Toward Scalable Collaborative Metaprogramming: A Case Study to Integrate Two Metaprogramming Environments

Herwig Mannaert
University of Antwerp
Antwerp, Belgium
Email: herwig.mannaert@uantwerp.be

Chris McGroarty
U.S. Army Combat Capabilities Development
Command Soldier Center (CCDC SC)
Orlando, Florida, USA
Email: christopher.j.mcgroarty.civ@mail.mil

Scott Gallant
Effective Applications Corporation
Orlando, Florida, USA
Email: Scott@EffectiveApplications.com

Koen De Cock
NSX BV
Niel, Belgium
Email: koen@nsx.normalizedsystems.org

Jim Gallogly
Cole Engineering Services Inc.
Orlando, Florida, USA
Email: james.gallogly@cesicorp.com

Anup Raval and Keith Snively
Dynamic Animation Systems
Fairfax, Virginia, USA
Email: araval,ksnively@d-a-s.com

***Abstract*—The automated generation of source code, often referred to as metaprogramming, has been pursued for decades in computer programming. Though many such metaprogramming environments have been proposed and implemented, scalable collaboration within and between such environments remains challenging. It has been argued in previous work that a meta-circular metaprogramming architecture, where the the metaprogramming code (re)generates itself, enables a more scalable collaboration and easier integration. In this contribution, an explorative case study is performed to integrate this meta-circular architecture with another metaprogramming environment. Based on a detailed description of the architectures of both metaprogramming environments, the various technical aspects and issues concerning this integration are analyzed. Some preliminary results from applying this approach in practice are presented and discussed.**

***Index Terms*—Evolvability; Normalized Systems; Simulation Models; Automated programming; Case Study**

I. Introduction

This paper extends a previous paper which was originally presented at the Fifteenth International Conference on Software Engineering Advances (ICSEA) 2020 [1].

The automated generation of source code, often referred to as automatic programming or metaprogramming, has been pursued for decades in computer programming. Though the increase of programming productivity has always been an important goal of automatic programming, its value is of course not limited to development productivity. Various disciplines like systems engineering, modeling, simulation, and business process design could reap significant benefits from metaprogramming techniques.

While many implementations of such automatic programming or metaprogramming exist, many people believe that automatic programming has yet to reach its full potential [2][3]. Moreover, where large-scale collaboration in a single metaprogramming environment is not straightforward, realizing such a scalable collaboration between different metaprogramming environments is definitely challenging.

In our previous work [4] [5], we have presented a meta-circular implementation of a metaprogramming environment, and have argued that this architecture enables a scalable collaboration between various metaprogramming projects. In this contribution, we perform an explorative case study to perform a first integration with another metaprogramming environment. To remain generic, the two metaprogramming environments are aimed at generative programming for completely different types of software systems, and based on totally different meta-models. At the same time, they are well suited for this study, as they both pursue a more horizontal integration architecture. The case study aims to serve as an architectural pathfinder for such integrations, and to identify remaining issues that hamper the scalability of the approach.

The remainder of this paper is structured as follows. In Section II, we briefly present some aspects and terminology with regard to metaprogramming, and argue the relevance of two related concepts: meta-circularity and systems integration. Based on this concept of systems integration, we argue for more horizontal integration architectures to enable scalable collaboration. The next two sections present the architecture and meta-model of both metaprogramming environments whose integration is explored in this contribution. Section III discusses the Normalized Systems metaprogramming environment and refers rather extensively to previous work. Section IV offers a detailed architectural description of the generative programming environment for simulation models. Based on these architectures, Section V elaborates on the integration of these metaprogramming environments, detailing the various

technical aspects, the achieved progress, and the remaining issues. Finally, we present some conclusions in Section VI.

II. Metaprogramming and Systems Integration

In this section, we give an overview of the main concepts and terminology regarding metaprogramming, and discuss the related concept of meta-circularity. Based on the basic charasteristics of metaprogramming, we propose to leverage the technique of systems integration to pursue collaborative and scalable metaprogramming. We also argue that the two selected metaprogramming environments are well suited for a representative case study.

A. Metaprogramming Concepts and Meta-Circularity

The automatic generation of source code is probably as old as software programming itself, and is in general referred to by various names. *Automatic programming*, stresses the act of automatically generating source code from a model or template, and has been called ”a euphemism for programming in a higher-level language than was then available to the programmer” by David Parnas [6]. *Generative programming*, ”to manufacture software components in an automated way” [7], emphasizes the manufacturing aspect and the similarity to production and the industrial revolution. *Metaprogramming*, sometimes described as a programming technique in which ”computer programs have the ability to treat other programs as their data” [8], stresses the fact that this is an activity situated at the meta-level, i.e., writing software programs that write software programs.

Academic papers on metaprogramming based on intermediate representations or *Domain Specific Languages (DSLs)*, e.g., [9], focus in general on a specific implementation. Also related to metaprogramming are software development methodologies such as *Model-Driven Engineering (MDE)* and *Model-Driven Architecture (MDA)*, requiring and/or implying the availability of tools for the automatic generation of source code. Today, these model-driven code generation tools are often referred to as *Low-Code Development Platforms (LCDP)* or *No-Code Development Platforms (NCDP)*, i.e., software that enables developers to create application software through configuration instead of traditional programming. This field is still evolving and facing criticisms, as some question whether these platforms are suitable for large-scale and mission-critical enterprise applications [2], while others even question whether these platforms actually make development cheaper or easier [3]. Moreover, defining an intermediate representation or reusing DSLs is still a subject of research today. We mention the contributions of Wortmann [10], presenting a novel conceptual model for the systematic reuse of DSLs, and Gusarov et al. [11], proposing an intermediate representation to be used for code generation.

Concepts somewhat related to metaprogramming are homoiconicity and meta-circularity. Both concepts refer to some kind of circular behavior, and are also aimed at the increase of the abstraction level, and thereby the productivity of computer

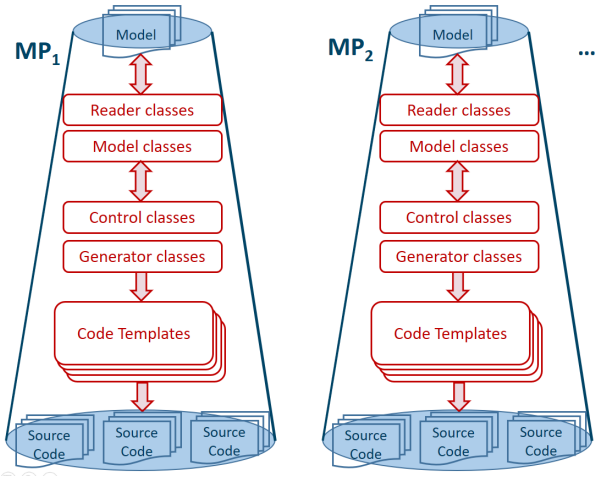

Fig. 1. Representation of the duplication of metaprogramming silos.

programming. Homoiconicity is specifically associated with a language that can be manipulated as data using that language, and traces back to the design of the language TRAC [12], and to similar concepts in an earlier paper from McIlroy [13]. Meta-circularity, first coined by Reynolds describing his meta-circular interpreter [14], expresses the fact that there is a connection or feedback loop between the meta-level, the internal model of the language, and the actual models or code expressed in the language. Such circular properties have the potential to be highly beneficial for metaprogramming through a reduction of complexity for the metaprogrammers. Indeed, metaprogrammers are forced to deal on a continuous basis with both the generative programming code and the generated code. A unified view on both the metaprogramming code and the source code being generated could potentially reduce the cognitive load for the metaprogrammers. Moreover, advancements in programming techniques could be applied simultaneously to both the generative and generated code.

B. Systems Integration and Scalable Metaprogramming

Based on a generic engineering concept, systems integration in information technology refers to the process of linking together different computing systems and software applications, to act as a coordinated whole. Systems integration is becoming a pervasive concern, as more and more systems are designed to connect to other systems, both within and between organizations. Due to the many, often disparate, metaprogramming environments and tools in practice, we argue that systems integration should be explored and pursued more at the metaprogramming level. Just as traditional systems integration often focuses on increasing value to the customer [15], systems integration at the metaprogramming level could provide value to their customers, i.e., the software developers.

Something all implementations of automatic programming or metaprogramming have in common, is that they perform a transformation from domain models and/or intermediate models to code generators and programming code. In general,

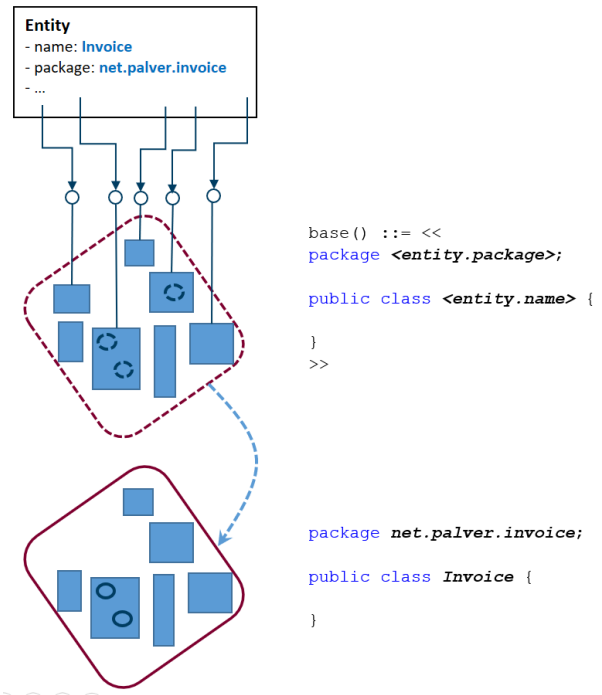

Fig. 2. Instantiating a coding template with model parameters.

metaprogramming or code generation environments also exhibit a rather straightforward internal structure. This structure is schematically represented for a single metaprogramming environment at the left side of Figure 1, and consists of:

- *model files* containing the model parameters.
- *reader classes* to read the model files.
- *model classes* to represent the model parameters.
- *control classes* selecting and invoking the different generator classes.
- *generator classes* instantiating the source templates, and feeding the model parameters to the source templates.
- *source templates* containing the parameterised code.

Figure 2 provides a schematic representation of a very elementary code generation. An instance of a model *entity*, with name *Invoice* and belonging to a package *net.palver.invoice*, is fed into a coding template for a base class. In this template, the values of the model entities are represented as parameters. The generator code will resolve these parameters and replace them with the actual values of the model entity, resulting in real source code for that domain entity.

Another metaprogramming environment will have a similar internal structure, as schematically represented at the right side of Figure 1. Such similar but duplicated architectures exhibit a *vertical integration* architecture. In this architecture, the functional entities are also referred to as *silos*, and metaprogramming silos entail several significant drawbacks. First, it is hard to collaborate between the different metaprogramming silos, as both the nature of the models and the code generators will be different. Second, contributing to the metaprogramming environment will require programmers to learn the internal structure of the model and control classes in the metaprogramming code. As metaprogamming code is intrinsically abstract, this is in general not a trivial task. And third, as contributions of individual programmers will be spread out across the models, readers, control classes, and actual coding templates, it will be a challenge to maintain a consistent decoupling between these different concerns.

We have argued in our previous work that in order to achieve productive and scalable adoption of automatic programming techniques, some fundamental issues need to be addressed [16][4]. First, to cope with the increasing complexity due to changes, we have proposed to combine automatic programming with the evolvability approach of *Normalized Systems Theory (NST)* providing (re)generation of the recurring structure and re-injection of the custom code [16]. Second, to avoid the growing burden of maintaining the often complex metacode and continuously adapting it to new technologies, we have proposed a meta-circular architecture to regenerate the metaprogramming code itself as well [4]. We will go into some more detail on NST and the corresponding metaprogramming environment in the next section.

As this meta-circular architecture establishes a clear decoupling between the models and the code generation templates [4], it allows for the definition of programming interfaces at both ends of the transformation. This should remove the need for contributors to get acquainted with the internal structure of the metaprogramming environment. It also enables a more *horizontal integration* architecture, by allowing developers to collaborate on both sides of the interface. Modelers and designers are able to collaborate on models, gradually improving existing model versions and variants, and adding on a regular basis new functional modules. (Meta)programmers can collaborate on coding templates, gradually improving and integrating new insights and coding techniques, adding and improving implementations of cross-cutting concerns, and providing support for modified and/or new technologies and frameworks. Moreover, an horizontal integration architecture could facilitate collaboration between different metaprogramming environments. Though many trade publications and academic papers on metaprogramming exist, they focus in general on specific implementations and not on the integration of different implementations. Exploring such a collaborative integration is the purpose of the case study in this paper.

C. An Explorative Case Study as a Proof of Concept

Our goal is to investigate the use of an horizontal integration architecture for the collaboration between different metaprogramming environments through an explorative case study. To serve as a representative case study and a valid *proof of concept*, two metaprograming environments were chosen that exhibit several key characteristics. First, these environments themselves are no mere prototypes. They have been developed for years and have been used in practice by many users in many different use cases. Second, these environments target the automatic programming of two totally different types of

software systems: multi-tier web-based information systems, and executable (army) models for simulation systems. Consequently, the two metaprogramming environments have a completely different meta-model. Third, these environments use different technologies at both sides of the horizontal integration architecture, i.e., both the front-end technologies capturing the models, and the target programming languages —even the code templating engines— are different. At the same time however, both metaprogramming environments share a structured decoupling between the definition of models and the generation of code, providing a starting point for an horizontal integration effort.

III. Normalized Systems Elements Metaprogramming

In this section, we present the structure of the metaprogramming environment for web information systems. Its meta-circular architecture explicitly aims to facilitate and realize horizontal integration and scalable collaboration.

Normalized Systems Theory (NST), theoretically founded on the concept of *stability* from systems theory, was proposed to provide an ex-ante proven approach to build evolvable software [16][17][18]. The theory prescribes a set of theorems (*Separation of Concerns*, *Action Version Transparency*, *Data Version Transparency*, and *Separation of States*) and formally proves that any violation of any of the preceding *theorems* will result in combinatorial effects thereby hampering evolvability. As the application of the theorems in practice has shown to result in very fine-grained modular structures, it is in general difficult to achieve by manual programming. Therefore, the theory also proposes a set of design patterns to generate the main building blocks of (web-based) information systems [16], called the *NS elements*: *data element*, *action element*, *workflow element*, *connector element*, and *trigger element*.

An information system is defined as a set of instances of these elements, and the NST metaprogramming environment instantiates for every element instance the corresponding design pattern. This generated or so-called *expanded* boiler plate code is in general complemented with custom code or *craftings* to add non-standard functionality, such as user screens and business logic. This custom code can be automatically *harvested* from within the anchors, and *re-injected* when the recurring element structures are regenerated.

While the NST metaprogramming environment was originally implemented in a traditional metaprogramming silo as represented in Figure 1, it has been evolved recently into a meta-circular architecture [4][5]. This meta-circular architecture, described in [5] and schematically represented in Figure 3, enables both the regeneration of the metaprogramming code itself, and allows for a structural decoupling between the two sides of the transformation, i.e., the domain models and the code generating templates.

In the following subsections, we briefly summarize the different parts of this metaprogramming environment.

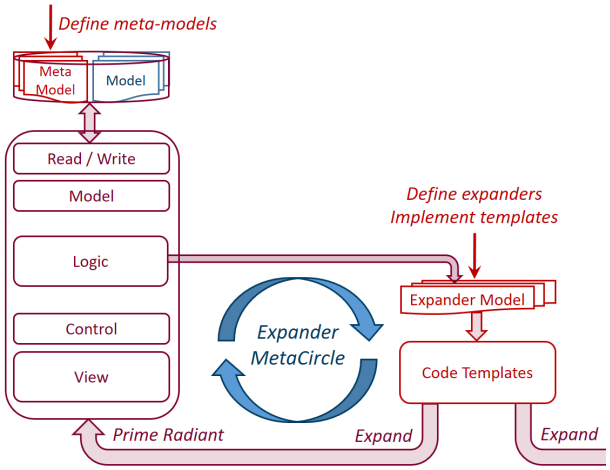

Fig. 3. Closing the meta-circle for expanders and meta-application.

A. Systems Modeling

The domain models for the web-based information systems are specified as sets of instances of the various types of NS elements. As the NS elements, e.g., data and task elements, are closely aligned to traditional primitives in information systems modeling and analysis, their definition and design is similar as well. To this purpose an *NS Modeler* was developed [19], allowing system analysts and designers to enter NS models graphically, in much the same way as traditional modeling and design tools do. Data elements can be modeled in a graphical interface similar to most *ERD (Entity Relationship Diagram)* visualizations and data modeling tools, allowing the designer to define and manipulate data entities, their attributes, and relationships. Task and flow elements can be identified in a graphical interface similar to most *BPM (Business Process Modeling)* visualizations, with the exception that designers are only allowed to define flows as state machines operating on a single target data element [19].

The various NS elements or models can also be designed and manipulated in a dedicated meta-application, called the *Prime Radiant*, using a table-based interface. This meta-application is a regular NS web application that can be generated and rejuvenated based on its own model, being the NS meta-model. Unlike the *NS Modeler*, the *Prime Radiant* allows the designers and developers to specifiy various application and technology settings [5], and to directly invoke both code generation and rejuvenation, and building and deployment of NS web applications.

The domain models of the various NS web applications are stored in XML files specifying the various NS elements, e.g., a data element with its attributes, relationships, and finder queries. The underlying structure of those XML files, i.e., the NST meta-model, is formally defined in corresponding *XSD (XML Schema Definition Language)* schema files. The XML models can be stored both locally and in central repositories, and can be exchanged between between different designers and developers, and between various instances of the NS Modeler

and Prime Radiant. The actual code generation can be invoked from the Prime Radiant meta-application, or from a command line interface accessing the XML files.

B. The NST Meta-Model

As the NS meta-model is just another NS model [4][5], the various elements of the meta-model can be specified in XML files, just like any other instance of a data element. Aimed at the automatic programming of multi-tier *web-based information systems*, the meta-model of the NST metaprogramming environment is a model for web-based information systems. The core part of the data model of this metaprogramming environment is represented in Figure 4 using a screenshot from the *NS Modeler* tool. It is, as mentioned above, similar to most *ERD (Entity Relationship Diagram)* visualizations, but uses colors to distinguish between different types of data entities [19], e.g., light blue for primary data entities and light red for taxonomy entities.

By looking at the NS meta-model, we can browse through the structure of a regular NS model. The unit of an NS domain model is a *component*, and within such a component model, we distinguish the various types of NS elements [16], such as *Data elements*, *Task elements*, and *Flow elements*. These elements, colored light blue and located in the top row, can have options, e.g., *Task options*. Both the entities representing elements and their corresponding options, are accompanied by a typing or taxonomy entity, e.g., *Task element type* or *Task option type*, represented in light red. The data elements contain a number of attributes or *Fields*, where a field can be either a data attribute or a relationship link, and provide a number of *Finders*. Both fields and finders can have options characterized by corresponding option types.

Apart from being more elaborate than the representation of Figure 4, the NS metaprogramming environment also defines a meta-model for the specification of technology settings, such as specific frameworks implementing cross-cutting concerns to be used in the generated code, and build or deployment parameters. In this way, web applications can be generated and deployed using different underlying technologies, while at the same time allowing developers to exchange models with corresponding technology settings to ensure repeatable code generation and deployment of application models.

C. Code Generation

As explained in detail in [5], the NST metaprogramming environment is highly modular and uses a declarative control mechanism. The code generation environment for web-based information systems consists of 182 individual code generators or *expanders*. Every individual code generator or *artifact expander* is declared in an `Expander` XML file. Such an expansion *control file* specifyies for instance the type of element it belongs to, the application layer it belongs to, the technology that it uses, and the various properties of the source artifact that it generates. In this way, the metaprogramming environment can be extended with alternative variations of expanders that provide another implementation and/or use another underlying technology or programming language.

For every declared artifact expander, one needs to provide a coding `Template`, based on the *StringTemplate (ST)* templating engine library. For the NS metaprogramming environment for web applications, the various templates contain currently source code in Java, JavaScript, HTML, XML, and SQL. A template for an individual expander is in general modularized or hierarchically structured itself. To avoid duplication and in accordance with the strategy of of *single sourcing* [20], a template for a *DTO (Data Transfer Object* would use for instance subtemplates for the variable declarations and the get- and set-methods. Other basic coding units like logging or error throwing are also defined in a single template. The templates also contain so-called *anchors*, enabling developers to write additional custom code that can be *harvested* and *re-injected* during consecutive (re)generations.

To access the various attributes and parameters from the elements in the domain models, an XML expander `Mapping` file needs to be defined for every individual expander. Such a mapping file specifies the various parameters that are made available to the template in terms of *Object-Graph Navigation Language (OGNL)* expressions. These expressions are evaluated on the object instances representing the elements of the domain model, e.g., `dataElement.name` [5].

IV. Generative Programming of Simulation Models

In this section, we present the structure of the metaprogramming environment for simulation models. This second metaprogramming environment is concerned with a completely different application domain, i.e., models for simulation systems, and is based on a totally different meta-model. However, by clearly separating the modeling in the front-end from the generative programming in the back-end, it is also pursuing a more horizontal integration architecture.

The United States Army has developed and documented hundreds of approved models for representing behaviors and systems, often separate from the simulation environments where they are to be implemented. The manual translation of these models into actual simulation environments by software developers, leads to implementation errors and verification difficulties, and is unable to avoid the workload of incorporating these models into other simulation environments.

In order to address these potential drawbacks, a generative programming approach is being pursued, aiming to capture military-relevant models within an executable systems engineering format, and to facilitate authoritative models to operate within multiple platforms. The goal of this work is to be able to capture authoritative conceptual models and then to generate software to implement those representations/behaviors. This generated software can be quickly integrated into multiple simulations regardless of their programming language thereby saving development cost and improving the consistency across simulation systems.

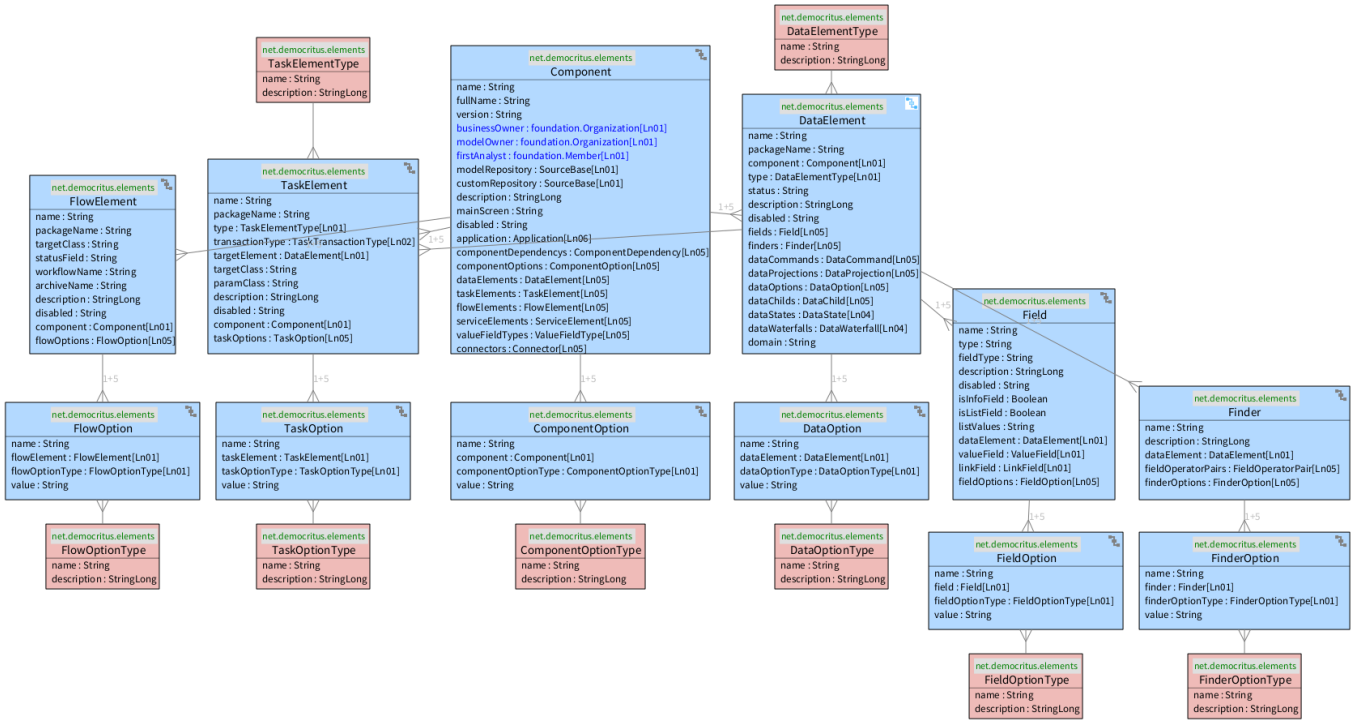

Fig. 4. A graphical representation of the core part the NS (data) meta-model.

The architecture of this metaprogramming environment, schematically represented in Figure 5, divides the problem into two domains, i.e., the front-end and the back-end. In the front-end, corresponding to the conceptual models at the left column, the *Subject Matter Experts (SME)*, scientists, and software model developers are able record the model definitions and behaviors or algorithms. In the back-end, represented in the three other columns, those model definitions and algorithms are transformed through templating and metaprogramming into executable code, targeted at specific architectures and implementations. To properly decouple these parts, an *Interchange Format (IF)* was created that allows one or more front-ends to be created to record models in a way that suits the needs of the front-end user community, and to pass those models to be used for code generation in the back-end.

In the following subsections, we describe the different parts of this generative programming architecture in more detail.

A. Front-End Visual Programming

The Generative Programming environment allows experts to create models using a flow-based programming tool. Flow-based programming [21] has become popular in game engines as well because it allows level designers, artists, and other non-programmers to create some complex business logic. The metaprogramming tool is based on the open-source project *PyFlow* [22], but with many improvements that allow modelers to represent structures and workflows most common within modeling and simulation. The goal of the project is to have a visual tool that allows subject matter experts to author their models without having to know how to develop software within a certain simulation system. The project was initially based on *Blockly* [23], a tool aiming to help non-programmers create software visually. As the structures used in *Blockly* resemble software logic puzzle pieces that fit together in specific ways, the users are fundamentally creating logic in a similar form to software, but without having to know syntax. This mechanism quickly became cumbersome with the more complex models and meant that the authors had to understand some basic software constructs. It was therefore decided to move towards flow-based programming because it was easier to create models as the visual representation of the business logic was based on the flow of data rather than using software constructs. Figure 6 represents a sample node that depicts a function that takes in two variables and outputs the maximum value. An `inExec` pin shows the execution coming into the node along with two other inputs seen on the left, `value1` and `value2`. The line going outwards from the `outExec` pin shows where execution is to go next, while the max line will go to whichever future function that will use that value.

A major concern was to make sure that the selected tool was *representation complete*, i.e., allowed one to represent any business logic that could be constructed in software, provided a rigid structure for inputs and outputs, and was easy to read and write. The selection of PyFlow for the flow-based programming tool allowed us to start with an open-source tool that could be improve upon for the simulation domain. Many additions were made to PyFlow, most notably we have made it more performant for large-scale graphs, added data querying capabilities that are most often used in our simulations, and added the ability for the user to write documentation within

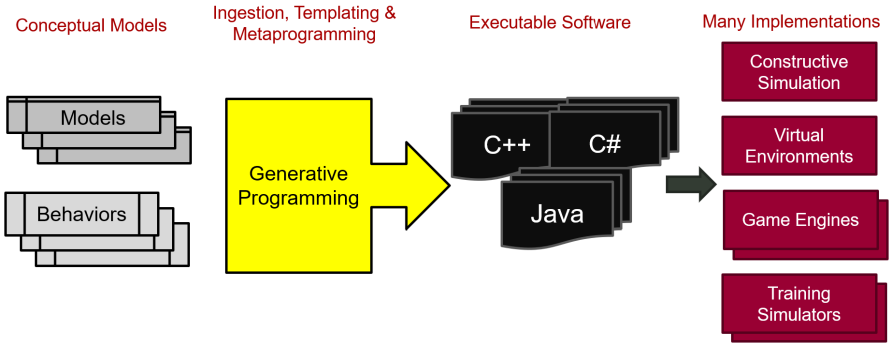

Fig. 5. Schematic representation of the generative programming architecture for simulation models.

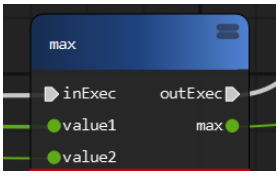

Fig. 6. A flow-based programming node.

the tool and associate that documentation to the nodes and data structures. We have also added an automated capability of exporting a Word document with the documentation and imagery of the graphs, keeping the models self-documenting. By keeping the explanation of how the model is supposed to work within the model development tool (like comments within software), there is more of a chance that the documentation keeps up to date rather than having documents separate from software implementations of the model. In addition to the Word document that gets generated, we also put the documentation text within comments in the generated software making it easier for integration software developers to understand what is being implemented. A screenshot impression of the integrated development environment is represented in Figure 7, showing a standard development environment around the central graph representation.

An important capability of the tool is to allow models to reference data. Datastores are a way to lookup data from a file, database, or potentially a data service. The goal is to make testing with the datastore easy for the user developing the graph, but allow the code generation and developers flexibility in how that data is queried. We created an editor for *Datastore* definitions in the *Types* editor, which allows the user to specify what data is available in the data store, what those types are, and to name and describe them in a typical user interface. The user can also specify *Queries* for that data, specify what the keys are for the lookup, and what values should be returned. Queries result in nodes that can be used in the graph to get results from the datastore as seen in Figure 7.

Once the model developers have created their model and described their data, they can use the visual programming tool to generate software, execute the software, and see the results return from a set of defined inputs. This allows the model developers to ensure that the model is working as they expect before involving simulation developers. Results can be graphed in many ways to visualize outputs. The test results are displayed in the bottom portion of Figure 7. There is a graph of the results that shows two strange oddities in the data. The model developer could recognize these oddities early in the development process and take corrective action before software developers even get involved. Issues can be found in the logic or data while developing the model which results in more accurate models, easier development, and better maintenance of models as logic changes or new data is used.

Finally, the logic and graphs developed in *PyFlow* can be exported to an intermediate format, that can be used to transfer the model from the front-end to the back-end.

B. STE Canonical Universal Format

The interchange format between the front-end and the back-end is based on XML documents, whose structure is defined by an XML Schema or *XSD (XML Schema Definition Language)*. This format structure is called the *Synthetic Training Environment (STE) Canonical Universal Format (SCUF)*.

This meta-model is not intended to support a full programming language, but rather to focus on the domain elements used within the U.S. Army's canonical descriptions of the simulation models. Nevertheless, it represents most concepts of a traditional procedural programming language. Specifically, these include the data type declarations, datastores, and various elements of algorithms, such as conditions, expressions and iterators. Moreover, the XML nature of the format means that it is easily extensible over time as long as the code generation tool is modified accordingly to handle any extended portions.

The logic and graphs developed in *PyFlow* are represented in the SCUF interchange format, and will ultimately be used by the code generation capability to create software. The aim was to provide an intermediary format between the visual development tool and the back-end code generation, in order to separate the two capabilities and to allow other future tools to output SCUF and still take advantage of the code generation capability without having to be compatible with *PyFlow*.

The SCUF meta-model is broken into two parts for ease of depiction. Figure 8 represents the elements related to

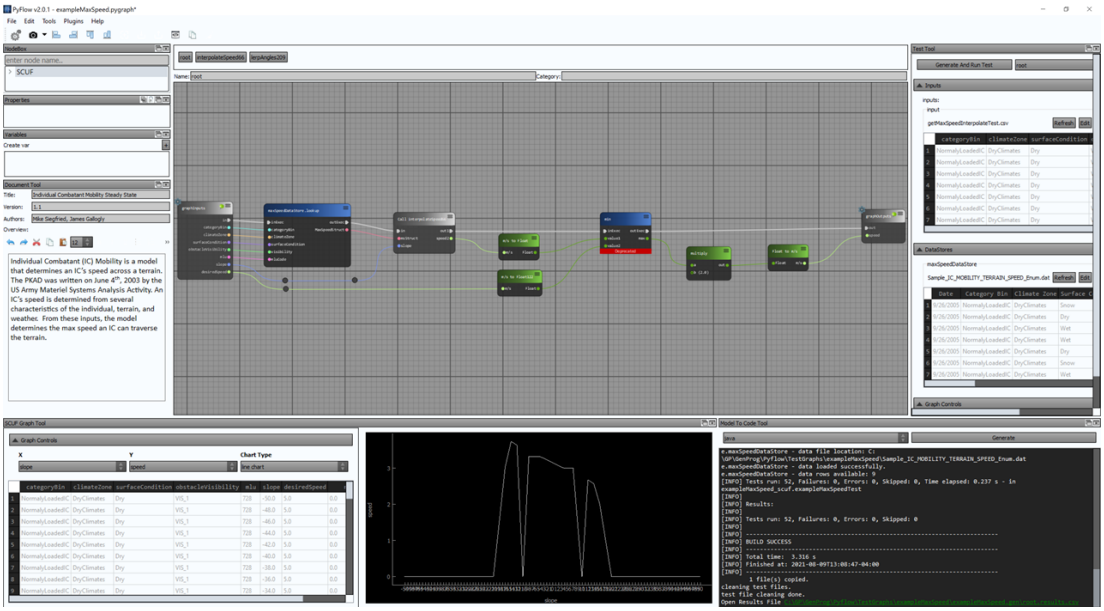

Fig. 7. A screenshot representing the PyFlow-based integrated development environment.

`TypeDefinition` and are static elements such as classes and types. The elements represented in Figure 9 are dynamic elements that describe modeling logic such as inputs, software structures, variables, and outputs. The structure and representation of the SCUF data models is similar to most standard *ERD (Entity Relationship Diagram)* visualizations. An example SCUF output is represented in Figure 10, where you can see the types, including type definitions, enumerations, classes, and datastores at the top of the file and then the dynamic logic in the bottom half of the XML document.

C. Back-End Code Generation

The code generation tool, the *Model To Code Tool (MTCT)* reads the SCUF model files into Java class representations in accordance with the meta-model above. Using the *Visitor pattern* [24] to process the model classes, it traverses the ingested model performing functions without impacting the model classes themselves. Once it is ensured that all dependencies are present and that the model is verified to be correct, the MTCT then uses a templating system to generate software. Currently, the *Apache Velocity* templating engine [25] is used.

Templates have been developed for three different programming languages: C++, C#, and Java. Each meta-model element corresponds to a template which handles the 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 referenced in the template using a modular structure. 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 approach is in accordance with the strategy of of *single sourcing* [20], similar to the NST approach, and reduces the number of templates that need to be implemented to support new languages or simulation environments. As the output from one template feeds into the input of another, we refer to this as a set of *cascading templates*. Figure 11 illustrates a simplified example of this process.

Reuse of existing templates allows us 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. MTCT utilizes a search path for locating the individual template files. It will look through the directories in the search path and use the first template it finds with the matching name. 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 override certain 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.

Control files 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. ASTs reduce the amount of integration work and code the developer must write to adapt generated models into a particular simulation system. ASTs are implemented by using Control Files for users to control how source code files are generated from the templates. They

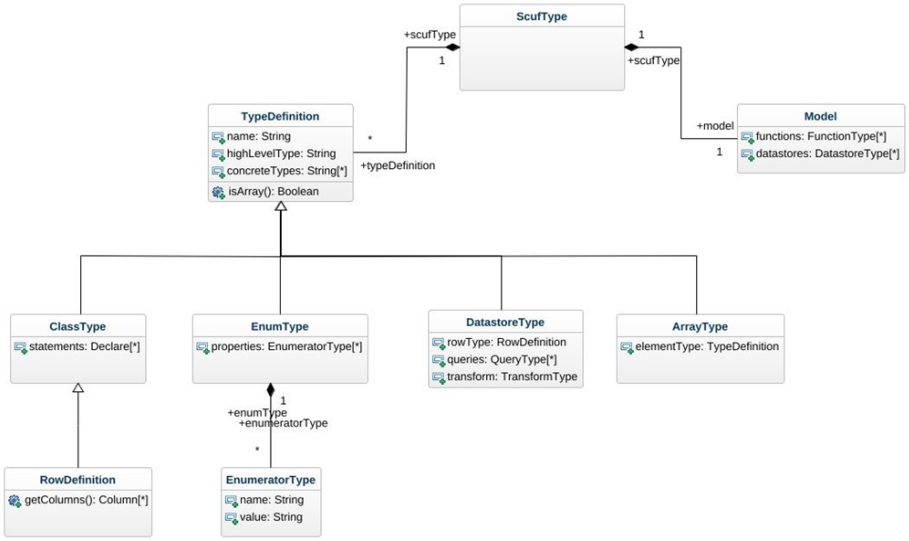

Fig. 8. A graphical representation of the part of the SCUF meta-model related to type definitions.

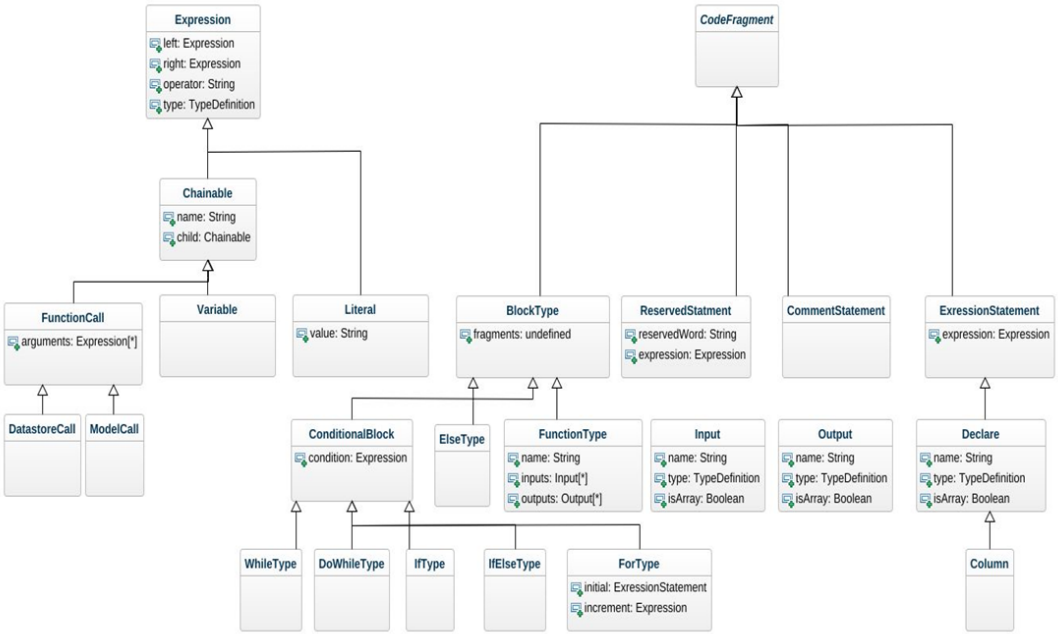

Fig. 9. A graphical representation of the part of the SCUF meta-model related to modeling logic.

allow the user to specify what parts of the SCUF model should be generated and how those parts will be 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 written to a single file or multiple files. A benefit of this approach is allowing for the differences in programming

```
<scuf name="MaxSpeedPKAD" version="Tue Nov 12 2019 09:27:30 GMT-0
  <types>
    <typedef name="degree" highlevelType="number" allowedConcrete
    <typedef name="m/s" highlevelType="number" allowedConcreteTyp
    <typedef name="MLUCode" highlevelType="number" allowedConcret
    <enum name="CategoryEnumBin" highlevelType="enum" allowedConc
      <property name="NormalyLoadedIC" val="13"/>
      <property name="FullyLoadedIC" val="14"/>
    </enum>
    <class name="MaxSpeedStruct">
      <statement name="categoryBin" type="CategoryEnumBin"/>
      <statement name="mluCode" type="MLUCode"/>
      <statement name="-40 Slope Speed" type="m/s"/>
      <statement name="Level Speed" type="m/s"/>
      <statement name="+40 Slope Speed" type="m/s"/>
    </class>
    <datastore highlevelType="datastore" name="maxSpeedDataStore"
      <keyset>
        <key name="categoryBin"/>
        <key name="mluCode"/>
      </keyset>
    </datastore>
  </types>
  <model>
    <function name="getMaxSpeed">
      <inputs>
        <input name="categoryBin" type="CategoryEnumBin"/>
        <input name="mlu" type="MLUCode"/>
        <input name="slope" type="degree"/>
        <input name="desiredSpeed" type="m/s"/>
      </inputs>
      <outputs>
        <output type="m/s"/>
      </outputs>
      <reserved-statement name="return">
        <function-call name="min">
          <arguments>
            <function-call name="interpolateSpeed">
              <arguments>
                <function-call name="lookup_maxSpeedDataStore">
                  <arguments>
                    <variable name="categoryBin"/>
                    <variable name="mlu"/>
                  </arguments>
                </function-call>
                <variable name="slope"/>
              </arguments>
            </function-call>
            <variable name="desiredSpeed"/>
          </arguments>
        </function-call>
      </reserved-statement>
    </function>
  </model>
</scuf>
```

Fig. 10. A sample of a SCUF XML file.

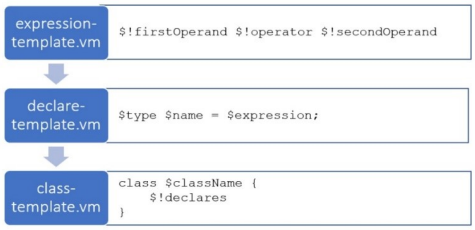

Fig. 11. An example of cascading templating.

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. Using control files, external function libraries, and custom template sets, users can write variation of these files to generate code specific to their own architecture, in the same way MTCT customizes generated code for the languages C++, Java, and C#.

Two additional use cases for code generation have been explored: generating a *Unity MonoBehavior* [26] as well as generating behaviors for *RIDE*, the Unity-based simulation simulation environment. To do this, a custom template needs to be written for the main model class, as well as Unity or RIDE specific configuration files in the project control section of the control file. A major challenge related to both 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. The 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. A solution is being investigated 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. 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 posed by the component modeling in Unity and RIDE are the methods that are meant to handle specific events within the framework. For example, Unity has a start method that runs during 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 a solution where we introduce 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 generation is executed.

V. Toward Integrating the Metaprogramming Environments

We have argued in Section II-C that both selected metaprogramming environments are well suited to be used as part of a representative case study for the horizontal integration of such

metaprogramming environments, and that the structured decoupling between the definition of models and the generation of code, a common characteristic of both models, is a good starting point for this integration. Moreover, the interchange format of the models in both environments is based on XML documents, whose structure is defined by an XML schema.

As the creation of a more horizontal integration architecture to facilitate the collaboration between metaprogramming environments [5] was one of the original goals of the NST metaprogramming environment, it seems logical to initiate this integration effort based on the NST environment architecture. This means that we attempt to map the generative programming environment for simulation models onto the collaboration architecture represented in Figure 3. In this section, we discuss some progress and remaining challenges.

A. Embracing the SCUF Meta-Model

The NST meta-circular metaprogramming environment [5] allows for the structural generation of all reader, writer, and model classes of any model —or meta-model— that can be expressed as a set of NST data elements. The SCUF meta-model, based on XML and defined by an XML Schema, satisfies this requirement. Based on the definition of the SCUF data entities (as represented in the class diagrams of Figures 8 and 9, e.g., *TypeDefinition*, *DatastoreType*, *ConditionalBlock*, *Expression*, *Declare*, *Statement*, etcetera), NST data elements can be created. For instance, *Input* needs to be defined as an NST data element with a *name* field which is a string, a *type* field that is a link to the *TypeDefinition* data element, and an *isArray* field that is a boolean. These data elements can be specified in XML, or in the user interface of the NST meta-application, or even directly generated from the XML Schema. For every data element, the various classes of the NST stack in the left part of Figure 3 can be generated. These include:

- Reader and writer classes to enable reading and writing the XML-based SCUF model files, e.g., *InputXmlReader* and *InputXmlWriter*.
- Model classes to represent and transfer the various SCUF entities, and to make them available as an object graph, e.g., *InputDetails* and *InputComposite*.
- View and control classes to perform *CRUDS (create, retrieve, update, delete, search)* operations in a generated table-based user interface.

This implies that the various existing SCUF models, representing instances of the SCUF data entities and therefore instances of the NST data elements, can be read and made available as an object graph, allowing to evaluate model parameters using *Object-Graph Navigation Language (OGNL)* expressions at the templating engine. Moreover, a NS web application with a table-based user interface can be generated to create, view, manipulate, and write SCUF models.

B. Integrating Modeling and Expansion

The meta-circular architecture of [5] enables the definition of alternative meta-models such as SCUF, and the development of new expanders based on the values and parameters of instances of these new models. Such alternative meta-models can be specified as any regular NS model, both in the NS Modeler and in the Prime Radiant. Upon defining an alternative meta-model such as SCUF, we are currently investigating two modes to provide integrated support for entering actual models based on the new meta-model and expanding these models.

- Based on the new meta-model, a slightly modified NS application is generated, dubbed *Secondary Radiant*, that allows to import/export the actual models from/to XML, and to pass them to the *Prime Radiant*. Importing expanders based on this new meta-model into the Prime Radiant then enables developers to invoke these expanders from the Prime Radiant.
- A runtime kernel, dubbed *Runtime Radiant*, is provided that allows regular NS applications to invoke the templating engine and to evaluate OGNL expressions for arbitrary trees of data objects. In this way, a regular NS application generated based on the new meta-model is able to perform expansion from its actual model data.

Both types of tooling are currently being tested in β-version, and will possibly merge into one solution.

Though conceptually agnostic with respect to different meta-models, a bias toward the web information systems was discovered in the NST metaprogramming environment. Both the invocation of expansion and deployment, and the various technology settings were implicitly linked to the entities *Application* and *Component*. As these enities are specific to web-based information systems, they have been generalized to *ProgramType* and *ModuleType* in the NST environment. At the same time, dedicated meta-elements to specicy various technologies, such as *PresentationLogicSettings*, have been generalized to a generic list of *TechnologyStackSettings*.

C. Streamlining the Control Files

Having defined the SCUF data entities as NST data elements, the NST metaprogramming environment allows to evaluate SCUF model parameters through OGNL expressions in SCUF model graphs, and to make them available to coding templates. In order to simply activate the existing coding templates of the simulation models, and to use the NST metaprogramming environment as a piece of evolvable middleware to pass the SCUF models to the code templates for the simulation models, two tasks remain to be performed at the level of the declarative control.

- Every coding template needs to be declared in a separate XML *Expander* definition.
- For every coding template, the appropriate OGNL expressions to evaluate the relevant model parameters, need to be defined in an XML *Mapping* file.

As the generative programming environment for simulation models has control files as well, they are a solid starting point to create these declarative control files. It is probably even possibe to write software that automates the conversion of

these JSON control files into the XML control files of the NST metaprogramming environment.

D. Supporting the Templating Engine

The fact that both metaprogramming environments use different templating engines causes a final integration issue. A first option would be to convert the *Velocity* templates of the simulation software to the *StringTemplate* format supported by the NST environment. In this scenario, the required effort would be proportional to the template base of the simulation models, and would need to be repeated for integration efforts with other environments using this templating engine. Moreover, *Velocity* templates allow more logic that would have to be ported to Java helper classes in the *StringTemplate* environment.

A second and preferable option is to include support in the NST metaprogramming environment for the *Velocity* templating engine. Considering the limited amount of templating engines being used by metaprogrammers, this scenario seems both manageable and worthwhile. Moreover, the effort would not be proportional to the size of the template base. As there is virtually no logic in the current NST templates, i.e., all model parameters are combined and processed in the software that feeds the templating engine, it is reasonable to say that we expect no major blocking issues.

Important to note is that both metaprogramming environments use modular or so-called cascading templates in accordance with the strategy of single-sourcing. This means that the modular structures of the templates could be preserved identically across the different templating engines.

VI. Conclusion

The automated generation of source code, often referred to as metaprogramming, has been pursued for decades in computer programming, and is considered to entail significant benefits for various disciplines, including software development, systems engineering, modeling, simulation, and business process design. However, we have argued that metaprogramming is still facing several issues, including the fact that it is challenging to realize a scalable collaboration within and between different metaprogramming environments, largely due to the often vertical integration architecture.

In our previous work, we have presented a meta-circular implementation of a metaprogramming environment, and have argued that this architecture enables a scalable collaboration, both within this environment and possibly with other metaprogramming environments. In this paper, we have explored such a collaborative integration with another metaprogramming environment. This second environment for metaprogramming targets the generation of a different type of software systems, and is based on a different meta-model, but exhibits a more horizontal integration architecture as well. This second metaprogramming architecture has been described in detail.

We have shown in this contribution how both metaprogramming environments can be integrated within the proposed meta-circular architecture. We have explained how the generation of the meta-code, i.e., the code that makes the actual parameter models available to the coding templates, can be extended to the second metaprogramming environement, resulting even in tooling that provides integrated support both modeling and expansion or code generation. We have also explained that the only reason that the actual code generation of this second metaprogramming environment cannot be seamlessly integrated yet, is the different format of the generation control files and the use of another templating engine. However, we have also indicated that it should be relatively straightforward to support, possibly even in an automated way, such an alternative control format and/or templating engine.

This paper is believed to make some contributions. First, we show in a constructive way that it is possible to perform an horizontal integration of two metaprogramming environments, and to enable collaboration and re-use between these environments. Such integrations could significantly improve the collaboration and productivity at the metaprogramming level. Moreover, we show that this integration is possible between metaprogramming environments that are based on completely different meta-models, are significant in size, and are being developed and used by application developers on a continuous basis. Second, we explain how the horizontal integration of a second metaprogramming environment with the meta-circular architecture, could largely remove the burden of maintaining the internal classes of such a metaprogramming environment.

Next to these contributions, it is clear that this paper is also subject to a number of limitations. It consists of a single case of integrating a second metaprogramming environment with the meta-circular architecture, although the case deals with two realistic and comprehensive development environments. Moreover, the presented results are still preliminary, and the second metaprogramming environment is not yet operational in the meta-circular architecture, as its control mechanism and templating engine is not yet fully supported in this architecture. Therefore, neither the complete horizontal integration, nor the productive collaboration between the two environments has been completely proven. However, this explorative but nevertheless representative case study can be regarded as an architectural pathfinder, and we have identified some remaining issues that hamper the scalability of the approach.

To further enhance the scalability of the approach, it is imperative to streamline and support the automated exchange of domain models and the corresponding meta-models. We are therefore working on enhanced tooling to allow metaprogrammers to easily define their existing meta-models. Based on these definitions of meta-models, the tooling should be able to extend itself, and to include support for the actual models that are based on these meta-models. The goal is to enable entering, manipulating and viewing these models, and to provide the automatic creation of standardized model data trees and/or control files that can be fed into the various tem-

plating engines. Besides addressing the issues that currently hamper the scalability of the approach, we have also initiated a collaboration with a third metaprogramming environment.

REFERENCES

[1] H. Mannaert, C. McGroarty, K. De Cock, and S. Gallant, "Integrating two metaprogramming environments: An explorative case study," in Proceedings of the Fifteenth International Conference on Software Engineering Advances (ICSEA) 2020, 2020, pp. 166–172.

[2] J. R. Rymer and C. Richardson, "Low-code platforms deliver customer-facing apps fast, but will they scale up?" Forrester Research, Tech. Rep., 08 2015.

[3] B. Reselman, "Why the promise of low-code software platforms is deceiving," TechTarget, Tech. Rep., 05 2019.

[4] H. Mannaert, K. De Cock, and P. Uhnak, "On the realization of meta-circular code generation: The case of the normalized systems expanders," in Proceedings of the Fourteenth International Conference on Software Engineering Advances (ICSEA) 2019, 2019, pp. 171–176.

[5] H. Mannaert, K. De Cock, P. Uhnak, and J. Verelst, "On the realization of meta-circular code generation and two-sided collaborative metaprogramming," International Journal on Advances in Software, no. 13, 2020, pp. 149–159.

[6] D. Parnas, "Software aspects of strategic defense systems," Communications of the ACM, vol. 28, no. 12, 1985, pp. 1326–1335.

[7] P. Cointe, "Towards generative programming," Unconventional Programming Paradigms. Lecture Notes in Computer Science, vol. 3566, 2005, pp. 86–100.

[8] K. Czarnecki and U. W. Eisenecker, Generative programming: methods, tools, and applications. Reading, MA, USA: Addison-Wesley, 2000.

[9] L. Tratt, "Domain specific language implementation via compile-time meta-programming," ACM Transactions on Programming Languages and Systems, vol. 30, no. 6, 2008, pp. 1–40.

[10] A. Wortmann, "Towards component-based development of textual domain-specific languages," in Proceedings of the Fourteenth International Conference on Software Engineering Advances (ICSEA) 2019, 2019, pp. 68–73.

[11] K. Gusarovs and O. Nikiforova, "An intermediate model for the code generation from the two-hemisphere model," in Proceedings of the Fourteenth International Conference on Software Engineering Advances (ICSEA) 2019, 2019, pp. 74–82.

[12] C. Mooers and L. Deutsch, "Trac, a text-handling language," in ACM '65 Proceedings of the 1965 20th National Conference, 1965, pp. 229–246.

[13] D. McIlroy, "Macro instruction extensions of compiler languages," Communications of the ACM, vol. 3, no. 4, 1960, pp. 214–220.

[14] J. Reynolds, "Definitional interpreters for higher-order programming languages," Higher-Order and Symbolic Computation, vol. 11, no. 4, 1998, pp. 363–397.

[15] M. Vonderembse, T. Raghunathan, and S. Rao, "A post-industrial paradigm: To integrate and automate manufacturing." International Journal of Production Research, vol. 35, no. 9, 1997, p. 2579–2600.

[16] H. Mannaert, J. Verelst, and P. De Bruyn, Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design. Koppa, 2016.

[17] H. Mannaert, J. Verelst, and K. Ven, "The transformation of requirements into software primitives: Studying evolvability based on systems theoretic stability," Science of Computer Programming, vol. 76, no. 12, 2011, pp. 1210–1222, special Issue on Software Evolution, Adaptability and Variability.

[18] ——, "Towards evolvable software architectures based on systems theoretic stability," Software: Practice and Experience, vol. 42, no. 1, 2012, pp. 89–116.

[19] P. De Bruyn, H. Mannaert, J. Verelst, and P. Huysmans, "Enabling normalized systems in practice : exploring a modeling approach," Business & information systems engineering, vol. 60, no. 1, 2018, pp. 55–67.

[20] K. Ament, Single Sourcing: Building Modular Documentation. Norwich, NY, USA: William Andrew Publishing, 2003.

[21] J. P. Morrison, Flow-Based Programming: A New Approach to Application Development. Van Nostrand Reinhold, 1994.

[22] M. Senthilvel and J. Beetz, "A visual programming approach for validating linked building data," URL: https://publications.rwth-aachen.de/record/795561/files/795561.pdf, 2022, [accessed: 2022-06-15].

[23] B. Rearick, Blockly. Cherry Lake Publishing, 2017.

[24] E. Gamma, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[25] "The Apache Velocity Project," URL: https://velocity.apache.org/, 2022, [accessed: 2022-06-15].

[26] "How to make a video game without any coding experience," URL: https://unity.com/how-to/make-games-without-programming, 2022, [accessed: 2022-06-15].