On the Realization of Meta-Circular Code Generation and Two-Sided Collaborative Metaprogramming

Herwig Mannaert
Normalized Systems Institute
University of Antwerp
Antwerp, Belgium
herwig.mannaert@uantwerp.be

Koen De Cock
Research & Development
NSX bv
Niel, Belgium
koen.de.cock@nsx.normalizedsystems.org

Peter Uhnák
Research & Development
NSX bv
Niel, Belgium
peter.uhnak@nsx.normalizedsystems.org

Jan Verelst
Normalized Systems Institute
University of Antwerp
Antwerp, Belgium
jan.verelst@uantwerp.be

***Abstract*—The automated generation of source code is a widely adopted technique to improve the productivity of computer programming. Normalized Systems Theory (NST) aims to create software systems exhibiting a proven degree of evolvability. A software implementation exists to create skeletons of Normalized Systems (NS) applications, based on automatic code generation. This paper describes how the NS model representation, and the corresponding code generation, has been made meta-circular, and presents its detailed architecture. It is argued that this feature may be crucial to improve the productivity of metaprogramming, as it enables scalable collaboration based on two-sided interfaces. Some preliminary results from applying this approach in practice are presented and discussed.**

***Index Terms*—Evolvability, meta-circularity, normalized systems, automatic programming; case study**

I. Introduction

This paper extends a previous paper that was originally presented at the Fourteenth International Conference on Software Engineering Advances (ICSEA) 2019 [1].

Increasing the productivity in computer programming has been an important and long-term goal of computer science. Though many different approaches have been proposed, discussed, and debated, two of the most fundamental approaches toward this goal are arguably automated code generation and homoiconic programming. Increasing the evolvability of Information Systems (IS) on the other hand, is crucial for the productivity during the maintenance of information systems. Although it is even considered as an important attribute determining the survival chances of organizations, it has not yet received much attention within the IS research area [2]. Normalized Systems Theory (NST) was proposed to provide an ex-ante proven approach to build evolvable software by leveraging concepts from systems theory and statistical thermodynamics. In this paper, we present an integrated approach that combines both Normalized Systems Theory to provide improved evolvability, and automated code generation and homoiconic programming to offer increased productivity. We also argue that this combined approach can enable entirely new levels of productivity and scalable collaboration.

The remainder of this paper is structured as follows. In Section II, we briefly discuss two fundamental approaches to increase the productivity in computer programming: automatic and homoiconic programming. In Section III, we give an overview of NST as a theoretical basis to obtain higher levels of evolvability in information systems, and discuss the NST code generation or expansion. Section IV presents the realization of the meta-circular metaprogramming architecture, and details the declarative control structure. Section V elaborates on the possibilities that the two-sided interfaces of the metaprogramming architecture offer for scalable collaboration. Finally, we report and discuss some results in Section VI, and present our conclusions in Section VII.

II. Automatic and Homoiconic Programming

A. Automatic or Metaprogramming

The automatic generation of code is nearly as old as coding or software programming itself. One often makes a distinction between *code generation*, the mechanism where a compiler generates executable code from a traditional high-level programming language, and *automatic programming*, the act of automatically generating source code from a model or template. In fact, one could argue that both mechanisms are quite similar, as David Parnas already concluded in 1985 that ”automatic programming has always been a euphemism for programming in a higher-level language than was then available to the programmer” [3]. In general, automatic programming performs a transformation from domain and/or intermediate models to programming code.

Another term used to designate automatic programming is *generative programming*, aiming to write programs "to manufacture software components in an automated way" [4], in the same way as automation in the industrial revolution has improved the production of traditional artifacts. As this basically corresponds to an activity at the meta-level, i.e., writing software programs that write software programs, this is also referred to as *metaprogramming*. Essentially, the goal of automatic programming is and has always been to improve programmer productivity.

Software development methodologies such as *Model-Driven Engineering (MDE)* and *Model-Driven Architecture (MDA)*, focusing on creating and exploiting conceptual domain models and ontologies, are also closely related to automatic programming. In order to come to full fruition, these methodologies require the availability of tools for the automatic generation of source code. Currently, these model-driven code generation tools are often referred to as *Low-Code Development Platforms (LCDP)*, i.e., software that provides an environment for programmers to create application software through graphical user interfaces and configuration instead of traditional computer programming. As before, the goal remains to increase the productivity of computer programming.

The field is still evolving while facing various challenges and criticisms. Some question whether low-code development platforms are suitable for large-scale and mission-critical enterprise applications [5], while others even question whether these platforms actually make development cheaper or easier [6]. Moreover, defining an intermediate representation or reusing *Domain Specific Languages (DSLs)* is still a subject of research today. We mention the contributions of Wortmann [7], presenting a novel conceptual approach for the systematic reuse of *Domain Specific Languages*, Gusarov et al. [8], proposing an intermediate representation to be used for code generation, and Frank [9], pleading for multi-level modeling. Hutchinson et al. elaborate on the importance of organizational factors for code generation adoption [10], and suggest that the benefit of model-driven development has to be found in a more holistic approach to software architecture [11]. We have argued in our previous work that some fundamental issues need to be addressed, like the increasing complexity due to changes during maintenance, and have proposed to combine automatic programming with the evolvability approach of *Normalized Systems Theory (NST)* [12].

B. Homoiconicity or Meta-Circularity

Another technique in computer science aimed at the increase of the abstraction level of computer programming, thereby aiming to improve the productivity, is homoiconicity. A language is homoiconic if a program written in it can be manipulated as data using the language, and thus the program's internal representation can be inferred just by reading the program itself. As the primary representation of programs is also a data structure in a primitive type of the language itself, reflection in the language depends on a single, homogeneous structure instead of several different structures. It is this language feature that can make it much easier to understand how to manipulate the code, which is an essential part of metaprogramming. The best known example of an homoiconic programming language is Lisp, but all Von Neumann architecture systems can implicitly be described as homoiconic. An early and influential paper describing the design of the homoiconic language TRAC [13], traces the fundamental concepts back to an even earlier paper from McIlroy [14].

Related to homoiconicity is the concept of a *meta-circular evaluator (MCE)* or *meta-circular interpreter (MCI)*, a term that was first coined by Reynolds [15]. Such a meta-circular interpreter, most prominent in the context of Lisp as well, is an interpreter which defines each feature of the interpreted language using a similar facility of the interpreter's host language. The term meta-circular clearly expresses that there is a connection or feedback loop between the activity at the meta-level, the internal model of the language, and the actual activity, writing models in the language.

There is a widespread belief that this kind of properties increase the abstraction level and therefore the productivity of programming. We will argue that this is even more relevant for automatic programming, as the metaprogramming code, i.e., the programming code generating the code, is often complex and therefore hard to maintain. Moreover, the potential of meta-circularity with respect to productivity can be seen in the context of other technologies. For instance, a transistor is a switch that can be switched by another transistor. Therefore, when a smaller and faster transistor/switch is developed, there is no need to develop a new version of the switching device, as such a new version of this device, i.e., the transistor itself, is already there, and smaller and faster as well. Such a shortcut of the design cycle can clearly foster rapid progress.

III. Normalized Systems Theory and Expansion

In this section, we discuss the code generation or *expansion* based on Normalized Systems Theory, attempting to address some fundamental issues that automatic programming is facing today: the lack of evolvability in information systems, and the increasing complexity due to changes.

A. Evolvability and Normalized Systems

The evolvability of information systems (IS) is considered as an important attribute determining the survival chances of organizations, although it has not yet received much attention within the IS research area [2]. Normalized Systems Theory (NST), applying the concept of *stability* from systems theory to the design cycle of information systems, was proposed to provide an ex-ante proven approach to build evolvable software [12], [16], [17]. Systems theoretic stability is an essential property of systems, and means that a bounded input should result in a bounded output. In the context of information systems development and evolution, this implies that a bounded set of changes should result in a bounded set of impacts to the software. Put differently, it is demanded that the impact of changes to an information system should

not be dependent on the size of the system to which they are applied, but only on the size of the changes to be performed. Changes rippling through and causing an impact dependent on the size of the system are called *combinatorial effects*, and are considered to be a major factor limiting the evolvability of information systems. The theory prescribes a set of theorems, and formally proves that any violation of any of the following *theorems* will result in combinatorial effects, thereby hampering evolvability [12], [16], [17]:

- *Separation of Concerns:*
  Every concern, defined as an independent change driver, should be separated in its own class or module.
- *Action Version Transparency:*
  Every computing action should be encapsulated to shield other modules from internal implementation changes.
- *Data Version Transparency:*
  Every data structure should be encapsulated to shield modules passing this data from internal data changes.
- *Separation of States:*
  Every result state from a computing action should be stored, to shield other modules from having to deal with new types of implementation errors.

The application of the theorems in practice has shown to result in very fine-grained modular structures within a software application. In particular, the so-called cross-cutting concerns, i.e., concerns cutting across the functional structure, need to be separated as well. This is schematically represented for three domain entities ('Order', 'Invoice', and 'Payment'), and three cross-cutting concerns ('Persistency', 'Access Control', and 'Remote Access') in Figure 1. Though the actual implementation of the cross-cutting concern is in general provided by an external framework (represented by the colored planes), the code connecting to that framework (represented by the small colored disks), is considered to be a change driver, and needs to be separated and properly encapsulated.

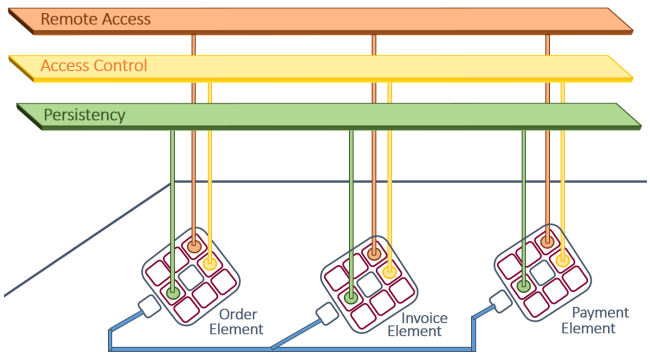

Fig. 1. Representation of domain entities connecting to external frameworks.

As every domain entity needs to connect to various additional external frameworks providing cross-cutting concerns, e.g., 'Transaction' or 'REST Service', these entities need to be implemented by a *set of classes*. Such a set of classes, schematically represented in Figure 2 for the domain entity 'Invoice', is called an *Element* in NST.

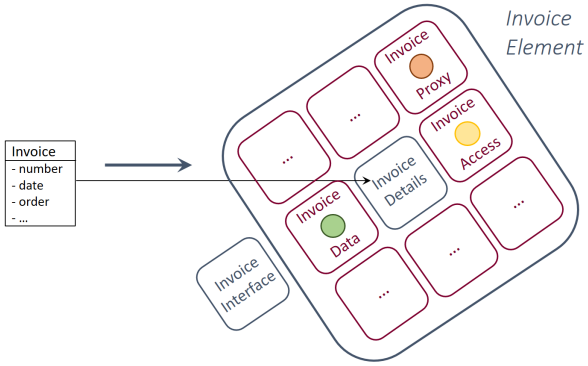

Fig. 2. Representation of an *element* for the domain entity 'Invoice'.

Such structures are, in general, difficult to achieve through manual programming. Therefore, the theory also proposes a set of patterns to generate significant parts of software systems which comply with these theorems. More specifically, NST defines five types of *elements* to provide the main functionality for information systems, and proposes five detailed design patterns to implement these element structures [17] [12]:

- *Data element* to represent a data or domain entity.
- *Action element* to implement a computing action or task.
- *Workflow element* to orchestrate a flow or state machine.
- *Connector element* to provide a user or service interface.
- *Trigger element* to perform a task or flow periodically.

The implementation or instantiation of the element structures results in a codebase with a highly recurring structure. Such a recurring structure is desirable as it *increases the consistency*, and *reduces the complexity* of the codebase. However, this recurring structure will have to be adapted over time based on new insights, the discovery of flaws, and/or changes in underlying technologies or frameworks. These structural changes may need to be applied in a retroactive way, but the efforts increase with the frequency of these adaptations. For instance, if one decides to adapt or refactor the element structures in a growing system in a retroactive way every time K additional elements have been created, the total amount of refactored element structures when the system reaches N different elements, will be equal to:

$$K + 2K + ... + N = \sum_{i=1}^{N/K} i.K = \frac{N(N+K)}{2K} \qquad (1)$$

Therefore, the element structures of NST software are generated *and regenerated* in a —rather straightforward— automated way. First, a model of the considered universe of discussion is defined in terms of a set of data, task and workflow elements. Next, code generation or automated programming is used to generate parametrized copies of the general element design patterns into boiler plate source code. Due to the simple and deterministic nature of this code generation mechanism, i.e., instantiating parametrized copies, it is referred to as *NS expansion* and the generators creating the individual coding artifacts are called *NS expanders*. This generated code can,

in general, be complemented with custom code or *craftings* to add non-standard functionality that is not provided by the expanders themselves, at specific places within the boiler plate code marked by *anchors*. This custom code can be automatically *harvested* from within the anchors, and *re-injected* when the recurring element structures are regenerated.

B. Expansion and Variability Dimensions

In applications generated by a Normalized Systems (NS) expansion process, schematically represented in Figure 3 around the symbolic blue icon, we identify four variability dimensions. As discussed in [18] [19], the combination of these dimensions compose an actual NS application codebase, represented in the lower right of Figure 3, and therefore determine how such an application can evolve through time, i.e., how software created in this way exhibits evolvability.

First, as represented at the upper left of the figure, one should specify or select the *models* or *mirrors* he or she wants to expand. Such a model is technology agnostic (i.e., defined without any reference to a particular technology that should be used) and represented by standard modeling techniques, such as ERD's for data elements and BPMN's for task and flow elements. Such a model can have multiple versions throughout time (e.g., being updated or complemented) or concurrently (e.g., choosing between a more extensive or summarized version). As a consequence, the chosen model represents a first dimension of variability or evolvability.

Second, as represented on top of the blue icon in the figure, one should provide the parametrized *coding templates* for the various classes of the elements according to a specific element structure as represented in Figure 2. The *expanders* will generate (boiler plate) source code by instantiating the various class templates or *skeletons* of the element structures, i.e., the design patterns, taking the specifications of the model as *parameters*. For instance, for a data element 'Invoice', a set of java classes *InvoiceDetails*, *InvoiceProxy*, *InvoiceData*, *InvoiceAccess*, etcetera will be generated. This code can be considered boiler plate code as it provides a set of standard functionalities for each of the elements within the model, though they have evolved over time to provide features like standard finders, master-detail (waterfall) screens, certain display options, document upload/download functionality, child relations, etcetera. The expanders and corresponding template skeletons evolve over time as improvements are made and bugs are fixed, and as additional features (e.g., creation of a status graph) are provided. Given the fact that the application model is completely technology agnostic, and that it can be used for any version of the expanders, these bug fixes and additional features become available for all versions of all application models: only a *re-expansion* or *"rejuvenation"* is required. As a consequence, the expanders or template skeletons represent a second dimension of variability or evolvability.

Third, as represented in the upper right of the figure, one should specify *infrastructural options* to select a number of frameworks or *utilities* to take care of several generic or so-called cross-cutting concerns. These options consist of global options (e.g., determining the build automation framework), presentation settings (determining the graphical user interface frameworks), business logic settings (determining the database and transaction framework to be used) and technical infrastructure (e.g., selecting versions for access control or persistency frameworks). This means that, given a chosen application model version and expander version, different variants of boiler plate code can be generated, depending on the choices regarding the infrastructural options. As a consequence, the settings and utility frameworks represent a third dimension of variability or evolvability.

Fourth, as represented in the lower left of the figure, "custom code" or *craftings* can be added to the generated source code. These craftings enrich, i.e., are put upon, the earlier generated boiler plate code. They can be *harvested* into a separate repository before regenerating the software application, after which they can *re-injected*. The craftings include extensions, i.e., additional classes added to the generated code base, as well as insertions, i.e., additional lines of code added between the foreseen anchors within the code. Craftings can have multiple versions throughout time (e.g., being updated or complemented), or concurrently (e.g., choosing between a more advanced or simplified version). These craftings should contain as little technology specific statements within their source code as possible (apart from the chosen background technology, i.e., the programming language). Indeed, craftings referring to (for instance) a specific UI framework will only be reusable as long as this particular UI framework is selected for the generation of the application. In contrast, craftings performing certain validations, but not containing any specific references to the transaction framework, e.g., *Enterprise Java Beans (EJB)*, can simply be reused when applying other versions or choices regarding such a framework. As a consequence, the custom code or craftings represent a fourth dimension of variability or evolvability.

In summary, each part in Figure 3 is a variability dimension in an NST software development context. It is clear that talking about *the* "version" of an NST application, as is traditionally done for software systems, becomes more refined in such a context. Indeed, the eventual software application codebase (the lower right side of the figure) is the result of a specific version of an application model, expander version, infrastructural options, and a set of craftings [19]. Put differently, with M, E, I and C referring to the number of available application model versions, the number of expander versions, the number of infrastructural option combinations, and the number of crafting sets respectively, the total set of possible versions V of a particular NST application becomes equal to:

$$V = M \times E \times I \times C \tag{2}$$

Whereas the specific values of M and C are different for every single application, the values of E and I are dependent on the current state of the expanders. Remark that the number of infrastructural option combinations (I) is equally a product:

$$I = G \times P \times B \times T \tag{3}$$

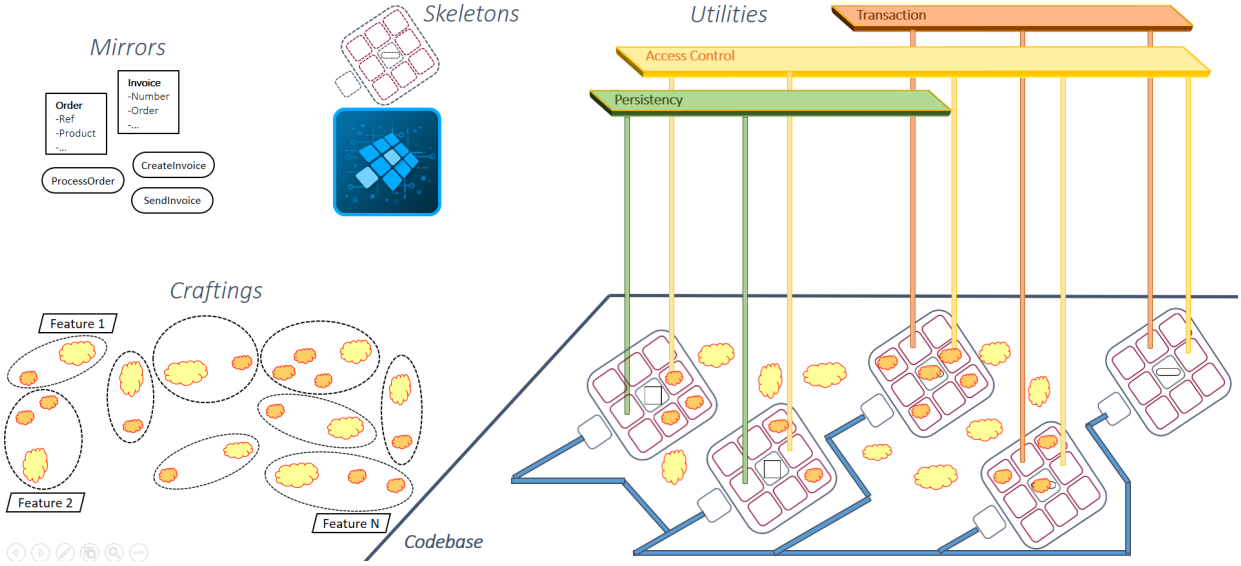

Fig. 3. A graphical representation of four variability dimensions within a Normalized Systems application codebase.

In this case, G represents the number of available global option settings, P the number of available presentation settings, B the number of available business logic settings, and T the number of available technical infrastructure settings. This general idea in terms of combinatorics corresponds to the overall goal of NST: enabling evolvability and variability by *leveraging the law of exponential variation gains* by means of the thorough decoupling of the various concerns, and the facilitation of their recombination potential [12].

IV. Meta-Circular Expansion Software

In this section, we present the meta-circular architecture of the NST expansion software, i.e., the automatic programming code that is also able to regenerate itself as well. Both the sequential phases toward achieving this meta-circular code (re)generation architecture, and the declarative control structure of the expansion software, are described.

A. Toward Meta-Circular Expansion

1) Phase 1: Standard Code Generation: The original architecture of the Normalized Systems expansion or code generation software is schematically represented in Figure 4. On the right side of the figure, the generated source code is represented in blue, corresponding to a traditional multi-tier web application. Based on a *Java Enterprise Edition (JEE)* stack [17] [19], the generated source code classes are divided over so-called layers, such as the logic, the control, and the view layer. On the left side, we distinguish the internal structure of the expanders or the code generators, represented in red. This corresponds to a very straightforward implementation of code generators, consisting 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 expander classes based on the parameters.
- *expander classes* instantiating the source templates, using the *String Template (ST)* library, and feeding the model parameters to the source templates.
- *source templates* containing the parametrized code.

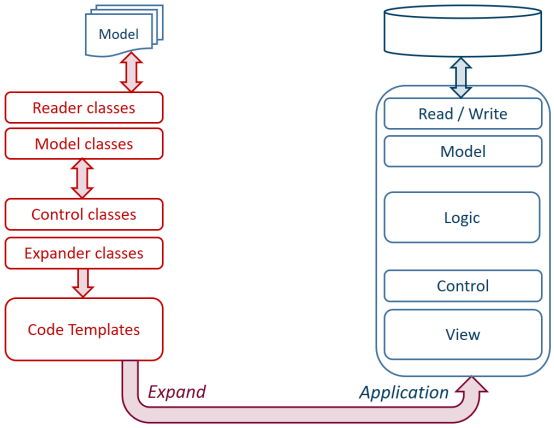

Fig. 4. Representation of a basic code generator structure.

2) Phase 2: Generating a Meta-Application: Essentially, code generation models or meta-models — and even all collections of configuration parameters — consist of various data entities with attributes and relationships. As the Normalized Systems element definitions are quite straightforward [17] [19], the same is valid for its meta-models. Moreover, one of the Normalized Systems elements, i.e., the data element, is basically a data entity with attributes. This means that the NS meta-models, being data entities with attributes, can be expressed as regular models. For instance, in the same way 'Invoice' and 'Order' can be specified as NS data elements with attributes and relationships in an information system

model, the NS 'data element' and 'task element' of the NS meta-model can be defined as NS data elements with attributes and relationships, just like any other NS model.

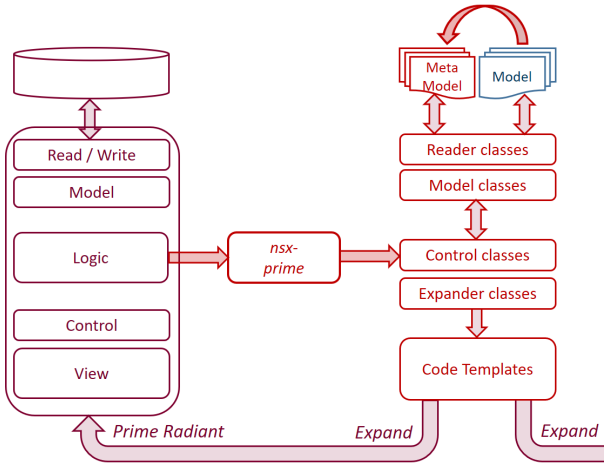

Fig. 5. Expansion of a meta-application to define meta-models.

As the NS models can be considered a higher-level language according to Parnas [3], the single structure of its model data and meta-model language means that the NS model language is in fact homoiconic in the sense of [14]. This also enables us to expand or generate a meta-application, represented on the left side of Figure 5 in dark red. This NS meta-application, called the *Prime Radiant*, is a multi-tier web application, providing the functionality to enter, view, modify, and retrieve the various NS models. As the underlying meta-model is just another NS model, the Prime Radiant also provides the possibility to view and manipulate its own internal model. Therefore, by analogy with the meta-circular evaluator of Reynolds [15], the Prime Radiant can be considered to be a *meta-circular application*.

For obvious reasons, the generated reader and model classes (part of the Prime Radiant on the left side of Figure 5) slightly differ from the reader and model classes that were originally created during the conception of the expansion or code generation software (on the right side of Figure 5). This means that in order to trigger and control the actual expansion classes to generate the source code, an integration software module needed to be developed, represented in the middle of Figure 5 as *nsx-prime*. Though the Prime Radiant meta-application is auto-generated, and can therefore be regenerated or rejuvenated as any NS application, this *nsx-prime* integration module needed to be maintained manually.

3) Phase 3: Closing the Expander Meta-Circle: Though the original reader and model classes of the expander software differed from the generated reader and writer classes, there is no reason that they should remain separate. It was therefore decided to perform a rewrite of the control and expander classes of the expander software (on the right side of Figure 5), to allow for an easier integration with the auto-generated reader and model classes (on the left side of Figure 5). Enabling such a near-seamless integration would not only eliminate the need for the reader and model classes of the expander software, it would also reduce the complexity of the *nsx-prime* integration component to a significant extent.

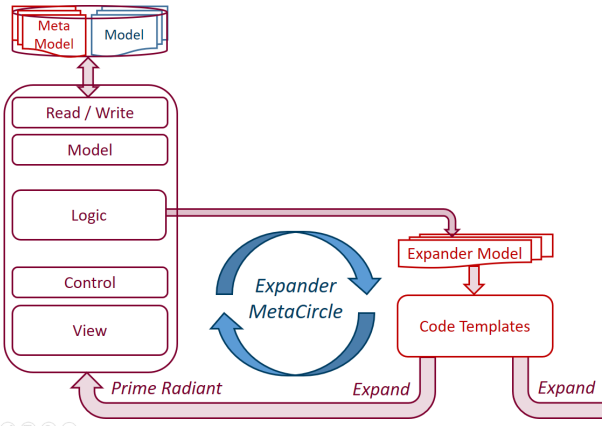

Fig. 6. Closing the meta-circle for expanders and meta-application.

Originally, the refactoring was only aimed at the elimination of the reader and control classes of the expander software. During the refactoring however, it became clear that the control and expander classes of the expander software implementation could be eliminated as well. Indeed, by adopting a declarative structure to define the expander templates and to specify the relevant model parameters, both the control classes (selecting and invoking the expander classes) and the expander classes (instantiating and feeding the parameters to the source templates) were no longer necessary. Moreover, as schematically represented in Figure 6, the refactoring also eliminated the need for the *nsx-prime* integration module. As extensions to the meta-model no longer require additional coding in the various expander software classes (e.g., reader, model, control, and expander classes), nor to the *nsx-prime* integration module, one can say that the *expander development meta-circle* has been closed. This is symbolically visualized in Figure 6. Indeed, expander templates can be introduced by simply defining them, and extensions to the NS meta-model become automatically available after re-running the expansion or code generation on this meta-model.

B. Declarative Expansion Control

The expansion process of an NS element is schematically represented in Figure 7. The internal structure of an NS element (data, task, flow, connector, and trigger element) is based on a detailed design pattern [16] [17] [12], implemented through a set of source code templates. We call this set of coding templates the *Element Template*, represented in dashed lines on the left side of Figure 7. During the actual expansion or code generation, for every instance of an NS element, e.g., a data element 'Invoice', the set of source code templates is instantiated, steered by the parameters of the model. This results in the actual element, e.g., *Invoice Element*, which is represented in solid lines and corresponds to Figure 2.

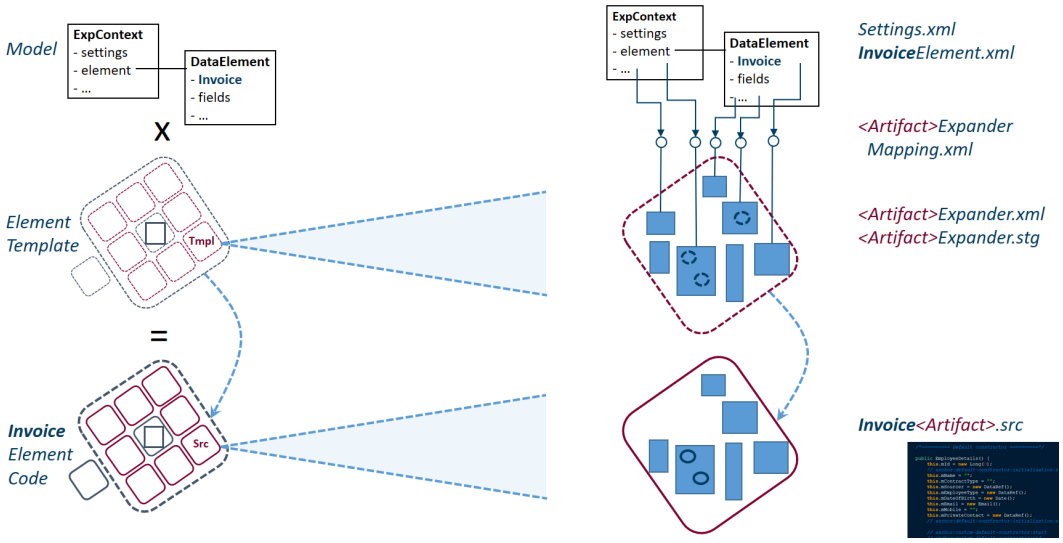

Fig. 7. Expansion of an *Invoice* element, zooming in on a single artifact.

1) Declarative Representation of Expanders: On the right side of Figure 7, we zoom in on the expansion of an individual source code template into a source code artifact, e.g., a class, applying and inserting the parameters of the model. We refer to this unit of code generation as an individual *expander*, and mention that the NS code generation environment for web-based information systems currently consists of 182 individual expanders. Every individual expander is declared in an XML document. An example of such an expander declaration, defining various properties, is shown below.

```
<expander name="DataExpander"
  xmlns="http://normalizedsystems.org/expander">
  <packageName>expander.jpa.dataElement</packageName>
  <layerType name="DATA_LAYER"/>
  <technology name="JPA"/>
  <sourceType name="SRC"/>
  <elementTypeName>DataElement</elementTypeName>
  <artifact>$dataElement.name$Data.java</artifact>
  <artifactPath>$componentRoot$/$artifactSubFolders$/
    $dataElement.packageName</artifactPath>
  <isApplicable>true</isApplicable>
  <active value="true"/>
  <anchors/>
</expander>
```

In this declaration, we find the following information.

- The identification of the expander, name and package name, which also identifies in an unambiguous way the source code template.
- Some technical information, including the tier or layer of the target artifact in the application, the technology it depends on, and the source type.
- The name and the complete path in the source tree of the artifact that will be generated, and the type of NS element that it belongs to.
- Some control information, stating the model-based condition to decide whether the expander gets invoked.
- Some information on the anchors delineating sections of custom code that can be harvested and re-injected.

2) Declarative Mapping of Parameters: The instantiation of an individual source code template for an individual instance of an NS element, is schematically represented on the right side of Figure 7. It can be considered as a transformation that combines the parameters of the model with the source code template, and results in a source code artifact. We therefore distinguish three types of documents or artifacts.

- The model parameters, represented at the top of Figure 7, consist of the attributes of the element specification, e.g., the data element 'Invoice' with its fields or attributes, and the options and technology settings. All these parameters are available through the auto-generated model classes, e.g., *InvoiceDetails*, and may either originate from the Prime Radiant database, or from XML files.
- An individual source code template, having a unique name that corresponds to the one of the expander definition as presented above. Such a template, represented in the middle of Figure 7, contains various insertions of parameter values, and parameter-based conditions on the value and/or presence of specific parts of the source code.
- An instantiated source file or artifact, represented at the bottom of Figure 7, where the various values and conditions in the source code template have been resolved.

An important design feature is related to the mapping of the parameters from the model to the parameters that appear in the source code templates, that are directly guiding the code instantiation. In order to provide loose coupling between these two levels of parameters, and to ensure a simple and straightforward relationship, it was decided to implement this mapping in a declarative *ExpanderMapping* XML file. As the entire NS model is made available as a graph of model classes, the parameters in the templates can be evaluated from the NS model using *Object-Graph Navigation Language (OGNL)* expressions. These expressions, e.g., *Invoice.number*, are declared in the XML mapping file of the expander.

V. Two-Sided Scalable Collaboration

In this section, we explain how the meta-circular architecture of the metaprogramming software enables an open and scalable collaboration based on its two-sided interfaces.

A. The Need for Meta-Level Interfaces

The main purpose of an *Application Programming Interface (API)* is to enable widespread and scalable collaboration in software development. It allows for a structured collaboration between developers implementing the interface on one side, and developers using or invoking the programming interface on the other side. The collaboration is possible both within companies and across companies, and in open source communities. The use of such an API has contributed significantly to the rich application offering that we have, including desktop applications and mobile apps, and to the abundant hardware support that we enjoy, providing drivers for a multitude of peripheral devices on a variety of operating systems.

In order to enable scalable collaboration in metaprogramming, we should define meta-level programming interfaces. However, defining meta-level interfaces is still a subject of research today. We have mentioned for instance research papers of Wortmann [7], presenting a novel approach for the systematic reuse of *Domain Specific Languages (DSLs)*, and Gusarov et al. [8], proposing an intermediate representation to be used for code generation. We believe that the complex architectures of metaprogramming environments, exhibiting high degrees of coupling as represented on the left side of Figure 8 (which is similar to the representation in Figure 4), make it nearly impossible to define clear meta-level interfaces.

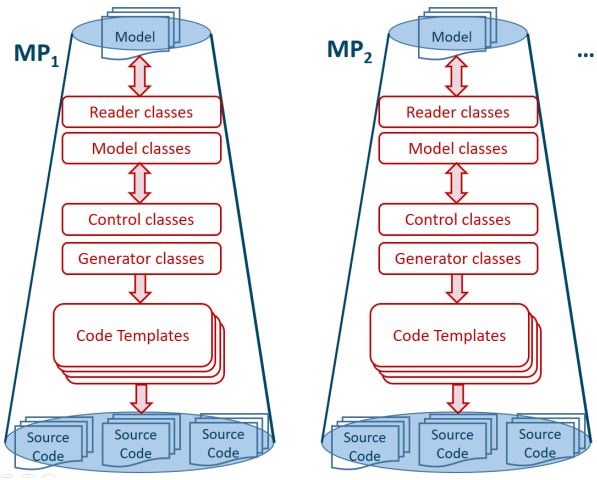

Fig. 8. Representation of various metaprogramming silos.

What 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. This implies that (programming) interfaces need to be defined at both ends of the transformation, allowing at the same time to define or extend domain models, and to implement or replace code generators or templates. We argue that the presented meta-circular architecture enables the definition of the interfaces at both ends of the transformation, as it allows to integrate —or at least accommodate— ever more extensions and alternatives at the two sides of the interface, without entailing the non-scalable burden of adapting the metaprogramming code.

B. Separating Model and Code Interfaces

Similar to the representation in Figure 4, the left hand side of Figure 8 represents a basic code generator or metaprogramming architecture. However, if we consider another metaprogramming environment (for instance on the right side of Figure 8), we will almost certainly encounter a duplication of such an architecture. This will in general result in what we could describe as *metaprogramming silos*, entailing 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.

The meta-circular environment presented in Section IV-A and Section IV-B addresses these drawbacks. The architecture establishes a clear decoupling between the models and the code generation templates, and removes the need for contributors to get acquainted with the internal structure of the metaprogramming environment. It could allow developers to collaborate in a scalable way at both sides of the metaprogramming interfaces. And, as schematically represented in Figure 9, the clear decoupling of this *horizontal integration* architecture could bring the same kind of variability gains as described in Equations (2) and (3). Indeed, in such a decoupled environment, it should be possible to combine every version or variant of the model with every version or variant of the coding templates to generate a codebase. This implies:

$$N \text{ models } + M \text{ templates } \Longrightarrow N \times M \text{ codebases} \quad (4)$$

The use of a metaprogramming environment with a clear decoupling between models and code (templates) could also entail significant productivity gains for regular software development projects. Consider for instance Figure 10, representing in a schematic way several collaborative projects, e.g., open source projects, for *Enterprise Resource Planning (ERP)* software. These application software projects consist of models, configuration data, and source code, and are in general organized as silos as well. Thus making it very difficult to collaborate between different projects. Using a metaprogramming environment that decouples models and coding templates in the way we have presented, could open up new possibilities.

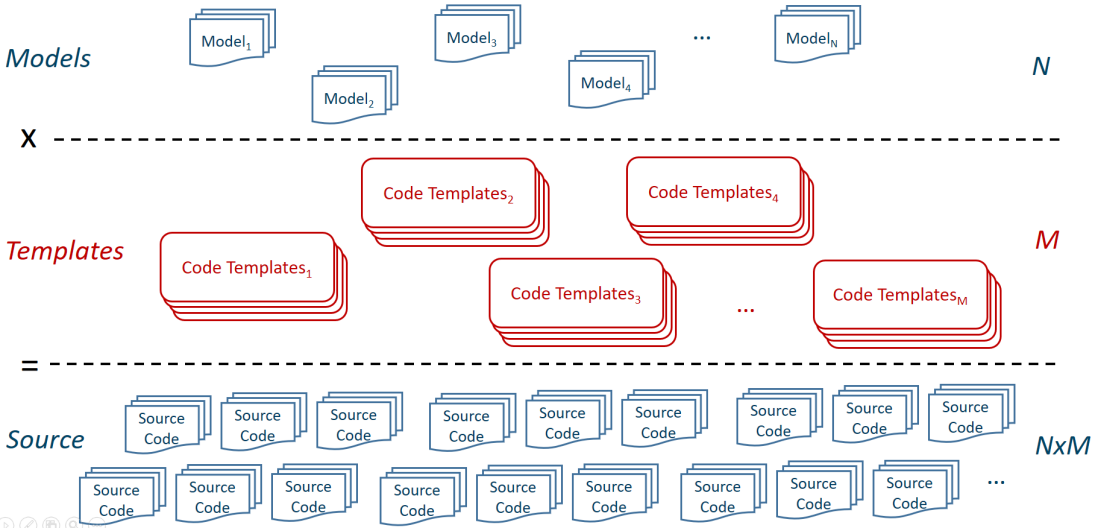

Fig. 9. A graphical representation of four variability dimensions within a Normalized Systems application codebase.

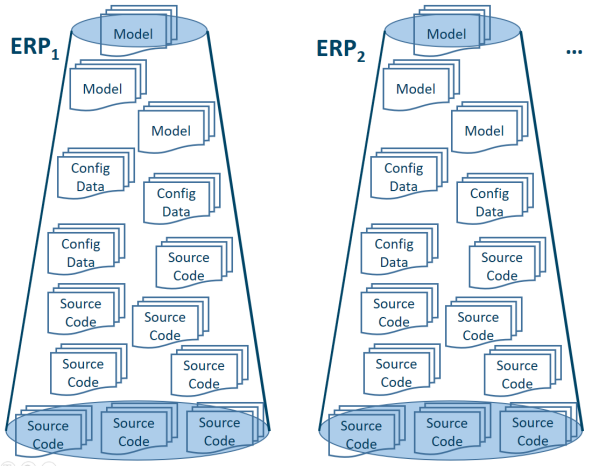

Fig. 10. Representation of various ERP software silos.

Modelers and designers would be able to collaborate on domain models, gradually improving existing model versions and variants, and adding on a regular basis new functional business modules, without having to bother about technicalities. (Meta)programmers would 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. Application software systems would be generated to a large extent for a selected set of versions of domain models, using a specified set of coding templates, being targeted at a specific set of technology platforms. This would realize the above mentioned variation gains as described in Equations (2), (3), and (4).

C. Enabling Alternative Meta-Models

The current meta-model of our metaprogramming environment, consists of data, action, workflow, trigger, and connector elements, and is to a large extent specific for (web-based) information systems. It is not only conceivable to modify and/or extend this meta-model, but one could also imagine to define completely different meta-models for other purposes. Such meta-models could for instance be created to model traditional computing functions based on sequencing-selection-iteration, or to represent microservices or scripting units running parts of data mining algorithms in the cloud.

The meta-circular architecture presented in Figure 6 supports not only the definition of other models, but enables the definition of other meta-models as well. Indeed, the reader, model, control, and view classes could be generated for such an alternative meta-model, allowing the specification of models based on this new meta-model, both in XML or in a newly generated meta-application. These model parameters could then be propagated to new sets of coding templates. This could allow the presented meta-circular architecture to generate all types of source code or configuration data for all kinds of applications and languages.

VI. Some Results and Discussion

In this section, we present and discuss some empirical results regarding the use of the meta-circular expansion software. It is based on qualitative research that has been performed within the company, i.e., NSX bv, that develops the expansion software, and relies on nonnumerical data obtained from first-hand observation and interviews.

A. Refactoring of Existing Expanders

The Normalized Systems expander software has been in development since late 2011. Over the years, it was used by several organizations to generate, and re-generate on a regular basis, tens of information systems [18] [19]. During these years, and often on request of these organizations, many additional features and options were built into the source code templates. The overall refactoring was triggered by a concern

over the growing size — and therefore complexity — of the model and control classes. It was also motivated by a desire to leverage the implicit homoiconicity of the NS (meta-)model to increase the productivity of improving and extending the expander software.

The complete refactoring was started in the last quarter of 2018, and performed in six months by two developers. Afterwards, the 182 expanders, each expanding or generating a specific source code artifact, were cleanly separated, and the software developers considered the expander codebase to be much better maintainable. Moreover, the learning curve for developers to take part in expander development was previously considered to be very steep, mainly due to the size and complexity of the model and control classes. After the refactoring, nearly all of the approximately 20 application developers of the company, have stated that the learning curve is considerably less steep, and that they feel comfortable to add features and options to the expander software themselves. One year later, most of them have indeed made such a contribution. The lead time of performing fixes and modifications to the expander coding templates, and to deliver it to the application project teams, has decreased from weeks to days.

B. Creating Additional Expander Bundles

Besides contributing to the original set of 182 expanders, application developers —even junior developers— are able to create additional *expander bundles* implementing a set of expanders for a specific functionality. Immediately after the refactoring, a junior developer has created in two months such a bundle of 20 expanders, targeted mainly at the implementation of REST services using Swagger. One year later, this bundle has been used successfully in numerous projects, and has grown significantly. Other —often junior— developers have created in the meantime various expander bundles providing valuable functionality. These include a bundle to generate mobile apps connecting to the expanded web-based information systems, a bundle to define more advanced search queries and reporting, and a bundle supporting various types of authentication. Recently, a newly hired graduate developed a small bundle of expanders to implement row-level security, only a couple of weeks after joining the company.

Currently, two different customers are developing expander bundles as well. As our goal is to establish a scalable collaboration in metaprogramming across a wide range of organizations and developers, we are setting up an exchange marketplace for expander bundles at *exchange.stars-end.net*.

C. Supporting Alternative Meta-Models

We have explained that the meta-circular metaprogramming architecture also provides the possibility to create and adopt other meta-models. Currently, a first implementation is available allowing to define other meta-models and providing coding templates, while all the code in between (readers, model, and control classes) of *this new* metaprogramming environment is automatically generated. A collaboration has been established with another group working on a metaprogramming environment, to perform a horizontal integration between the two metaprogramming environments. The first promising results of this integration have been reported [20].

One could argue that there is an implicit *meta-meta-model* underlying the various possible meta-models, and that this meta-meta-model could create a *silo-effect* hampering the integration of various metaprogramming efforts. However, the implicit meta-meta-model is based on the data elements of the NS meta-model, containing only data fields or attributes, and link fields or relationships. This is very similar, if not identical, to both the data entities of *Entity Relationship Diagrams (ERD)* with their data attributes and relationship links, and the entities or classes of the *Web Ontology Language (OWL)* with their datatype properties and object properties. Having for instance demonstrated the bi-directional transformation between our models and domain ontologies [21], we are confident that the dependency on the underlying meta-meta-model will not impede scalable collaborations.

VII. Conclusion

The increase of productivity and the improvement of evolvability are goals that have been pursued for a long time in computer programming. While more research has traditionally been performed on techniques to enhance productivity, our research on Normalized Systems Theory has been focusing on the evolvability of information systems. This paper presents a strategy to combine both lines of research.

While the technique of automated programming or source code generation was already part of our previous work on Normalized Systems, we have explored in this paper the incorporation of homoiconicity and meta-circularity to increase the productivity of our metaprogramming environment. A method was presented to turn the metaprogramming environment into a meta-circular architecture, using an homoiconic representation of the code generation models, and resulting in a considerable simplification of the expanders, i.e., the code generation software. We have argued that such a reduction of complexity could lead to a significant increase in productivity at the level of the development of the code generation software, and that the two-sided interfaces could enable a scalable collaboration on metaprogramming across organizations and developers. We have presented some preliminary results, indicating that the increase in metaprogramming productivity is indeed being realized, and have established a first collaborative integration effort with another metaprogramming environment.

This paper is believed to make some contributions. First, we show that it is possible to not only adopt code generation techniques to improve productivity, but to incorporate meta-circularity as well to improve both productivity and maintainability at the metaprogramming level. Moreover, this is demonstrated in a framework primarily targeted at evolvability. Second, we have presented a case-based strategy to make a code generation representation homoiconic, and the corresponding application architecture meta-circular. Finally, we have argued that the simplified structure of the code generation

framework improves the possibilities for collaboration at the level of metaprogramming software.

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 making a code generation or metaprogramming environment meta-circular. Moreover, the presented results are both qualitative and preliminary, and the achieved collaboration on metaprogramming software is limited to a small amount of organizations. However, we are currently working to set up a collaboration of developers on a much wider scale at the level of metaprogramming, and to prove that this architecture can lead to new and much higher levels of productivity and collaboration in the field of automatic programming.

REFERENCES

[1] H. Mannaert, K. De Cock, and P. Uhnák, "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)*, November 2019, pp. 171–176.

[2] R. Agarwal and A. Tiwana, "Editorial—evolvable systems: Through the looking glass of IS," *Information Systems Research*, vol. 26, no. 3, pp. 473–479, 2015.

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

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

[5] 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.

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

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

[8] 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, pp. 74–82.

[9] U. Frank, "Specification and management of methods - a case for multi-level modelling," *Business Process and Information Systems Modeling*, vol. 352, pp. 311–325, 2019.

[10] J. Hutchinson, M. Rouncefield, and J. Whittle, "Model-driven engineering practices in industry," in *Proceedings of the International Conference on Software Engineering 2011*, 2011, pp. 633–640.

[11] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE Software*, vol. 31, no. 3, pp. 79–85, 2014.

[12] 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.

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

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

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

[16] 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, pp. 1210–1222, 2011, special Issue on Software Evolution, Adaptability and Variability.

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

[18] P. De Bruyn, H. Mannaert, and P. Huysmans, "On the variability dimensions of normalized systems applications: Experiences from an educational case study," in *Proceedings of the Tenth International Conference on Pervasive Patterns and Applications (PATTERNS)*, 2018, pp. 45–50.

[19] ——, "On the variability dimensions of normalized systems applications: experiences from four case studies," *International Journal on Advances in Systems and Measurements*, vol. 11, no. 3, pp. 306–314, 2018.

[20] 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)*, October 2020, pp. 166–172.

[21] M. Suchánek, H. Mannaert, P. Uhnák, and R. Pergl, "Bi-directional transformation between normalized systems elements and domain ontologies in owl," in *Proceedings of the fifteenth International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, May 2020, pp. 1–12.