* is a thread that streams every new line from the log file, parses it, and populates the data cache. Additionally, the *Log Streamer* will forward the same parsed data meant for the *Model Data Cache* to the *Current Model View* which is how we get real time updates on the applications' execution. The purpose of storing the same data in the *Model Data Cache* instead of solely sending it to the *Current Model View* is to ensure continuity.

When the user switches models, we need to be able to update the newly rendered model with the last known state and input/output data.

3.7 Architecture Component: Custom Simulator Configurations

The *Custom Simulator Configurations* block of the PROMETHeUS architecture is not recommended to a novice M&S practitioner. Only advanced DEVS practitioners should use it so as to be able to configure a custom simulator for use in PROMETHeUS. It requires knowledge to understand semantics between the DEVS formalism and the simulator. This architecture block is left to the end because it provides a good segway into the case study in Section 4.

One of the goals of PROMETHeUS is to serve as a platform for anyone who wants to integrate their own simulator while benefiting from its existing graphical features. We want to provide sufficient generality in the program so that the user is not subject to our dictated simulators or configurations. For this to be successful, we have produced a set of configurations that will allow any user to use PROMETHeUS with the ability to use any simulator that they desire. The structure of the *Custom Simulator Configurations* block is shown in Figure 7.

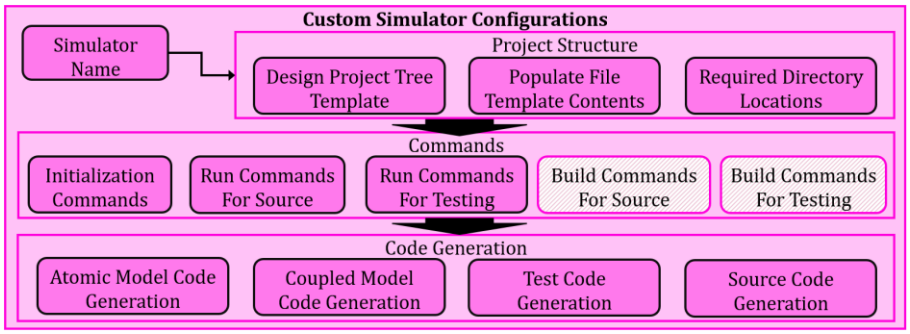


Figure 7: Structure of the Custom Simulator Configurations in the PROMETHeUS architecture.

The *Custom Simulator Configurations* block defines the workflow that a user should follow when making configurations for their own simulator. Once the user has named their simulator, they must consider the project structure for the simulator. At its core, a PROMETHeUS project is like any other software project and each simulator must have a specified project structure. The first part of the project structure is designing the project tree structure which is a tree of directories and files. In Section 3.2, when a user initializes a project with this simulator, the project tree specified by this configuration will be created wherever the user sets the project's location. Assuming there are files within the project tree, we have the option to populate them with their own template contents. This is useful for files that are related to organizing dependencies such as *gitsubmodule* files so that the user does not have to handle this every time they start a project with this simulator. Finally, the final part of the *Project Structure* block is specifying the different directory locations required by PROMETHeUS. This is critical, as this is how PROMETHeUS interfaces between the underlying data and the graphical interfaces. A user will need to specify where they want code and metadata stored as well as locations of any essential files.

The next configurations that must be considered are the *Commands*. While using PROMETHeUS, there are parts where commands need to be executed. *Initialization Commands* are executed when the user initializes their project, and it gives the user the opportunity to run configurations for their build system. During real-time simulation and visualization, PROMETHeUS executes commands specified under *Run Commands for Source*, which are used by the *Asynchronous Inputs* block detailed in Section 3.5. Similarly, for the *Model Testing* block outlined in Section 3.4, commands required to execute the tests are defined under *Run Commands for Testing*. The final two blocks in *Commands* are the *Build Commands for Source* and *Build Commands for Testing*. Similarly to what was described in 3.4 and 3.5, if the simulator being used was

developed where its native language does not require compilation, it should be skipped. Otherwise, the user must specify how the code we generated by the *Model Testing* and *Asynchronous Inputs* block is compiled.

The final and most labor-intensive part of *Custom Simulator Configurations* is the *Code Generation*. As previously mentioned, the *Atomic Model Builder*, *Coupled Model Builder*, *Test Designer* and *Asynchronous Inputs Designer* all generate code in the background when in use. This code is generated using functions in python [13] files that take in the information populated within the graphical interfaces as arguments for these functions. Each block in *Code Generation* represents a function that needs to be complete for each designer to appropriately generate the code for use by the configured simulator. The *Atomic Model Code Generation* is used by the *Atomic Model Builder*, *Coupled Model Code Generation* by the *Coupled Model Builder*, *Test Code Generation* by the *Test Designer*, and *Source Code Generation* by the *Asynchronous Inputs Designer*. To assist the user when writing scripts, the function template and docstring have already been constructed.

When all the previous parts of the simulator configurations have been completed, any user will be able to select the created simulator for a PROMETHeUS project. It should be noted that if updates are made to any simulator, the user can go into the *Custom Simulator Configurations* for the specific simulator and make changes to any of the configurations if needed.

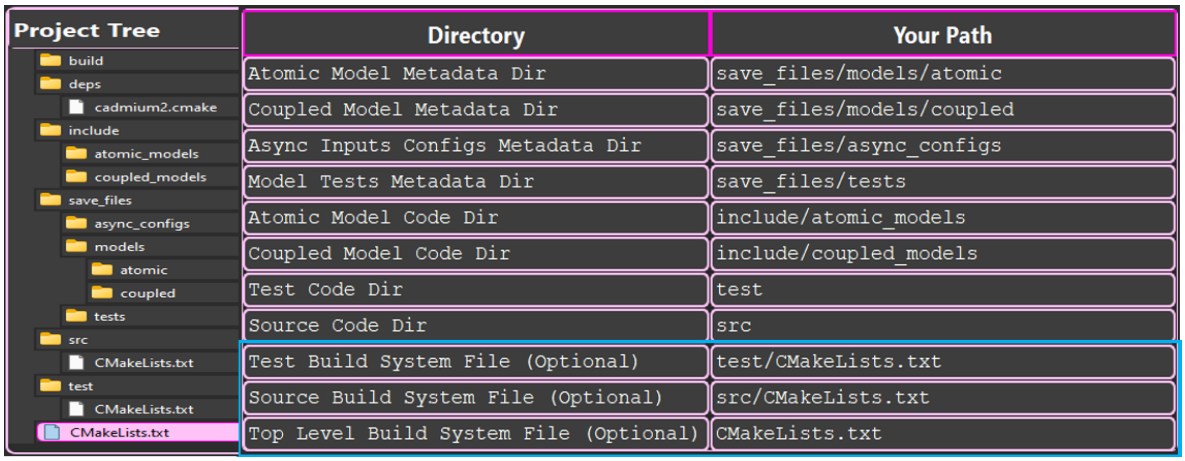
4 CASE STUDY: ADDING CADMIUM IN PROMETHEUS

In this section we introduce a case study that demonstrates how we add our own simulator into PROMETHeUS. More specifically, we will add the second version of the Cadmium simulator, i.e., Cadmium2, which is a C++ library that implements the DEVS formalism [11]. During the case study, we will follow the structure shown in Figure 8 to add the Cadmium2 simulator to PROMETHeUS. The goal of this section is to serve as a practical guide for anyone seeking to add a simulator to the platform. Additionally, we want to demonstrate that when any simulator of any language is integrated and configured into PROMETHeUS, any user will be able to utilize it.

4.1 Project Structure

When we open the PROMETHeUS Simulator Configurations window and name the simulator (in our case *Cadmium2* is the simulator name) we are first met with the task of configuring the project structure. We have an interactive menu like a file explorer for creating the template project directories and files. Additionally, we provide a tabular view to specify and manage the directory paths required for PROMETHeUS' operation. Figure 8 illustrates the *Cadmium2* project tree template on the left, while the right showcases the directory locations for the project tree as displayed in the PROMETHeUS application.

The project tree template we set in Figure 8 is a set of directories and files for any *Cadmium2* project created in PROMETHeUS. It is important to note that the only files we added to the template are *CMake* files. Since *Cadmium2* is a C++ based simulator, we choose to use the *CMake* build system for the lower-level project configurations. The template features a top-level *CMakeLists* file and additional *CMakeLists* files located in the *src* and *test* folders. This is to handle specific build configurations for the *Model Testing* block shown in Figure 4 and *Asynchronous Inputs* block shown in Figure 5. The *cadmium2.cmake* file is a file that contains a command to download the Cadmium2 simulator as a dependency.



Directory	Your Path
Atomic Model Metadata Dir	save_files/models/atomic
Coupled Model Metadata Dir	save_files/models/coupled
Async Inputs Configs Metadata Dir	save_files/async_configs
Model Tests Metadata Dir	save_files/tests
Atomic Model Code Dir	include/atomic_models
Coupled Model Code Dir	include/coupled_models
Test Code Dir	test
Source Code Dir	src
Test Build System File (Optional)	test/CMakeLists.txt
Source Build System File (Optional)	src/CMakeLists.txt
Top Level Build System File (Optional)	CMakeLists.txt

Figure 8: Cadmium2 project template in PROMETHeUS.

To utilize the project tree in a project and for PROMETHeUS to function, we need to specify the directory locations. The first column of the table presented in Figure 8 contains descriptions of the directories required by PROMETHeUS. The directories required correspond to where PROMETHeUS will read/write the meta data and code within the project tree template. Furthermore, these directories are what make up the *Underlying Project Data* block withing the *Underlying Data* shown in Figure 2. In the right column of the table, all directories specified must be relative to the top-level root of the project tree. As an example from the table, we specify that directory where all atomic model meta data should be stored (*Atomic Model Meta Data Dir*) in *save_files/model/atomic*. In a practical case, if the user has a PROMETHeUS project using Cadmium2 at *C:/my/project*, then this directory will be dynamically transcribed by PROMETHeUS as *C:/my/project/save_files/model/atomic* during run time.

The optional file paths circled in blue in Figure 8 are the build system specific files. These are required under the condition that the simulator programming language is compilation based. As Cadmium 2 is C++ and therefore requires code to be compiled for execution, we specify the build system files for the top level, test and source to match our project tree. The *Test Build System File* is where the test configurations in the *Model Testing* block of Figure 4 are written. Similarly, the *Source Build System File* is where source program configurations from the *Asynchronous Inputs* block of Figure 5 are written. Since build systems are generally hierarchical, we also require a top-level build system file to serve as the root over the source and test build system files. In the case of *Cadmium2*, we use the top-level build system file for dependency management which includes handling the *Cadmium2* simulator as a dependency.

To finalize our configuration of the project structure is populating any files with templated contents. For this, we populate the top-level *CMakeLists* with code for handling the *Cadmium2* simulator dependency, as well as linking the two lower-level *CMakeLists* files. For the sake of space, the templated contents of the top-level *CMakeLists* and *cadmium2.cmake* are shown in the Appendix.

4.2 Commands

This section addresses the set of commands that PROMETHeUS will use for different parts of the program. The required commands that we must provide PROMETHeUS are the initialization commands and the run commands. Once again, the build commands are required under the condition that the simulators' programming language requires compilation which *Cadmium2* does. As a demonstration of how we provide PROMETHeUS commands to use, Figure 9 shows the commands for building tests. Note that the format is the same for the other command sets hence shown in the Appendix.

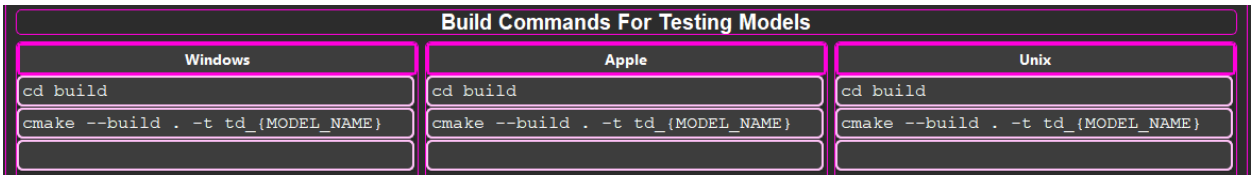


Figure 9: Cadmium2 commands template in PROMETHeUS.

First, PROMETHeUS accounts for the widely used platforms (Windows, Apple/MacOS, and Unix) and we allowed users to specify the commands per those platforms. For our case, the commands for building are the same for each platform. When building tests for *Cadmium2*, we execute two commands consecutively: *cd build* and *cmake --build . -t td_{MODEL_NAME}*. These commands change the current working directory to the build folder of the project and run a command to build a *CMake* executable.

In PROMETHeUS, we assume that the default current working directory is the user's current top-level directory of a project, as in the project directory template shown in Figure 8. Another feature we have when configuring a simulators' commands are keywords. The keyword *{MODEL_NAME}* used above evaluates to the model's name we are doing model testing. As an example, if we are testing a model *MyExampleModel*, then at runtime PROMETHeUS would execute *cmake --build . -t td_MyExampleModel*.

4.3 Code Generation

The final part of the *Cadmium2* configurations into PROMETHeUS is the code generation. For this, we need to complete a set of python [13] functions with the parameters in the provided function definitions. The goal of this is to create code fit for use with the *Cadmium2* simulator. The four functions we must complete are: *generate_atomic_model_code*, *generate_coupled_model_code*, *generate_tests*, and *generate_source*. These functions correspond to the *Atomic Model Code Generation*, *Coupled Model Code Generation*, *Test Code Generation*, and *Source Code Generation* blocks in Figure 7 respectively. In PROMETHeUS' Simulator Configurations window, we can complete each function through a set of editors. The process for completing these functions is identical, so due to space constraints, we will use *generate_tests* as a demonstration. The *generate_tests* function is shown in Figure 10.

```
def generate_tests(_input_port_data: dict,
                   _output_port_data: dict,
                   _constructor_data: dict,
                   _model_name: str,
                   _location_of_model: str,
                   _test_data: dict) -> dict:

    test_driver_code = write_test_driver(_model_name)
    test_data_code = write_test_data(_model_name, _test_data)
    top_coupled_code = write_top_coupled(_input_port_data, _output_port_data, _constructor_data, _model_name, _location_of_model)
    variant_goblin_code = write_variant_goblin(_model_name, _input_port_data, _output_port_data, _constructor_data, _location_of_model)

    code_dict = {"td_" + _model_name + ".cpp": test_driver_code,
                 "test_data.hpp": test_data_code,
                 "top_coupled_" + _model_name + ".hpp": top_coupled_code,
                 _model_name + "_Variant_Goblin.hpp": variant_goblin_code}

    return code_dict
```

Figure 10: *generate_tests* function for *Cadmium2* in a PROMETHeUS text editor.

The first requirement when completing the required functions is to understand the meaning and contents of the parameters. Our example in Figure 10 shows the *generate_tests* function in the blue square. Parameters for the code generation functions can correspond to one or more graphical elements within the graphical interfaces and any preprocessed information that may be useful to the user. The specific data within these parameters is described in detail within the PROMETHeUS manual provided in the Appendix.

Once the parameters are understood, we must write the python [13] code for generating our tests represented by the contents in the green square of Figure 11. For *Cadmium2* we follow the testing framework proposed in [14] hence why we have the functions *write_test_driver*, *write_test_data*, *write_top_coupled*, and *write_variant_goblin*. These functions are passed a subset of parameters of *generate_tests* and they all return a string of code which is written to their own respective files.

After we have all the code in a string format that we need, we must consider how it gets written into files for execution. This logic lies in how we return from our code generation functions highlighted by the red square from our example in Figure 11. The data structure that we return is a dictionary where the keys are the filenames, and the value is the associated code string. For example, the *test_driver_code* is written to the filename *"td_"* + *_model_name* + *".cpp"*. This file is located where we specified the *test code dir* in Figure 8, within a subdirectory named after the model. To fully demonstrate this, if we have a model named *MyExampleModel*, then our *test_driver_code* string will be written to the location {*PROJECT_DIR*}*/test/MyExampleModel/td_MyExampleModel.cpp*.

5 CONCLUSIONS

We discussed some limitations that can be a barrier for new practitioners to get involved with M&S, particularly with DEVS. To remedy these limitations, we designed an architecture implemented in a DEVS tool called PROMETHeUS, aimed at catering to both beginners and experienced practitioners. Each major part of the architecture was examined, their purpose described, and functionality explained to achieve our goal. For beginners, they can choose from the existing simulators configured in the tool and utilize the graphical interfaces and for learning DEVS. On the other hand, advanced DEVS practitioners can add their own simulator to the project through a documented configuration workflow.

To demonstrate this, the *Cadmium2* simulator built with C++ was configured for use in PROMETHeUS. Through designing the project structure template, configuring CLI commands, and completing code generation functions, *Cadmium2* was configured for use in PROMETHeUS. Users can run a *Cadmium2* project without needing prior knowledge of *Cadmium2* semantics. They can leverage the PROMETHeUS' graphical features for building, testing and real time visualization of DEVS models, significantly lowering the entry barrier for practicing DEVS while preserving flexibility for experts.

In the future we plan to improve the architecture by allowing users to configure asynchronous inputs for both atomic models and coupled models—currently, this feature is only available for coupled models. Additionally, we intend to publish a detailed overview of PROMETHeUS's graphical interfaces and DEVS modeling methodologies.

APPENDICES

The appendices are available here.

REFERENCES

[1] "Simulation software market size, share & forecast report - 2032," Global Market Insights Inc., https://www.gminsights.com/industry-analysis/simulation-software-market (accessed Jan. 10, 2025).

[2] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of modeling and simulation*. Academic press, 2000.

[3] M. A. Abdelmegid, V. A. González, M. Poshdar, M. O'Sullivan, C. G. Walker, and F. Ying, 'Barriers to adopting simulation modelling in construction industry', *Automation in construction*, vol. 111, p. 103046, 2020.

[4] "Simulation software market size, share: Global report [2032]," Simulation Software Market Size, Share | Global Report [2032], https://www.fortunebusinessinsights.com/simulation-software-market-102435 (accessed Jan. 10, 2025).

[5] A. Linder and M. Nilsson, 'Designing a Design System for Coherent Tactical GUIs', 2022.

[6] J.-B. Filippi and P. Bisgambiglia, 'JDEVS: an implementation of a DEVS based formal framework for environmental modelling', *Environmental Modelling & Software*, vol. 19, no. 3, pp. 261–274, 2004.

[7] M. Bonaventura, G. A. Wainer, and R. Castro, 'Graphical modeling and simulation of discrete-event systems with CD++ Builder', *Simulation*, vol. 89, no. 1, pp. 4–27, 2013.

[8] S. Mittal and J. L. Risco-Martín, 'DEVSML studio: a framework for integrating domain-specific languages for discrete and continuous hybrid systems into DEVS-based m&s environment', in *Proceedings of the Summer Computer Simulation Conference*, 2016, pp. 1–8.

[9] D. Karagiannis, H. Kühn, and Others, 'Metamodelling platforms', in *EC-web*, 2002, vol. 2455, p. 182.

[10] J. Horner, T. Trautrim, C. R. Martin, G. Wainer, and I. Borshchova, 'Discrete-Event Supervisory Control for the Landing Phase of a Helicopter Flight', in *2022 Winter Simulation Conference (WSC)*, 2022, pp. 441–452.

[11] R. Cárdenas and G. Trabes, "Cadmium 2: An Object-Oriented C++ M&S Platform for the PDEVS Formalism," [Online]. Available: https://github.com/SimulationEverywhere/cadmium_v2. Accessed: Jan. 6, 2025.

[12] "GitHub · build and ship software on a single, Collaborative Platform," GitHub, https://github.com/ (accessed Jan. 10, 2025).

[13] "Welcome to Python.org," Python.org, https://www.python.org/ (accessed Jan. 13, 2025).

[14] C. Winstanley and G. Wainer, 'Testing Methodology for DEVS Models in Cadmium, in *2024 Winter Simulation Conference (WSC)*, 2024, to be published.

AUTHOR BIOGRAPHIES

CURTIS WINSTANLEY is an M.A.Sc student at Carleton University. He has a bachelor's degree in communications engineering from Carleton University, Canada. His research interests are in automated software verification and validation, and digital twins. His email address is curtiswinstanley@cmail.carleton.ca

GABRIEL WAINER is a professor at the department of Systems and Computer Engineering at Carleton University. He received is M.Sc. (1993) from the University of Buenos Aires, Argentina, and his Ph.D (1998, highest honors) from UBA/ Université Aix-Marseille-III, France. He is a fellow of SCS. His email address is gwainer@sce.carleton.ca.

IRYNA BORSHCHOVA is employed by the National Research Council of Canada, Flight Research Laboratory (NRC FRL). Dr. Borshchova also serves as an Adjunct Professor in two departments at Carleton University and holds the position of Associate Editor for the Canadian drone-focused journal "Drone Systems and Applications." Dr. Borshchova's research interests include discrete-event supervisory control in autonomous systems and statistical airspace modeling. Her multidisciplinary expertise underscores her valuable contributions to the aviation and computer science sectors.