On the Realization of Meta-Circular Code Generation: The Case of the Normalized Systems Expanders

Herwig Mannaert

Normalized Systems Institute
University of Antwerp, Belgium
Email: herwig.mannaert@uantwerp.be

Koen De Cock and Peter Uhnak

Research and Development
NSX BVBA, Belgium
Email: koen.de.cock@nsx.normalizedsystems.org

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, a feature that may be crucial to improve the productivity of the development of software for source code generation. The detailed architecture of this meta-circular code generation software is presented, and some preliminary results are discussed.**

***Keywords*–Evolvability; Normalized Systems; Meta-circularity; Automated programming; Case Study**

I. Introduction

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 [1]. 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 evolvability, and automated code generation and homoiconic programming to offer increased productivity.

The remainder of this paper is structured as follows. In Section III, we briefly discuss two fundamental approaches to increase the productivity in computer programming: automatic and homoiconic programming. In Section III, we briefly present NST as a theoretical basis to obtain higher levels of evolvability in information systems. Section IV discusses the application of these fundamental concepts to the Normalized Systems code expansion in general and the Prime Radiant in particular. Section V elaborates on the declaration of both the various expanders generating the code artifacts, and the configuration parameters that control the expansion process. Finally, we discuss some results in Section VI and present our conclusion in Section VII.

II. Automatic and Homoiconic Programming

In this section, we briefly discuss two fundamental and long-standing approaches to increase the programming productivity: automatic and homoiconic programming.

A. Automatic or Meta-Programming

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” [2].

Another term used to designate automatic programming is *generative programming*, aiming to write programs ”to manufacture software components in an automated way” [3], 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 *meta-programming*. 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 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, though the realization of this goal is not always straightforward [4].

B. Homoiconicity or Meta-Circularity

Another technique in computer science aimed at the increase of the abstraction level, and thereby the productivity, of computer programming, 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 conceptually enables meta-programming to become much easier. 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 [5], traces the fundamental concepts back to an even earlier paper from McIlroy [6].

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 [7]. 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 meta-level, the internal model of the language, and the actual activity, writing models in the language.

III. Normalized Systems Theory and Expansion

In this section, we introduce NST as a theoretical basis to obtain higher levels of evolvability in information systems, and its realization in a code generation or *expansion* framework.

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 [1]. 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 [8][9][10]. 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, this implies that a bounded set of changes should only result in a bounded impact 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 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) [8][9][10]:

- *Separation of Concerns*
- *Action Version Transparency*
- *Data Version Transparency*
- *Separation of States*

The application of the theorems in practice has shown to result in very fine-grained modular structures within a software application. Such structures are, in general, difficult to achieve by 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 proposes five *elements* that serve as design patterns [9][10]:

- *data element*
- *action element*
- *workflow element*
- *connector element*
- *trigger element*

Based on these elements, NST software is generated in a relatively straightforward 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 parameterized 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 well specified places (anchors) within the boiler plate code.

B. Variability Dimensions and Expansion

In applications generated by a Normalized Systems expansion process, we identify four variability dimensions, as visualized in Figure 1. As discussed in [11][12], the combination of these dimensions compose an actual Normalized Systems application codebase, and therefore determine how such an application can evolve throughout 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 DFD'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, the *expanders* (represented by the big blue icon in the figure) generate (boiler plate) source code by instantiating the various class templates or *skeletons*, taking the specifications of the model as *parameters*. For instance, for a data element Person, a set of java classes PersonBean, PersonAgent, PersonView, PersonData, 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 standard finders, master-detail (waterfall) screens, certain display options, document upload/download functionality, child relations, etcetera. The expanders or template skeletons themselves evolve throughout time, as bugs of the previous version are solved and additional features (e.g., creation of a status graph) are provided. Given the fact that the application model is completely technology agnostic and can be used as argument

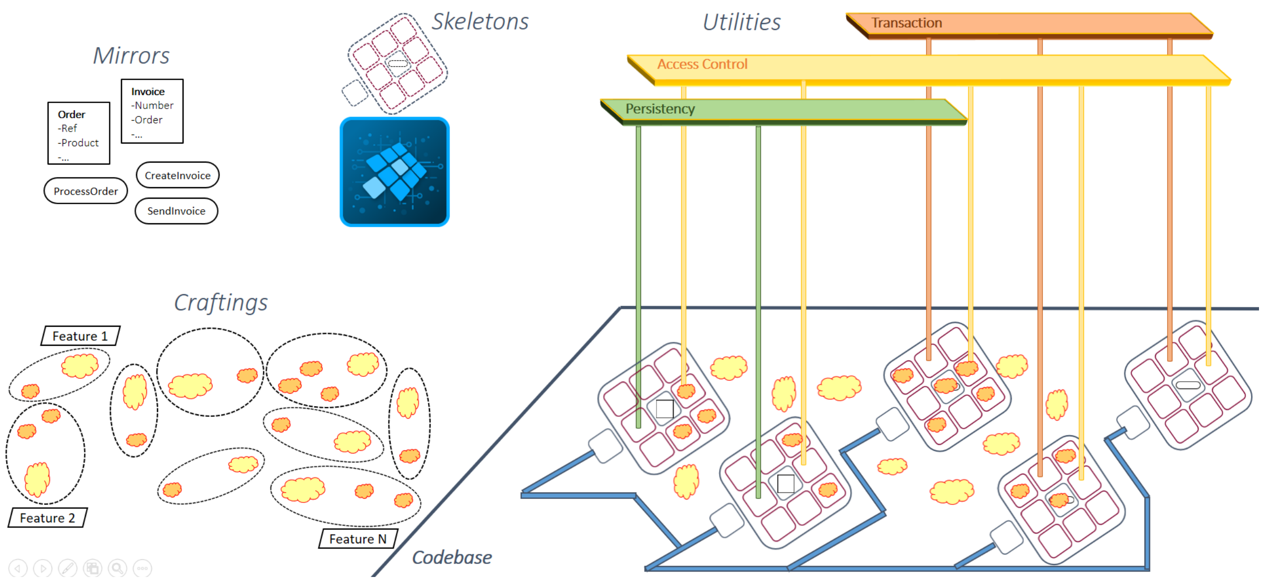


Figure 1. A graphical representation of four variability dimensions within a Normalized Systems application codebase.

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 concerns. These consist of global options (e.g., determining the build automation framework), presentation settings (determining the graphical user interface framework), business logic settings (determining the database used) and technical infrastructure (e.g., access control or persistency). 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 (are put upon) the earlier generated boiler plate code and can be harvested into a separate repository before regenerating the software application (after which they can be applied again). This includes extensions (e.g., 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). Indeed, craftings referring to (for instance) a specific GUI framework will only be reusable as long as this particular GUI framework is selected during the generation of the application. In contrast, craftings performing certain validations but not containing any EJB specific statements will be able to be reused when applying other versions or choices regarding such framework. As a consequence, the custom code or craftings represent a fourth dimension of variability or evolvability.

In summary, each part in Figure 1 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) in such context becomes more refined. 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 [12]. 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 crafting sets respectively, the total set of possible versions V of a particular NST application becomes equal to:

$$V = M \times E \times I \times C$$

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:

$$I = G \times P \times B \times T$$

Where 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 concerns and the facilitation of their recombination potential [10].

IV. Toward Meta-Circular Expansion Control

In this section, we discuss the application of automatic programming and homoiconicity to the Normalized Systems expansion in general and the Prime Radiant in particular.

A. Phase 1: Standard Code Generation

The original architecture of the Normalized Systems expansion or code generation software is schematically represented in Figure 2. In the right part 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 [9][12], the generated source code classes are divided over so-called layers, such as the logic, the control, and the view layer. On the left, 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:

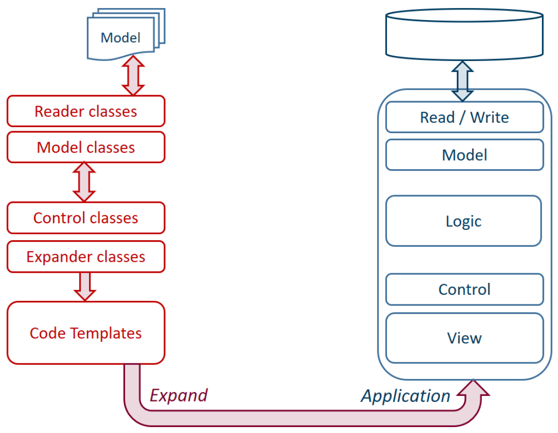


Figure 2. Representation of a basic code generator structure.

- *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.
- *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.

B. Phase 2: Generating a Meta-Application

In essence, 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 [9][12], the same is valid for its metamodels. 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 'Person' and 'Invoice' can be 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 like any other NS model.

As the NS models can be considered a higher-level language according to Parnas [2], 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 [6]. This also enables us to expand or generate a meta-application, represented on the left of the figure in dark red, as represented in Figure 3. This NS meta-application, called the *Prime Radiant*,

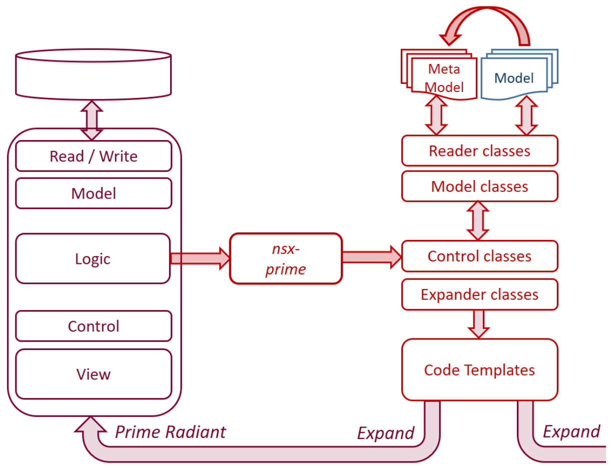


Figure 3. Expansion of a meta-application to define meta-models.

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 model. Therefore, by analogy to the meta-circular evaluator of Reynolds [7], the Prime Radiant can be considered as a *meta-circular application*.

For obvious reasons, the generated reader and model classes (part of the Prime Radiant on the left of Figure 3) 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 of Figure 3. 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 3 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.

C. 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 so. It was therefore decided to perform a rewrite of the control and expander classes of the expander software (right side of Figure 3), in order to allow for an easier integration with the auto-generated reader and model classes (left side of Figure 3. 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.

Originally, the refactoring was only aimed at the elimination of the reader and control classes of the original expander software. However, during the refactoring, it became clear that it became possible to retire the control and expander classes of the expander software as well. Indeed, by adopting a declarative structure to define the expander templates and to specify the relevant model parameters, both the control classes

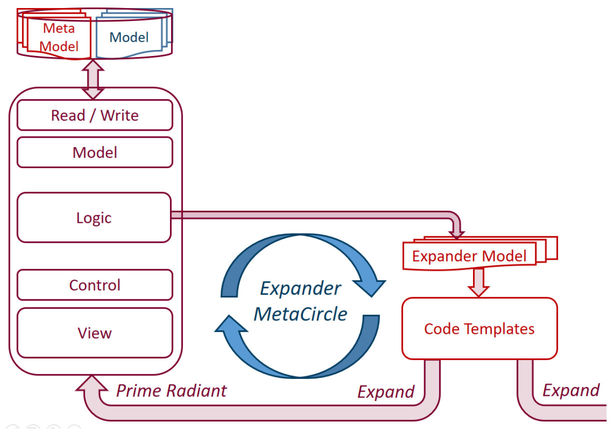


Figure 4. Closing the meta-circle for expanders and meta-application.

(selecting and invoking the expander classes) and the expander classes (instantiating and feeding the source templates) were no longer necessary. Moreover, as schematically represented in Figure 4, the refactoring also eliminated the *nsx-prime* integration module. As extensions to the meta-model no longer require additional coding various expander software classes (e.g., reader, model, control, and expander classes), nor in the nsx-prime integration module, one can say that the *expander development meta-circle* has been closed, as visualized in Figure 4. Indeed, expander templates can be introduced by simply defining them, and extensions to the NS meta-model simply become available after re-running the expansion or code generation on this meta-model.

V. Declarative Expansion Control Structure

In this section, we elaborate on the declaration of both the various expanders generating the code artifacts, and the configuration parameters that control the expansion process.

A. Declarative Representation of Expanders

The basic NS expander software currently consists of 181 individual expanders. Every expander is able to instantiate a specific template into a corresponding artefact, using the parameters of the model. An example of the definition of such an individual expander is shown below.

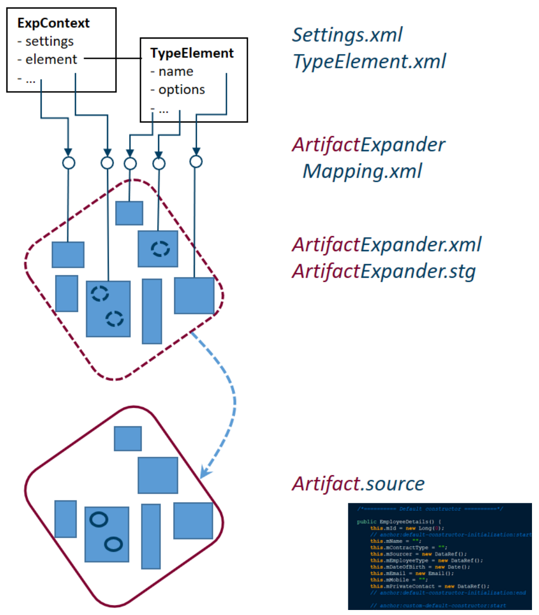
```
<expander name="DataExpander"
  xmlns="http://normalizedsystems.org/expander">
  <packageName>expander.jpa.dataElement</packageName>
  <layerType name="DATA_LAYER"/>
  <technology name="JPA"/>
  <sourceType name="SRC"/>
  <elementTypeName>DataElement</elementTypeName>
  <artifact>$dataElement.name$Data.java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

Such an expander definition in XML format contains 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.

B. Declarative Mapping of Parameters

The internal structure of an NS element (data, task, flow, connector, and trigger element) is based on a detailed design pattern [8][9][10], implemented through a set of source code templates, each represented by an expander definition. During the actual expansion or code generation, for every instance of an NS element, e.g., a data element 'Person', the set of source code templates is instantiated, steered by the parameters of the model. The instantiation of an individual source code template for an individual instance of an NS element, is schematically represented in Figure 5, and contains the following aspects.



```
/*========== Default constructor ==========*/

public EmployeeDetails() {
    this.mId = new Long(0);
    // anchor:default-constructor-initialization:start
    this.mName = "";
    this.mContractType = "";
    this.mSourcer = new DataRef();
    this.mEmployeeType = new DataRef();
    this.mDateOfBirth = new Date();
    this.mEmail = new Email();
    this.mMobile = "";
    this.mPrivateContact = new DataRef();
    // anchor:default-constructor-initialization:end

    // anchor:custom-default-constructor:start
```

Figure 5. Expansion of a single artifact.

- The model parameters, represented on the top of Figure 5, consisting of the attributes of the element specification, e.g., the data element 'Person' with its attributes, and the options and technology settings. All these parameters are available through the auto-generated model classes, and may either originate from the Prime Radiant database, or from XML files.
- An individual source code template, having unique name that corresponds to the one of the expander definition as described above. Such a template, represented in the middle of Figure 5, contains various 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 5, where the various 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 and are thereby guiding the 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 in the NS model using *Object-Graph Navigation Language (OGNL)* expressions. These expressions are declared in the XML mapping file of the expander.

VI. Some Preliminary Results

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 [11][12]. 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 control classes, but also motivated by a desire to leverage the implicit homoiconicity of the NS model to increase the productivity of improving and extending the expander software.

The complete refactoring was performed in six months by two developers. Afterwards, the 181 expanders 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 widely considered to be very steep, mainly due to the size and complexity of the 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. As an additional result, we mention the fact that a junior developer has created in two months a new set of 20 expanders. This newly developed collection of expanders, targeted mainly at the development of REST services using Swagger, has already been successfully used in several projects.

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 has been focusing on the evolvability of information systems. This paper presented a strategy to combine both lines of research.

While the technique of automated programming or source code generation was already incorporated in our work on Normalized Systems, we have explored in this paper the technique of homoiconicity or meta-circularity to increase the productivity of the automatic or meta-programming. A method was presented to make the representation of the code generation models homoiconic, resulting in a considerable simplification of the expanders, i.e., the code generation software. 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 we have presented some very preliminary results that this is indeed the case.

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 the productivity of the code generation. 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 meta-circular. Finally, we believe that the simplified structure of the code generation framework improves the possibilities for collaboration at the level of code generation 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 application meta-circular. Moreover, the presented results are very preliminary, and the achieved collaboration on code generation software is still limited to nearby colleagues. However, it is our goal to set up a collaboration of developers on a much wider scale at the level of code generation software, and to prove that this architecture can lead to new and much higher levels of productivity for developing automatic programming.

References

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

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

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

[4] 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.

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

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

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

[8] 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.

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

[10] 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.

[11] 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, 2018, pp. 45–50.

[12] ——, "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, 1960, pp. 306–314.