Integrating Two Metaprogramming Environments: An Explorative Case Study

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.de.cock@nsx.normalizedsystems.org

***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. 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

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 [1][2]. 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 [3], 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 suited for this study, as they both pursue a more horizontal integration architecture.

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. In Section III, we explain the need for collaborative metaprogramming and the issues that need to be solved. Section IV presents the architecture and meta-model of both metaprogramming environments whose integration is explored in this contribution. Section V elaborates on the possible integration of these metaprogramming environments, detailing the possibilities, progress, and remaining issues. Finally, we present some conclusions in Section VI.

II. Metaprogramming, Meta-Circularity, and Systems Integration

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 [4]. *Generative programming*, ”to manufacture software components in an automated way” [5], emphasizes the manufacturing aspect and the similarity to production and the industrial revolution. *Metaprogramming*, sometimes described as a programming technique in which

 ISBN: 978-1-61208-827-3

”computer programs have the ability to treat other programs as their data” [6], 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., [7], 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)*, 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 [1], while others even question whether these platforms actually make development cheaper or easier [2]. Moreover, defining an intermediate representation or reusing DSLs is still a subject of research today. We mention the contributions of Wortmann [8], presenting a novel conceptual model for the systematic reuse of DSLs, and Gusarov et. al. [9], 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 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 [10], and to similar concepts in an earlier paper from McIlroy [11]. Meta-circularity, first coined by Reynolds describing his meta-circular interpreter [12], 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, as they could enable a unified view on both the metaprogramming code and the generated source code, thereby reducing the complexity for the metaprogrammers.

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 [13], systems integration at the metaprogramming level could provide value to their customers, i.e., the software developers.

III. Toward Scalable Collaborative Metaprogramming

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,

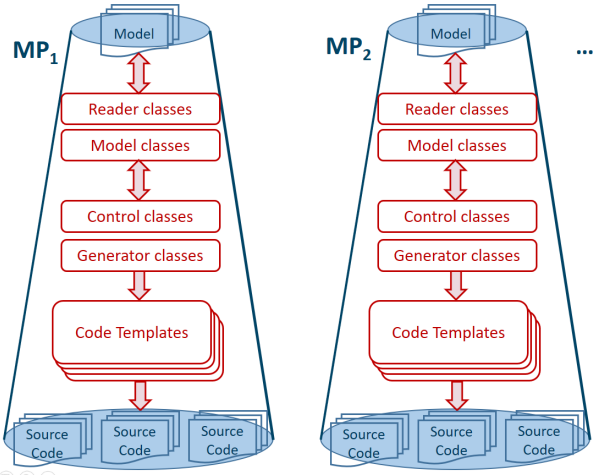

Fig. 1. Representation of the duplication of metaprogramming silos.

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.

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

 ISBN: 978-1-61208-827-3

techniques, some fundamental issues need to be addressed [14][3]. 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 [14]. Second, to avoid the growing burden of maintaining the often complex meta-code and continuously adapting it to new technologies, we have proposed a meta-circular architecture to regenerate the metaprogramming code itself as well [3]. 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 [3], 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 two metaprogramming environments. Exploring such a collaboration is the purpose of the case study in this paper.

IV. Structure of the Metaprogramming Environments

In this section, we present the architectures and meta-models of the two metaprogramming environments considered in this integration case study. The first metaprogramming environment is the NST meta-circular architecture, as it explicitly aims to realize horizontal integration and scalable collaboration. The 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.

A. Normalized Systems Elements Metaprogramming

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 [14][15][16]. 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 [14], 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 [3]. This meta-circular architecture, described in [3] and schematically represented in Figure 2, enables both the regeneration of the metaprogramming code

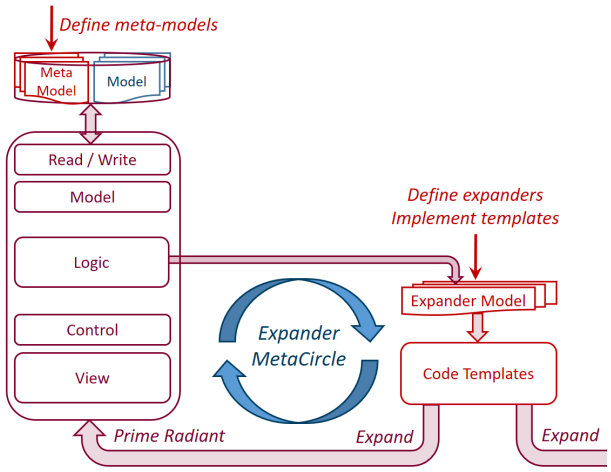

Fig. 2. Closing the meta-circle for expanders and meta-application.

itself, and allows for a structural decoupling between the two sides of the transformation, i.e., the domain models and the code generating templates.

The domain models for the web-based information systems are specified as sets of instances of the various types of NS elements. While these elements can be entered in a meta-application and/or graphical modeler, they are formally specified in XML files, whose structure is defined in a corresponding XML Schema.

As the NS meta-model is just another NS model [3], the various types of elements 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.

 ISBN: 978-1-61208-827-3

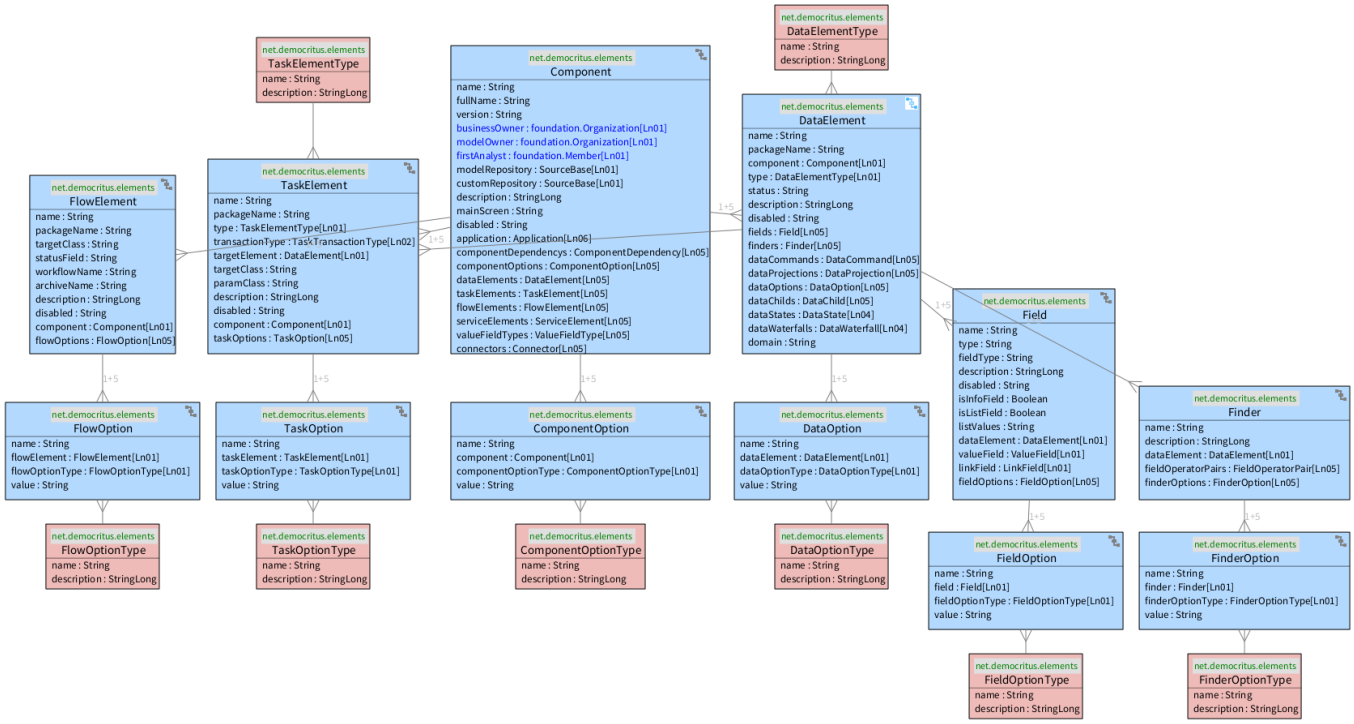

Fig. 3. A graphical representation of the core part the NS (data) meta-model.

The core data model of this metaprogramming environment is represented in Figure 3. This graphical representation, a screenshot from the *NST Modeler* tool, is similar to most *ERD (Entity Relationship Diagram)* visualizations, but uses colors to distinguish between different types of data entities [17]. The unit of an NS model is a *component*, and within such a component model, we distinguish the various types of NS elements [14], 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 characterized by a typing or taxonomy entity, e.g., *Task element type* or *Task option type*, represented in light red. The data elements contain a numer of attributes or *Fields*, where a field can be either a data attributes or a relationship link, and provide a number of *Finders*. Both fields and finders can have options characterized by corresponding option types.

Every individual code generator or *NS expander* is declared in an `Expander` XML file, specifying for instance the type of element it belongs to, and the various properties of the source artifact that it generates. For every such artifact expander, one needs to provide a coding `Template`, based on the *StringTemplate (ST)* engine library, and an XML expander `Mapping` file, specifying the various template parameters in terms of model parameters through *Object-Graph Navigation Language (OGNL)* expressions.

B. Generative Programming of Simulation Models

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 examined, 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.

The architecture of this metaprogramming environment, schematically represented in Figure 4, 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

 ISBN: 978-1-61208-827-3

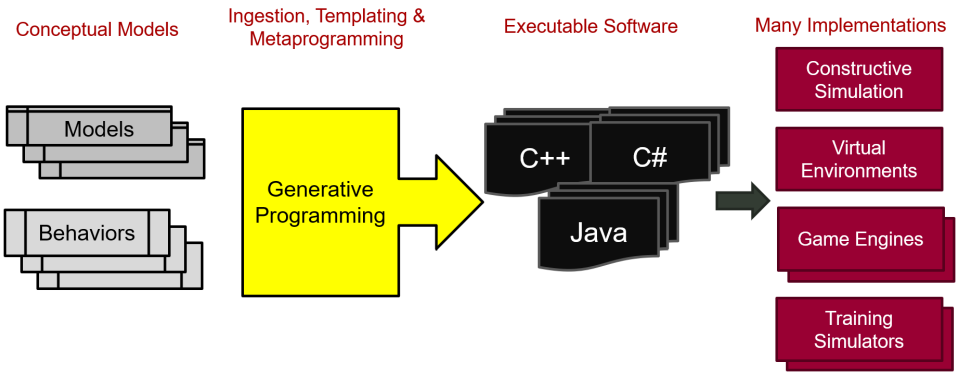

Fig. 4. Schematic representation of the generative programming architecture for simulation models.

suits the needs of the front-end user community, and to pass those models to be used for code generation in the back-end.

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 interchange format structure, i.e., the XSD, 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. Figure 5 provides a class diagram of the SCUF meta-model, anew similar to most ERD visualizations.

To capture the human readable text of the canonical simulation model descriptions along with executable code in the front-end, the generative programming environment uses *PyFlow* [18], which is an open source project that is similar to other visual scripting frameworks including Unity's Bolt or Playmaker [19], and Unreal's Blueprints [20]. The U.S. Army added custom additions to PyFlow that includes both the ability to execute the models, as well as the capability to generate the *SCUF code*, the interchange format to transfer the model from the front-end to the back-end. The back-end code generator uses the *Apache Velocity* templating engine to create the output files in multiple programming languages (C#, C++, and Java currently).

V. Toward Integrating the Metaprogramming Environments

The two metaprogramming environments target the automatic programming of two 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. Moreover, both the front-end technologies capturing the models, and the target programming languages —even the code templating engines— are different.

What both metaprogramming environments have in common is a structured decoupling between the definition of models and the generation of code. Moreover, the interchange format of the models is in both environments based on XML documents, whose structure is defined by an XML schema. This means that it is conceptually possible to map the generative programming environment for simulation models onto the collaboration architecture represented in Figure 2.

A. Embracing the SCUF Meta-Model

The NST meta-circular metaprogramming environment 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 diagram of Figure 5, e.g., *TypeDefinition*, *DatastoreType*, *ConditionalBlock*, *Expression*, *Declare*, *Statement*, etcetera), NST data elements can be created. For instance, *Statement* 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 *expression* field that is a link to the *Expression* data element. 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 2 can be generated. These include:

- Reader and writer classes to read and write the XML-based SCUF files, e.g., *StatementXmlReader* and *StatementXmlWriter*.
- Model classes to represent and transfer the various SCUF entities, and to make them available as an object graph, e.g., *StatementDetails* and *StatementComposite*.
- View and control classes to perform *CRUDS (create, retrieve, update, delete, search)* operations in a generated table-based user interface.

 ISBN: 978-1-61208-827-3

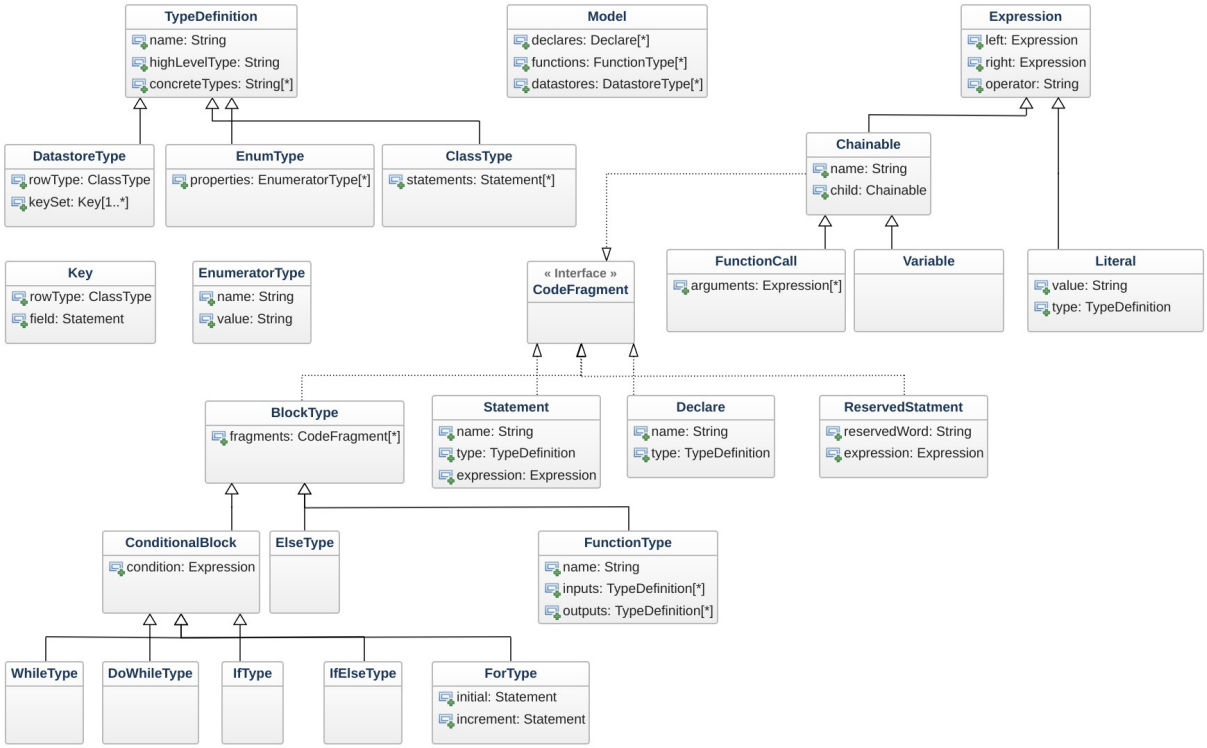

Fig. 5. A graphical representation of the core part the SCUF (data) meta-model.

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 emplating engine. Moreover, an additional application with a table-based user interface is available to create, view, manipulate, and write SCUF models.

B. Supporting the Templating Engine

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.

- 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.

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. And 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.

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 due, to the often vertical integration architecture.

 ISBN: 978-1-61208-827-3

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 also exhibits a more horizontal integration architecture.

We have shown in this contribution how both metaprogramming environments can be integrated within the proposed meta-circular architecture, by extending the generation of the meta-code, i.e., the code that makes the actual parameter models available to the coding templates, to the second metaprogramming environement. We have explained that the only reason that the coding templates of this second metaprogramming environment cannot be seamlessly integrated yet, is that they use another templating engine. However, we have also indicated that it should be relatively straightforward to support this alternative templating engine.

This paper is believed to make some contributions. First, we show 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. Second, we explain that the horizontal integration of a second metaprogramming environment with the meta-circular architecture, could largely remove the burden of maintaining the internal classes of this 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. Moreover, the presented results are quite preliminary, and the second metaprogramming environment is not yet operational in the meta-circular architecture, as its templating engine is not yet supported in this architecture. Therefore, neither the complete horizontal integration, nor the productive collaboration between the two environments has actually been proven. However, this explorative case study can be seen as an architectural pathfinder, and we are planning to both broaden and deepen the collaboration on the horizontal integration of different metaprogramming environments.

REFERENCES

[1] 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.

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

[3] 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.

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

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

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

[7] L. Tratt, "Domain specific language implementation via compile-time meta-programming," ACM transactions on programming languages and system, vol. 30, no. 6, 2008, pp. 1–40.

[8] 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.

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] M. Senthilvel and J. Beetz, "A visual programming approach for validating linked building data." [Online]. Available: https://publications.rwth-aachen.de/record/795561/files/795561.pdf

[19] "How to make a video game without any coding experience." [Online]. Available: https://unity.com/how-to/make-games-without-programming

[20] B. Sewell, Blueprints Visual Scripting for Unreal Engine. Packt Publishing, 2015.

 ISBN: 978-1-61208-827-3