Code Generation of Simulation Models for the US Army's Synthetic Training Environment

Chris McGroarty, Christopher Metevier
US Army DEVCOM SC
Orlando, FL
christopher.j.mcgroarty.civ@mail.mil
christopher.j.metevier.civ@mail.mil

Jim Gallogly
Cole Engineering Services Inc.
Orlando, FL
james.gallogly@cesicorp.com

Keith Snively, Anup Raval, Paul Tucker
Dynamic Animation Systems
Orlando, FL
ksnively@d-a-s.com, araval@d-a-s.com
ptucker@d-a-s.com

Scott Gallant
Effective Applications Corporation
Orlando, FL
Scott@EffectiveApplications.com

ABSTRACT

The United States (US) Army is creating a new Synthetic Training Environment (STE), with the goal of applying the best of breed simulation technologies to allow for training anytime, anywhere, and at massive scale. This goal is leading the US Army to obtain, build, and integrate technologies that are not traditionally within the simulation domain. Instead of starting from scratch and re-building all existing military models within a new simulation engine, the US Army has initiated a Generative Programming project to capture authoritative models within a machine-readable systems engineering format and then have the ability to automatically generate working software that implements the authoritative models consistently within many different simulation applications.

This effort includes two main components. The first component is a graphical modeling tool that allows subject matter experts to create, execute, and test their models. The model author does not need to know how to write software due to the use of the increasingly popular Flow-Based Programming paradigm and can see their model execute based on their actual data to ensure that the model is complete and accurate before it gets integrated into a simulation. The second component is the code generation capability to output Java, C++, and C# software that executes the models. Generating software from a single canonical source allows for consistent model representation across simulation systems, reduces software development, integration and testing costs, and reduces implementation and semantic errors system-wide.

This paper describes the graphical modeling paradigm, the machine-readable systems engineering format, code generation capability and some preliminary results. We believe this technology and process for capturing models in a format that can be tested, used for code generation, documented, and then verified will be valuable to many use cases within the simulation domain beyond STE and the US Army.

ABOUT THE AUTHORS

Chris McGroarty is the Chief Engineer for Advanced Simulation at the Combat Capabilities Development Command, Soldier Center, Simulation and Training Technology Center (DEVCOM SC STTC). His research interests include distributed simulation, novel computing architectures, innovative methods for user-simulation interaction, methodologies for making simulation more accessible by non-simulation experts, service-oriented architectures and future simulation frameworks. He manages and leads a variety of research efforts that mature, integrate and demonstrate these technologies in a relevant Army and Department of Defense context. He received his Master of Science and Bachelor of Science in Electrical Engineering from Drexel University in Philadelphia, Pennsylvania.

Christopher J. Metevier is the Chief of the Advanced Modeling and Simulation Branch at the United States Army Combat Capabilities Development Command, Soldier Center, Simulation and Training Technology Center (DEVCOM SC STTC). He has over 30 years of experience with the Army and Navy in the Modeling and Simulation

(M&S) field. His M&S experience extends across the acquisition lifecycle and includes the research, development, adaptation, integration, experimentation, test and fielding of numerous simulation technologies and systems. He received his Master of Business Administration from Webster University and his Bachelor of Science in Electrical Engineering from the University of Central Florida.

Jim Gallogly is a Senior Software Engineer with Cole Engineering Services Inc. He has over 20 years of experience with distributed modeling and simulation. Jim has worked with a diverse set of simulations developing with the High-Level Architecture (HLA), Common Training Instrumentation Architecture (CTIA), Distributed Interactive Simulation (DIS), Test and Training Enabling Architecture (TENA), Simulation Object Runtime Database (SORD), MSSV and other custom protocols in Live, Virtual, and Constructive environments. Mr. Gallogly holds a Bachelor's of Engineering in Computer Science and Electrical Engineering from Vanderbilt University

Anup Raval is a Senior Software Engineer with Dynamic Animation Systems. He has over 15 years of software development experience in modeling-simulation, capital markets, healthcare, aviation, and aerospace domains. His experience includes model driven development using modeling languages like UML and SysML. Anup has worked on enterprise application interoperability, message security, full-text search, performance tuning and build automation. He is currently working on the Generative Programming effort.

Keith Snively is currently a Principal Software Engineer for Dynamic Animation Systems. He has worked extensively in distributed simulation for the Department of Defense. Mr. Snively has supported development of numerous simulation protocol standards and implementations, including HLA, Modeling Architecture for Technology, Research, and Experimentation (MATREX) Run-Time Infrastructure Next Generation (RTI-NG), WebLVC, TENA and DIS. He has also supported development of techniques for orchestrating simulations as part of the Executable Architecture Systems Engineering (EASE) project. Currently he supports developing cloud computing simulation frameworks as well as code generation of models. Mr. Snively received a Master of Science and Bachelor degree in Mathematics from the University of Virginia.

Scott Gallant is a Systems Architect with Effective Applications Corporation. He has over 25 years of experience in distributed computing including United States Army Modeling & Simulation (M&S). Scott has led technical teams on distributed M&S programs for distributed software and System of Systems design, development, and execution management in support of technical assessments, data analysis and experimentation. He also serves as the technical lead for the implementation of the described Generative Programming project for the U.S. Army.

Paul Tucker is a Software Engineer with Dynamic Animation Systems. He has 4 years of experience building modeling and simulation applications as well as enterprise applications. He has a Bachelor’s of Science degree in Computer Science from the University of Central Florida.

Code Generation of Simulation Models for the US Army's Synthetic Training Environment

Chris McGroarty, Christopher Metevier
US Army DEVCOM SC
Orlando, FL
christopher.j.mcgroarty.civ@mail.mil,
christopher.j.metevier.civ@mail.mil

Jim Gallogly
Cole Engineering Services Inc.
Orlando, FL
james.gallogly@cesicorp.com

Keith Snively, Anup Raval, Paul Tucker
Dynamic Animation Systems
Orlando, FL
ksnively@d-a-s.com, araval@d-a-s.com
ptucker@d-a-s.com

Scott Gallant
Effective Applications Corporation
Orlando, FL
Scott@EffectiveApplications.com

BACKGROUND

There are countless simulations used within the defense industry to represent military forces across training, experimentation, analysis, and many more use cases. The physics models within these simulations are based on authoritative models and data but are implemented differently in each application. With each new simulation, the models must be recreated from scratch or derived from another simulation application with a different technical architecture, and most likely, a different programming language. Software engineers must work with subject matter experts in implementing the models which can cause some errors in translation, assumptions, or misunderstandings. This leads to inconsistent modeling, a lack of interoperability between systems, and high costs of simulation development and maintenance. Testing of the models across simulation applications is also unique to the simulation application because of the differing technical architectures, slightly different model implementation, and the current inability to test a model before it is implemented within a complex simulation application. Finally, when an update of the physics model becomes available for an authoritative model, getting that updated algorithm and/or data to each simulation requires developers to be involved with the entire lifecycle of software updates, documentation revisions and testing.

The United States (US) Army has begun development of a new simulation-based training program call the Synthetic Training Environment (STE). The goals of STE are to combine many different types of training applications including live, virtual (including game-based architectures), and constructive applications in an integrated environment. This will require interoperability between systems, including consistent implementation of physics models. STE also has the goal of software reuse, continuous integration and testing, and reducing the time and costs for model development, updates, and documentation. The Generative Programming project (McGroarty, 2021) was created to address these issues within STE as well as the broader M&S domain. This project is in its second year of activity and includes an initial Generative Programming implementation that is able to capture military-relevant models within an executable systems engineering format, to facilitate authoritative models to operate within multiple platforms (gaming, constructive, etc.) while saving time/money for development, easier updating of simulations with authoritative model updates, allowing for a central, government-owned repository of models, and to allow for the same testing data (inputs, outputs, and algorithmic data) to be used for model creation as well as software testing. This paper will describe our progress towards improving model development, consistent implementation across disparate systems, reducing long term software engineering cost and errors, as well as the path forward of the Generative Programming project.

IMPLEMENTATION

The current implementation includes three technical components of: a user-friendly visual programming interface for subject matter experts to create, test, and document their models before software engineers become involved, a structured and machine-readable format that can represent the model, and a template-based code generation back-end that can output working software in multiple programming languages. The toolset allows users to develop models using a flow-based programming interface, executes tests to ensure complete business logic and data alignment, generates documentation for the model, as well as generating code in target programming languages, currently Java, C++, and C#. Each generated language allows for executing tests and showing the model developer data results of their models. The generated code has been integrated into several simulation environments. We have integrated the generated C++ software into a constructive simulation system, VR-Forces (MAK Technologies, 2021) and we have integrated the generated C# software into a Unity-based research and development testbed, the Rapid Integration & Development Environment (RIDE) (Hartholt, 2021). For both integrations, we showed an improved model for individuals walking whereas individuals would be affected by the load they carry, the slope of the terrain, and the type of the terrain upon which they were moving. We were able to show consistent implementation of the individual movement model across two systems that were developed by different organizations at different times, and were even developed with different programming languages and technical architectures.

The Generative Programming team is working closely with developers of RIDE to guide requirements, prototype, and test capabilities within a relevant simulation environment. RIDE is a research and development focused simulation environment that allows rapid prototyping and is used across many industry and government projects as a modular, easy to use simulation. Working with RIDE developers and having access to an actual simulation environment has accelerated the Generative Programming development and is enabling a testing environment for current and future capabilities described below.

CAPTURING THE MODEL

Modelers know their area of expertise very well, but not the technical architecture used by the many simulations where the model may be used. The Generative Programming project aims at allowing modelers to capture their models in a way that can be used across simulation systems without needing to work with a software engineer. Then once the model is generated and integrated into a simulation system, it becomes easier to update the model or the data over time. To meet the goals of the Generative Programming project, we need to provide a user-friendly interface for model experts to capture their logic, data, and documentation in a single place and in a simulation agnostic way. This interface tool should output a structured and machine-readable representation of the model that will be used by the code generation component to ultimately create the software implementations across multiple simulations. The format should:

- Allow and enforce a complete representation of models without gaps in logic or data inconsistencies.
- Be programming language agnostic so the model can be used in any simulation.
- Allow for the embedding of documentation of the model's purpose, logic, and data within the model to keep the explanation of the model aligned with the logic of the model over time.
- Be machine-readable and rich in expressions and types to permit code generation.

STE CANONICAL UNIVERSAL FORMAT (SCUF)

Previously we had searched for a single existing language or tool that accomplished all the above goals. As described in our paper "Automatic Code Generation of US Army Models" (Metevier et al., 2021), we quickly realized no one tool existed to address all these needs. We split the problem into two distinct pieces: model capturing and code generation. We then explored possible formats to use to capture the model's logic, information about data stores, and everything else required by the model to generate working software. We examined five options including MATLAB (Moore, 2017), SysML (Delligatti, 2013), Modelica (The Modelica Association, 2017; Fritzson, 2005), A Mathematical Programming Language (AMPL) (Fourer, 2002), and Blockly (Pasternak, 2017). Each of those options had strengths and weaknesses, but none of them provided both the flexibility and completeness that was required for this project. Modelica and AMPL require specialized engineering knowledge that would be hard to staff and maintain over time. MATLAB is proprietary and although there is a work being done by the Object Management Group towards a SysML standard, currently the files that are output from SysML tools are extended with proprietary extensions.

Finally, we created a prototype using the Blockly tool, but the logic was very tedious and unwieldy. Therefore, we created a custom format for this project called the STE Canonical Universal Format (SCUF) that afforded us the two main benefits needed: freedom from proprietary tools and flexibility in representation. The SCUF acts as the interface between the model capturing and the generated code and allows us to work these two problem spaces in parallel.

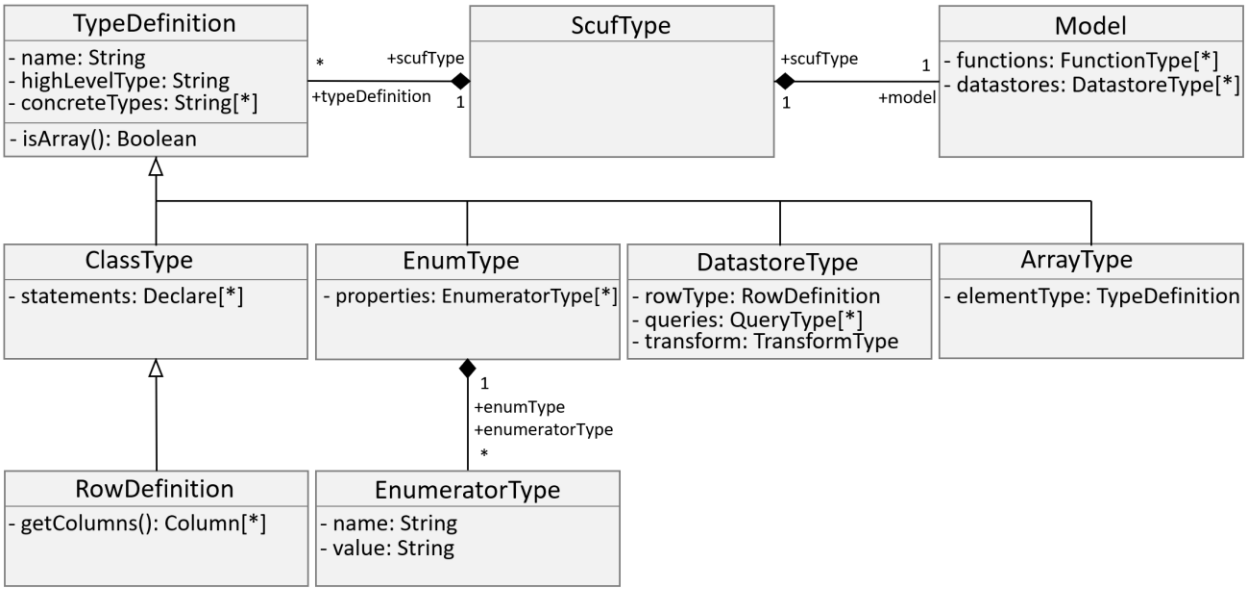

Figure 1 SCUF Meta-Model: Static Elements

The SCUF meta-model captures the abstract concepts needed to implement a model in a language agnostic manner. It represents both dynamic and static logic. Figure 1 shows static portion such as classes and types, which will be referred to and explained throughout the remainder of this document.

FLOW BASED PROGRAMMING

The SCUF is the format used by code generation, but we needed a user-friendly interface to allow SMEs to create the model and then output the SCUF representation. We chose a flow-based programming approach, which is a style of programming that does not require knowledge of a specific syntax. Instead of traditional programming languages Flow based Programming is not text based but graphic based. The gaming industry has had great success building both complex and high performant systems using Flow-based programming such as Unreal's Blueprint (Unreal Games, 2021) or more recently Unity's Bolt languages (Unity, 2021). Game studios will often have Technical Artists use flow-based systems to implement game behaviors and models without the need for Software Engineers.

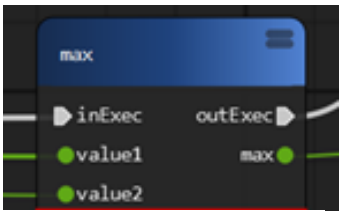

Figure 3 PyFlow Node

Flow-based programming focuses on the path that data takes between various black box processes. These black box processes are called nodes and perform a certain action based on the data that is passed into it. Figure 3 shows a node that performs a basic maximum function selecting the largest value from the two values provided. The inputs, on the left (value1, value2) have green pins that connect via lines to the node that provides the input. The output, on the right, max is also a green pin that is connected to another node. The color of the pin tells you which type of data is flowing between the nodes. If two pins are different colors, i.e., different types, they cannot be connected directly. The white pentagon pins represent execution order and are how the flow of execution is represented in a flow diagram. A complete flow diagram is referred to as a graph. Each graph has its own sets of inputs and outputs defined as well.

This black box model of treating everything like a structured input is one of the things we were trying to achieve for Generative Programming, so this was a perfect fit. Flow based programming achieved two of our goals, firstly it allowed us to provide a rigid structure for inputs and outputs and secondly with its proven accessibility to non-programmers its shows it can capture an algorithm in an accessible format to allow experts of different backgrounds.

While Unreal Blueprint and Unity Bolt are proven systems, they are built into their own architectures and generate code targeted at their respective engines so we had to find another flow-based programming authoring tool. PyFlow is a tool inspired by Blueprint and has the basics elements of a flow-based authoring tool. PyFlow is designed to run Python code at runtime and has no means to generate code. Fortunately, PyFlow was built with a plugin architecture and allowed us to extend its capabilities from running python code live, to generating SCUF and triggering our Model to Code Tool to generate code to our target languages. We took advantage of the PyFlow architecture but needed to add a lot of additional functionality specific for capturing the types of models used within simulations and to output the SCUF representation of the models, which will be described in the Capabilities section below.

CODE GENERATION

Model to Code Tool (MTCT) provides ability to read the SCUF and generate source code in various programming languages. As part of the core functionality, MTCT parses the SCUF into Java class representations of the meta-model, which serves as the basis for the code generation system. MTCT then uses the Visitor pattern to process the model classes. The Visitor pattern allows traversal of the ingested model, performing various functions, without impacting the model classes themselves. One such visitor, the scoping visitor, ensures any required arrays or pairs of types needed by the model are present. This visitor also updates associations within the meta-model, which include links for *TypeDefinition* to *Variable*, *Input*, and *Output* instances as well as *FunctionType* to any *FunctionCall* instances. This linking helps verify the correctness of the SCUF as well as simplifies access to this information during the code generation process. An example is accessing the type information for a *Variable* to determine how to perform the comparison operation, which be different for primitive and user defined types.

MTCT also contains a set of language dependent templates to turn the SCUF into code. The templating system is based upon the Apache Velocity. Once again, MTCT uses a visitor to traverse the SCUF and invoke templates to generate code. Generally, each meta-model element corresponds to a template which handles generation for that element. For instance, *ClassType* model elements use the *class-template.vm* to generate code. To simplify the implementation of these templates, the code generated from any contained elements is handed into the template. For instance, zero or more *Declare* elements can be contained in a *ClassType*. The code for *Declare* is generated using *declare-template.vm* and passed into the *class-template.vm*. In addition to simplifying the containing element template, this allows changes to low level elements to be implemented in a single location and take effect throughout the generated code. This reduces the number of templates that need to be implemented to support new languages or simulation environments. As the output from one template feed into the input of another, we refer to this as a set of cascading templates.

Currently, MTCT supports three programming languages: C++, C#, and Java. We wanted a capability to reuse existing templates as much as possible to simplify the process of adding support for new languages. For example, the processing of creating expressions with basic operators or making function calls is the same across multiple languages. To enable this goal, MTCT utilizes a search path for locating the individual template files. MTCT looks through the directories in the search path and uses the first template it finds with the matching name. This is similar to how executables are located within a user environment on Windows or Linux. Many low-level templates, such as *expression-template.vm* can be reused across languages while specializing those that contain differences, such as *class-template.vm*. The search path for templates also allows users to also override particular templates by pre-pending this path with the location of the customized version. In this way, a user could customize how enumerations or even enumerators are generated while reusing all the other existing templates. We also created control files, which determine how the generated code is placed into files and a directory structure. The control files, along with the template search path and cascading templates allow the user to completely control the code generation to create Architecture Specific Templates (AST) without modifying the MTCT core functionality. The ultimate goal of the AST is to reduce the amount of glue code the developer must write to adapt new models to a particular simulation environment. Control files and AST will be described in more detail at the bottom of the Capabilities section.

CAPABILITIES

Types

The key to a well-defined structure is well defined types. The SCUF meta-model provides elements to define primitive types, enumerations, structures, arrays, pairs, and datastores. The *TypeDefinition* model class handles

definition of primitive types and aliases. This can be used, for instance, to create a type m/s to represent speed in meters per second. The *TypeDefinition* also indicates the high-level type and concrete types the defined type pertains to. The m/s type may have a high-level type of number and a low-level type of double. These are indications to the front end PyFlow how to treat the type and backend code generation in how to generate code for the type. The *EnumerationType* handles enumerations and allows definition of enumerations by the user The *ClassType* handles user defined structures as a collection of fields. Currently classes may be composed but do not support inheritance. Any type may be contained in an array or a pair. Creating pairs for types was particularly motivated by use of datastores. The datastores are used to access input data and are discussed in the next section.

In PyFlow, each type is a pin, that represent the kind of data that can be connected to a node and therefore what data flows into or out of a node. We modified PyFlow to allow declaration of above types. For structures (*ClassType*), require associated nodes be generated to access them. We generate three types of access nodes: Break, Make, and Set. Break nodes take a structure type as an input and breaks it into pins for each part. Make is the opposite of break and takes all the parts and creates, or constructs, an instance of the struct. Set is used for modifying an existing struct and allows setting one of the values of its parts. There is one set of nodes for each struct part. The Break and Set translate into SCUF as accessing child fields and retrieving or assigning the value. The Make becomes the corresponding *Make* SCUF element. This is a special function-call that allows for the construction of user defined data type that can be generated as appropriate into a particular programming language. For instance, in Java this becomes a "new" statement.

Datastores

Many models we explored require the lookup of data from column delimited tables. Often, one or more queries are performed on a table to obtain a row or rows of data used for processing within the functions of the model. We wanted to provide a simple mechanism to declare and use datastores in a model. We also recognized that at run-time, a datastore could have different representations. A simple representation would read data from a file and store it internally, while another would be to have a datastore hook into a database. Therefore, the representation of a datastore within SCUF should not constrain its implementation in generated code. To accomplish these goals, we defined an abstract representation of the datastore and its related components within the meta-model.

Within the SCUF, a datastore contains columns, queries, and transforms. These are captured by the *DatastoreType*, *QueryType*, and *TransformType* meta-model elements. The columns are part of *DatastoreType* and define the name and type of each entry within the row of a datastore. The row definition itself becomes a *ClassType* within the model that can be used inside functions. The query provides a mechanism to retrieve data from a datastore and is the means by the model interacts with the datastore. The transform is an optional component that provides flexibility in how the datastore populates data into its columns from a data source.

A *QueryType* consists of a key set, which is a subset of the datastore columns that will be used to lookup data, and a return type, which is another subset of the columns and defaults to the full row type. Each query will generate methods on the datastore, which become Nodes in PyFlow. Two basic methods are always included: one to return a single matching result and another to return a list of all matching results. As we implemented more models, we determined some additional capabilities were also frequently required. The first was ordering results by a column within the datastore. This allows lists of results to be ordered by a user specified column. When ordering is specified, this will also add another method to retrieve the maximum and minimum entries for a given key based upon that ordering. The next was bounded lookup. This adds a method to retrieve an upper and lower bound for a given key based on the value contained in a user specified bounded column. The lower bound entry will be the entry whose bounded column contains the largest value not greater than provided bounded value. The upper bound entry will be the entry whose bounded column contains the lowest value not less than provided bounded value. The last capability was interpolated lookup. The interpolated lookup automatically interpolates values in the return type for a bounded lookup based on the bounded value. This last method was provided as models that performed bounded lookup often interpolated the result to use within the model.

Generating code for a datastore and its queries presented an interesting challenge as there are many ways to implement a datastore. An implementation may simply read data into internal map-like data structures or connect to a backend database. In both cases, implementation decisions can trade off speed of lookup and amount of internal storage. It was clear there was no correct decision for all simulation environments. Therefore, MTCT allows for differing implementations in a couple ways. First, interface (or abstract) classes are generated in all languages for

each datastore. This allows the simulation program using the model to supply a custom version of the datastore at run-time. For instance, the simulation could write (or generate) concrete classes that wrap simulation specific data lookup classes. Second, as the datastore is captured in a set of meta-model classes, the generated code can be customized by overriding the default template files for those meta-model elements. Either strategy could be used, for instance, to tie the datastore to a database as opposed to the default representation.

The final component of a datastore is the *TransformType*, which allows translating from source data columns into the datastore columns. As we implemented a number of models over the last year, we found that model data files are typically not formatted well for runtime use. The files may contain information with regards to the data that would not be used at runtime, such as date collected. In addition, data may be represented in the data source columns so as to minimize its size as opposed to supporting easy lookup. An example would be a row that contains a single system and maximum speed for that system for multiple terrain slopes across the columns. A transform addresses this problem by allowing users to translate from source data columns into the datastore columns. A transform contains two types of mappings that allow a row from the data source to be translated into one or more rows in the datastore. The first, a direct mapping, allows values from a source data column to populate values in a datastore column. The columns need to have the same contained data type but can have different names. The second, a pivot mapping, allows a set of columns from the source data to be pivoted into two datastore columns, perhaps adding multiple rows. Since these transform types are part of the meta-model, MTCT templates determine how the transformation occurs within generated code for each target language or architecture. As with all meta-model elements, the generated code can be customized as desired.

Extensible Capabilities

When writing a simulation, it quickly becomes obvious that some basic functionality is expected but will not be provided by a given language or that it may be provided but with slightly different Application Programming Interfaces (APIs). Examples include math functions, such as sine, cosine, or lerp, as well as more basic array operations. During our initial design, we had manual alignment between the Model to Code Tool (MTCT) and PyFlow as to the name of these generic methods and what the inputs and outputs would be. The MTCT was responsible for mapping the built-in methods to the target languages but this approach proved problematic. If you add a new node to PyFlow before you add it to the MTCT, you could not generate code; if there was a small discrepancy it would end up as compiler errors that you could only catch by generating code that had those methods. To mitigate this, we began creating tests for each individual method that was agreed upon to prove through continuous integration that the capability worked as expected. This whole process was tedious, error prone, and required very tightly matched versions between PyFlow and the MTCT. Additionally, this did not follow our design pattern of allowing alternative front ends as any new front end would have to implement all of these built-in methods in some way. So, we instead had the MTCT publish what built-in functions it has in a structured format (a file named ‘capabilities.json’) that can be read by PyFlow at runtime. PyFlow can then read in types and nodes that are published by MTCT and make them available in the graph editor for model development. In turn, if you changed the version of the MTCT then you would get that version of nodes available guaranteeing compatibility. Decoupling is achieved between PyFlow and MTCT by publishing and processing of function definition file. This also allows for the dynamic extension of the library of model functions available in PyFlow.

While the use of capabilities.json provides extensibility on the PyFlow front end and keeps it in sync with MTCT, these functions were still hard coded into MTCT. This became cumbersome as we implemented more models and the set of common functions grew. We therefore developed a capability to specify these functions through a JSON definition file and a template set for generating code. This JSON file differs from the capabilities.json in containing only information required by MTCT, such as library name and method signature. The separation also allows these features to evolve independently. MTCT uses an external-function template set, similarly to how is done for the SCUF elements, to generate code for different languages. While these functions are intended to be created and extended by MTCT developers, it does allow for an advanced use case for simulation developers to add experimental functions. However, use of such extensions would impact the portability of any model using such functions and should be done with care.

An interesting aspect of external functions is presented with overloaded functions. PyFlow does not support overloaded methods when it comes to the nodes themselves. MTCT adopted this approach as well as it simplified customization of function calls and the reason for over loading methods, simplifying programming code, does not

really exist in SCUF. However, the implementations of a lot of libraries have overloaded methods in them. MTCT can invoke these methods though using the customized external function templates. For example, the current capabilities.json has an intMin method and a doubleMin method. Both of these methods represent a Math.min method in Java. The customizable function calls allow the user to convert the intMin and doubleMin function calls to Math.min function calls during code generation.

Model Using Models

As systems grow more complex defining all the methods within your model becomes cumbersome and does not promote re-use. Ultimately one model needs to be able to use the methods and types of another model. While we recognized this as a fundamental need, importing imposes several challenges on both PyFlow and MTCT. Importing other graphs is not a functionality PyFlow supported by default. We address this in PyFlow in much the same way we capture the capabilities.json from MTCT. We save the signatures of the methods and types as a snapshot at the time a model is imported and save the snapshot as part of the current model. This allows fast loading and unloading of nodes, which correspond to methods of the imported model, as well as allows the user to better control when the updating the version of the imported model. One downside to this approach, though, is that the file sizes will increase but thus far that has not posed an issue for our research purposes.

More challenges arise on the code generation side in MTCT. How does generated code implement one model using another model? Should this use inheritance, composition, or association? The same questions apply to how models use datastore from other models, or even the current model. We did not want to have to make these decisions on within PyFlow or SCUF and preferred to leave it to the code generation system. Therefore, we introduced several meta-model classes to support models using models in a generic fashion. The first was *ImportType*, which handles importing the methods and types of one SCUF into another SCUF. The next were *ModelCall* and *DatastoreCall*. These elements are similar to a basic *FunctionCall*, but imply the function is being invoked on another model (i.e., not the current model) or on a datastore. Since these types will correspond to their own template files at code generation time, this allows a lot of freedom within the defined templates in how dependent models and datastores are accessed. This frees the model developer from what is a low level programming concern and allows this determination to be made, and independently customized, within the code generation system.

Several other considerations became apparent when generating code for models using models. One was in name collision. To avoid type names overlapping, we added packaging to SCUF. Every SCUF defines a single package for the model that is unique to that SCUF file. When one SCUF uses a type from another SCUF, it includes this package name to clarify where it originates. Another issue was output directories for generated code. Here again, the package is in the code generation system to create directories into which code is generated for a model by default. This particular behavior can be customized with Control File, as discussed in Architecture Specific Templates section. The last was generating code for self-contained tests. To address this issues MTCT added a mode to generate code for all dependencies, along with the current model into a single project. This allows the system to maintain the single button testing capability within PyFlow.

Test and Execution

A critical element of the Generative Programming effort is providing the ability for the model developer to run the code and see the results; the key to finding omissions in data or algorithms is the ability to run and test the model against a dataset and visualize the output. With this key design tenet in mind, the MTCT must be able to generate running code from any given graph as well as run tests with the data that is provided. The MTCT therefore provides some reference implementations, such as reading datastores from delimited files. This does not mean that all users must use delimited files, but it provides an accessible and approachable way to test.
To begin the testing process, a user provides a file that contains the inputs to the model as well as files that contain data for any datastores. A user selects the language to code generate to -- currently supported are Java, C++, and C# -- and then pushes the “Generate and Test” button. PyFlow exports the SCUF for the current graph, and any imported graphs, and calls the MTCT to generate the code from the SCUF. MTCT generates test source code with the model code to read test inputs, execute model functions and create test results. It also generates scripts for compiling and executing the tests. During code generation, MTCT streams console output into a widget within PyFlow and includes hyperlinks for generated code, which can be opened in a user specified editor. Once generation is complete, PyFlow calls a test script, supplying the input files and data files for the test. The output of the test run

is recorded into a file in CSV format. The file is then picked up by PyFlow and presented to the user along with a graph of the data that the user can modify and inspect. Repeatability of results is important to prove the model function works consistently, we observed an issue with repeatability of results when random numbers are used in model. To achieve repeatable results the implementation of random number was updated to use seed value. This allowed the repeatability of results for model functions using random numbers.

The test capability allows the model developer to see the output of the model, methods within that model, and inspect the results to verify that the model is operating as desired from within the PyFlow interface. The model developer can make changes to the model or data and run additional test iterations. This provides a tight and easy test loop to explore ideas and learn the tool as well as learn any gaps that may exist in the data or in the model.

Documentation

Even with strong structure, well-defined types, and the ability to run tests, it is still useful to document what happens within the code. PyFlow has comment nodes to draw and group nodes together. We also added a Rich Text Editor that allows individual nodes to have comments. Each graph allows free form Rich Text editing of text for the graph, and inputs and outputs. While looking through the code in PyFlow, this documentation can prove useful and enlightening. Sometimes it is useful to just be able to print out and read the details of the model, so we also implemented a document export feature which creates a Word document including the model description, the method descriptions, method inputs and outputs, types and data store descriptions. This document further highlights any methods that call other methods and which ones depend on data stores. This provides a clear tabular view of the data details as well as the free form text that might help explain how the model is intended to be used or integrated into a system. The same documentation that is exported into the word document, is also exported as comments within the generated code to put the documentation as close to the user as possible. The comments are captured within the SCUF meta-model. Generated comments use language specific formatting and can be customized using corresponding templates. The goal is to make this comment conform standards defined by different documentation generation tools and be able to generate html documentation along with the source code.

Architecture Specific Templates

One of the problems that we faced with the MTCT was that the generated code had to be manually integrated into any application that wanted to use it. The users of the generated code also had to adapt their architecture to fit in the generated model. This proved to be tedious and repetitive especially when adding in multiple models to the same simulation system. To make the integration easier, we created the concept of architecture specific templating. This allows the user full control over what gets generated and how things get generated and leverages some of the concepts discussed above such as custom template sets and external function libraries. This not only simplifies integration but can support improving performance of generated code within for a particular simulation system.

A key concept of Architecture Specific Templates is the Control File. Control Files are used for providing more power to the user in how source code files are generated from the templates. They allow the user to specify what parts of the SCUF model should get generated and how those parts will get generated. The Control File itself is distinct from the templates and specified in JSON format. The file contains a list of sections, each of which can generate code for specific parts of the SCUF. These parts are the enumerations, classes, datastores, and the model. Each section can also use its own template search path. This allows for multiple passes over the meta-model, each potentially with its own configuration. For example, our default implementation for the datastores uses two main templates: one for an interface for the datastore and one for the implementation. The control files specify that the datastore should be passed over twice by the code generator, once with a template for the interface class and once with a template for the implementation class. The cascading template sets simplify this by only requiring the main datastore template to be overridden in this case.

Control files also control whether the output for of all model classes of a certain type should be output to a single file or multiple files. A benefit of this approach is allowing for the differences in programming languages for how data types and classes may or may not be collocated in the same file. For example, in Java, the classes are expected to be in their own file with the name of the file matching that of the class. As opposed to C++ where the classes can all be included in the same file if the developer chooses to design it that way. One of the other capabilities of control files are to include external files or “project files” into the code generation process. These files can be whatever the

user needs to include in their project such as utility classes or build files. The contents of the file should just be added to a velocity template and then the template and file name just need to be added to the project control section of the control files. These templates also get a default set of information provided by velocity that includes things like the model's name, the list of enumerations, the list of datastores and more. This provides the user with complete flexibility to do anything that they need to do with the project files. The control files simplify the creation of custom template sets for a particular language and simulation environment, allowing adaptation of models to new platforms. Within a control file, the user can specify the initial template search path to be appended to the cascading template search. This allows the user to choose what template set they want to use for each pass over the meta-model. With the power of control files, external function libraries, and custom template sets, a user can write their own variation of these files to generate code specific to their own architecture. In fact, this is exactly how MTCT customized generated code for the languages C++, Java, and C#.

Two additional use cases we have been looking at is generating a Unity MonoBehavior as well as generating behaviors for RIDE, the Unity-based simulation that we are generating models for. To do this, we will need to write a custom template set for the main model class. We will also need to include and Unity or RIDE specific configuration files in the project control section of the control file. A major challenge we faced with both of these architectures is that the models follow a component structure. This means that the models have an internal state that the model itself interacts with and updates. Our current generated code supports a more static architecture where all of the inputs are provided as parameters and it produces the output from a static context. We are investigating a solution where the MTCT would automatically detect which class members in the model need to be included in the component state. This can be done by looking at which variables are being passed into the functions with the Init and Update tags, method tagging is described in the next paragraph. These tagged methods represent the methods to initialize a component and to update a component. Doing this will automatically create a component with internal state. These class members will be part of the component and can be handled or used within the glue code as needed. The state of the model would be controlled and customized through the custom template sets.

Another challenge was that some of the component modeling in Unity and RIDE have methods that are meant to handle specific events within the framework. For example, Unity has a start method that runs at the initialization of the component and an update method that runs during each frame. The MTCT needed a way to be aware of these types of methods so that they are customizable and so that they can be handled differently compared to other methods in the model. We are currently looking into introducing method tagging within the SCUF. This would allow users to tag certain methods as Update, Start, etc. Doing this would let the MTCT know what type of method it is, and the code generator would be able to handle the different cases. A method having a tag would also let the MTCT know that it is a component style model which would potentially change how the code generating happens.

SUMMARY AND PLAN FORWARD

This paper described the purpose of the Generative Programming project, described the capabilities of the toolset, and detailed technical design choices made throughout the project. The Generative Programming toolset takes a data-centric form of modeling that allows subject matter experts to capture their model and data structures without requiring software engineering expertise. We have adapted an open source tool rather than starting from scratch and made adaptations specific to modeling and simulation requirements. Instead of linking the modeling tool directly with the code generation process, we have created an intermediary file format, SCUF, that is independent of both simulations and modeling tools. This allows the possibility of additional modeling tools to output SCUF and still take advantage of the code generation functionality. We have successfully used the Generative Programming toolset to represent dozens of models across two different simulation environments. This is an active project, and we will continue to expand the number and types of models that can be represented within the toolset. We will also be adding additional target simulation environments for the software that is automatically generated from the models.

Our future plans for the Generative Programming project are to expand the capabilities to address a broader set of requirements for additional simulation projects and determine if we can address common code generation issues. One issue when generating software from a centralized model, is that there could be potential quality issues where the generated code is not as good as what a software developer would create if developing the model from scratch. Our approach to alleviate this issue is twofold: improve the architecture specific templating capability as mentioned in the previous section to allow for continuous improvement based on the target simulation architectures; and to

allow for multiple representations of the same model across differing levels of fidelity and resolution. This will allow for use case appropriate usage of models. For example, force on force simulations can use low resolution models to remain performant at large scales while entity level simulations can use the highest resolution models for the most accurate representation. As models are developed, managing the collection could become unwieldy over time. To address this concern in addition to the multiple levels of resolution and fidelity of the same model, we plan on developing a mechanism for models to be related to each other based on what is being modeled, how models use other models, level of resolution, pedigree, and any other descriptive attribute useful to subject matter experts and simulation developers. The future plans of applying Generative Programming to the US Army STE program will require the utilization of a very large set of models across a wide variety of simulation systems. We will continue to create and mature capabilities towards the goals of making it easier and faster to create models and integrate the generated software into simulations. This will provide the M&S community with valuable insights into model-based systems engineering, development, and application across the live, virtual, constructive, and gaming systems.

REFERENCES

Delligatti, L. (2013). *SysML Distilled - A Brief Guide to the Systems Modeling Language*, New York, Addison-Wesley.

Epic Games (2021). Blueprint Visual Scripting Retrieved May 14, 2021, from https://docs.unrealengine.com/en-US/ProgrammingAndScripting/Blueprints/index.html

Fourer, F., Gay, D., & Kernighan, B. (2002). *AMPL: A Modeling Language for Mathematical Programming*, Cengage Learning.

Fritzson, P., Aronsson, P, Lundvall, H., Nystrom, K, Pop, A., Saldamli, L, & Broman, D. (2005). The OpenModelica Modeling, Simulation, and Development Environment, *46th Conference on Simulation and Modelling of the Scandinavian Simulation Society (SIMS2005)*.

Hartholt, A. (2021). Introducing RIDE: Lowering the Barrier of Entry to Simulation and Training through the Rapid Integration & Development Environment. *Simulation Innovation Workshop*.

MAK Technologies. (2021). VR-Forces Capabilities. Retrieved May 14, 2021 from https://www.mak.com/product-capabilities/967-vr-forces-4-9-capabilities/file

McGroarty, C. et al. (2021). The Application of Flow-Based Programming to the Code Generation of Simulation Models. *Simulation Innovation Workshop*.

Metevier, C. et al. (2020). Automatic Code Generation of US Army Models. *Simulation Innovation Workshop*.

The Modelica Association (2017). *Modelica – A Unified Object-Oriented Language for Systems Modeling. Language Specification 3.4*. Retrieved May 14, 2021, from https://modelica.org/documents/ModelicaSpec34.pdf

Moore, H. (2017). *MATLAB for Engineers*. Pearson; 5 edition.

Pasternak, P, Fenichel, R, & Marshall, A. (2017). Tips for Creating a Block Language with Blockly. *IEEE Blocks and Beyond Workshop (B&B)*.

Unity (2021). Visual Scripting. Retrieved May 14, 2021, from https://unity.com/products/unity-visual-scripting