International Journal on Advances in Software, vol 16 no 1 & 2, year 2023, http://www.iariajournals.org/software/

Toward Leveraging Code Generation Architectures for the Creation of Evolvable Documents and Runtime Artifacts

Herwig Mannaert, Gilles Oorts, Jan Verelst
Normalized Systems Institute
University of Antwerp, Belgium
Email: herwig.mannaert@uantwerp.be

Koen De Cock and Jeroen Faes
Research and Development
NSX bv, Belgium
Email: koen.de.cock@nsx.normalizedsystems.org

***Abstract*—Many organizations are often required to produce large amounts of documents in various versions and variants. Though many solutions for document management and creation exist, the streamlined automatic generation of modular and evolvable documents remains challenging. The challenges are to some extent similar to the automatic generation of modular and evolvable software, which has been the subject of previous work on metaprogramming. In this contribution, a proof of concept architecture is presented to generate modular documents from runtime information systems through the use of a reduced runtime version of this metaprogramming environment. The configuration and integration of this expansion kernel into regular applications, and its use to generate some basic administrative document sources are explained. Based on this architecture, several use case scenarios are explored to generate other types of documents and artifacts using live runtime data.**

***Index Terms*—Evolvability; Normalized Systems Theory; Metaprogramming; Document Creation; Single Sourcing**

I. INTRODUCTION

This paper extends a previous paper, which was originally presented at the Thirteenth International Conference on Pervasive Patterns and Applications (PATTERNS) 2021 [1].

Organizations are often required to produce large amounts of versions and variants of certain documents. While they have traditionally focused their efforts into streamlining technical product documentation [2], they are now also looking to build business value by creating personalized customer-faced documents [3]. At the same time, current information systems are producing massive amounts of relatively simple documents, e.g., invoices and timesheets, based on corporate data.

The streamlined and possibly automatic generation of such documents has been associated with concepts like modularization and single sourcing [2], both reminiscent of similar techniques used in the creation of software. Similar to software development, dealing with versions and variants of modules may lead to so-called ripple effects, i.e., changes in certain versions or variants of modules may require changes in other modules. To facilitate such often necessary changes and to provide a desired level of evolvability, document structures need to be designed that deplete this rippling of changes [4]. Moreover, the use of parameter data during the instantiation of document variants seems similar to the inner workings of code generation environments.

In our previous work, we have presented a meta-circular implementation of a metaprogramming environment [5], and have argued that this architecture enables a scalable collaboration between various metaprogramming projects featuring different meta-models [6][7]. In this contribution, we investigate the use of a reduced version or runtime kernel of this code generation environment within the generated software applications. More specifically, we present a prototype implementation for the creation of evolvable artifacts, such as documents, where runtime data of the generated software application is used to instantiate the artifacts. The presented approach is neither confined to a specific domain, nor to a specialized type of software. However, the implementation is currently limited to the generation of relatively simple and straightforward documents.

The remainder of this paper is structured as follows. In Section II, we briefly discuss some aspects and terminology related to the creation and single sourcing of documents, an important class of artifacts created by information systems at runtime. In Section III-A, we explain the basic concept of Normalized Systems Theory with regard to the design of evolvable artifacts. Section III-B recapitulates the architecture of our meta-circular code generation environment, and explains that this expansion of source code artifacts is not limited to programming code. Section IV presents how a runtime kernel of this generation environment can be configured, and integrated into regular applications, to instantiate and expand runtime artifacts such as documents based on live data. Section V explores some use case scenarios to leverage this runtime expansion environment for the automated generation of various document types. Finally, we present some reflections and conclusions in Section VI, and discuss future work.

II. MODULAR AND EVOLVABLE DOCUMENT CREATION

While organizations have traditionally focused their efforts in document management into streamlining product documentation [2], there is a widespread belief that personalized customer-faced documents can build business value by enhancing customer loyalty [3]. However, repurposing internal documents to be used for online purposes, such as sales, marketing, product documentation and customer support has proven to

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

be diffcult. Moreover, it is hard to find any best practices or repeatable models developed that address this challenge [2]. In this section, we briefly discuss some techniques and issues regarding the creation of evolvable documents.

A. Document Creation and Single Sourcing

A successful approach to handle any complex system or problem is modularization [8][9]. An example of such an approach in the area of document management is *Component Content Management (CCM)*, defined as *a set of methodologies, processes, and technologies that rely on the principles of reuse, granularity, and structure to allow writers to author, review, and repurpose organizational content as small components* [2]. One of the fundamental ideas of component content management is the separation of content and layout [10]. The granularity of a component in CCM is defined by the smallest unit of usable information [11]. Several standards exist that define practical and technical implementation guidelines for creating modular and reusable content. According to Andersen and Batova [2], the most widely implemented standard is the *Darwin Information Typing Architecture (DITA)*.

Originally regarded as the broader discipline of CCM in the early 2000s, *single sourcing* has been defined as one of the fundamental aspects of CCM concerned with the design and production of modular, structured content. An elaborate description of single sourcing and its concepts, advantages, methodology, guidelines and practical examples, can be found in [12]. There are three fundamental aspects to single sourcing. First, content is made *reusable* by separating content from format. A second aspect is *modular writing*. Content is written in stand-alone modules instead of whole documents. This allows content to be assembled into documents from *singular source files that contain unique content*, the third aspect of single sourcing. Besides *assembling* the content modules into documents, i.e., combining source files in a hierarchical and sequential way with a distinct combination of audience, purpose and format, the modules need to be *linked*, i.e., connected to make them into coherent documents.

Enabling the content creators to focus on the actual substance of documents instead of having to deal with layout and publishing technologies, should lead to various advantages: saving time and money, improving document usability, and increasing team synergy [12]. Single sourcing recognizes two types of document creation. *Repurposing* entails merely reusing content modules for a different output format. *Re-assembly* on the other hand, is a more impactful way of reusing modules to develop documents for different purposes or audiences. Contrary to repurposing, re-assembly also includes changing the sequence of modules, the conditional inclusion, and the hierarchical level of inclusion.

B. Modular and Parametrized Document Generation

The emergence of concepts like modularization, CCM, and single sourcing regarding the management of certain classes of documents, e.g., technical documentation or personalized documents, is highly reminiscent of similar concepts in software codebases. Indeed, software developers have been striving for decades to modularize codebases, to separate concerns into singular source files, and to assemble source code modules into software applications, in a continuous effort — or quest — to reuse and repurpose these source modules. The component in CCM, defined by the smallest unit of usable information [11], seems to be consistent with the concept of a module in a software source base.

Both documents and software source bases can have successive *versions* in time that contain additions, corrections or omissions to its content, and can be branched into concurrent *variants* when variations in content and/or purpose occur. Just like in software, dealing with versions and variants of a document requires the design of document structures to provide a desired level of evolvability. Evolvable documents are documents that do not hinder or limit the application of changes made to their structure or content. They are free from ripple effects that would cause changes to the documents to be highly difficult and costly [4].

Documents, such as technical and/or personalized documents, can have many concurrent variants. In technical documents, these variants range from the variation or even conditional presence of entire technical descriptions and procedures due to differences in the components of various installations, to simple parameter values like the serial number, color, or location of the documented installation or product. But short and simple documents, like letters, invoices or timesheets, can also be considered to have many variants due to different parameter values. This aspect of parameter-based or *model-based instantiation of document variants*, is highly reminiscent of environments for *code generation in software development*.

III. Expansion of Evolvable Modular Structures

In this section, we discuss the expansion and assembly of evolvable modular structures. We introduce *Normalized Systems Theory (NST)* as a theoretical basis to design information systems —and conceptually other kinds of modular structures— with higher levels of evolvability, and its realization in a framework to generate and assemble programming code, and possibly other types of source artifacts.

A. Normalized Systems Theory and Evolvable Structures

NST was proposed to provide an ex-ante proven approach to build evolvable software [13][14][15]. It is theoretically founded on the concept of *systems theoretic stability*, a well-known systems property demanding 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. This implies that the impact of changes to an information system should only depend on the size of the changes to be performed, and not on the size of the system to which they are applied. Changes causing an impact dependent on the size of the

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

system are called *combinatorial effects*, and 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) [13][14][15]:

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

Applying the theorems in practice results in very fine-grained modular structures in software applications, which 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 that comply with these theorems. More specifically, NST proposes five *elements* that serve as design patterns for information systems [14][15]:

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

Based on these elements, NST software is generated in a relatively straightforward way. Due to this simple and deterministic nature of the 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 be complemented with custom code or *craftings* at well specified places (anchors) within the skeletons or boiler plate code. This results in the structural separation of four dimensions of variability [15][7]:

1) *Mirrors* representing data and flow models, using standard techniques like Entity Relationship Diagram (ERD) and Business Process Model and Notation (BPMN).
2) *Skeletons* expanded by instantiating the parametrized templates of the various element patterns.
3) *Utilities* corresponding to the various technology frameworks that take care of the cross-cutting concerns.
4) *Craftings* or custom code to add non-standard functionality that is not provided by the skeletons.

It has been extensively argued that the design theorems and structures of NST are applicable to all hierarchical modular architectures that exhibit cross-cutting concerns [16]. More specifically related to documents, the software theorems and element patterns of NST are very similar to the principles of CCM that rely on reuse and fine-grained modular structures to allow writers to author, review, and repurpose organizational content as small components, and to the concept of single sourcing, demanding the separation of content and layout. Moreover, it has been shown that the application of NST to the design of evolvable document management systems leads to architectures that are in accordance with the principles of CCM and single sourcing [4][17][18].

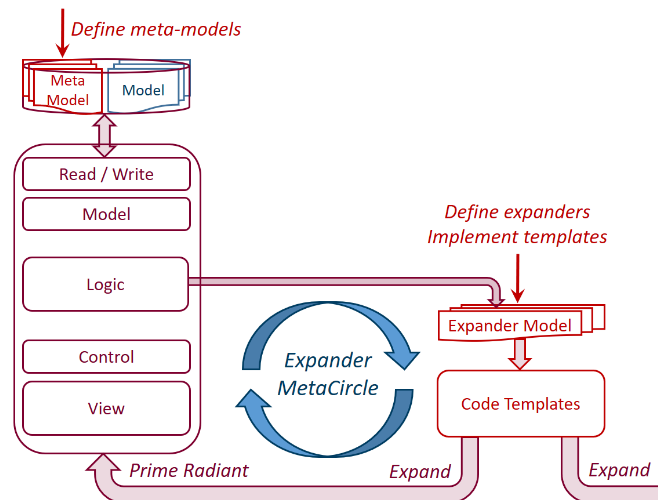

Figure 1. The meta-circular architecture for NS expanders and meta-application.

B. Meta-Circular Code Generation or Artifact Expansion

NST has been realized in software through a code generation environment to instantiate instances of the various elements or design patterns. Due to the simple and deterministic nature of this code generation, i.e., instantiating parametrized copies, it is referred to as *NS expansion*. We have also argued that nearly all metaprogramming or code generation environments exhibit a rather similar and straightforward internal structure [6][7], distinguishing:

- *model files* containing the model parameters.
- *reader classes* to read the model parameter files.
- *model classes* to represent the model parameters.
- *control classes* to select and invoke the generator classes.
- *generator classes* instantiating the source templates, and feeding the model parameters to the source templates.
- *source templates* containing the parametrized code.

As the NST metaprogramming environment was developed for the creation of web information systems, it has always included the generation of various building blocks, e.g., reader and model classes, which are similar to those of the code generation environment itself. This has made it possible to merge those generated code modules with the corresponding code generation modules, thereby evolving the metaprogramming environment into a meta-circular architecture [5]. This meta-circular architecture, described in [5][6][7], is schematically represented in Figure 1 and entails several advantages. First, this architecture enables the *regeneration of the metaprogramming code itself*, thereby avoiding the growing burden of maintaining the often complex meta-code, such as adapting it to new technologies. Second, it allows for a structural decoupling between the two sides of the code generation transformation, i.e., the domain models and the code generating templates. This also removes the need for contributors to get acquainted with the — basically non-existing —internal code structure of the metaprogramming environment, as additional

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

expanders with corresponding coding templates can be defined and activated using a declarative control mechanism.

We have argued in previous work that the meta-circular architecture presented in Figure 1 enables what we call two-sided collaboration [7]. By allowing the definition of interfaces at both ends of the code generation transformation, different groups of developers can collaborate simultaneously on the models and on the implementation templates. At the template side of the interface, templates are not limited to programming code containing statements and commands of programming languages like *Java* or *JavaScript*. Analogous to programming code, templates may be defined that contain commands and settings of markup languages and/or typesetting systems like *Markdown* or LaTeX, leading to the generation of non-code artifacts such as document sources. At the meta-model side of the interface, the definition of additional meta-models is not limited to programming structures either. Instead of representing software components, data and task elements, the meta-models may just as well correspond to hierarchical document modules such as chapters and sections, that will be realized by the corresponding commands and settings of the typesetting systems like *Markdown* or LaTeX.

Therefore, any model representing parameter data and/or small content components, may serve as a meta-model and drive the expansion or instantiation of the document. The meta-circular architecture does not require any explicit programming to support the new model entities representing the document. As we have seen, the various classes corresponding to the new model entities (XML readers and writers, model classes, control and generator classes) will be automatically generated. Analogous to the meta-circular *Prime Radiant*, the meta-application of the NST metaprogramming environment, a generated application that is based on such a new meta-model and allows the expansion of artifacts based on this meta-model, was originally called a *Secondary Radiant*. However, as the need for the creation of documents is nearly omnipresent in information systems, it is desirable to enable the creation of artifacts such as documents for all information systems and their models. Therefore, an architecture was designed to support the expansion of artifacts in every regular normalized systems application, treating every model as a meta-model. We refer to this architecture as the *Runtime Radiant*.

It should be noted that a meta-circular code generation environment is not necessarily complex. For instance, we have presented in detail how an elementary meta-circular code generation environment can be bootstrapped [19]. During this bootstrapping process, it is also shown how the generated code itself becomes able to generate other artifacts.

IV. Toward a Systematic Implementation for the Runtime Expansion of Documents

In this section, we present a prototype implementation of the *Runtime Radiant* architecture for the expansion of parametrized documents using the NST meta-circular code generation or metaprogramming environment.

A. Document Creation and Information Systems

As explained in Section II, an interesting duality exists between information systems and document creation. Information systems often support the creation of simple documents, such as invoices or timesheets, incorporating data that is entered and managed within the information system. At the same time, the streamlined creation of large amounts of document variants, for instance in the case of technical product documentation, requires some tooling to specify and manage the various parameters that drive the creation of the document variants. In other words, information systems often create documents, and document creation systems usually require a supporting information system.

For the prototype implementation targeted at the creation of documents using the NST meta-circular code generation environment, we have opted for the first scenario. The streamlined creation of variants of complex documents would require the definition of an elaborate meta-model describing the structure and domain parameters of the documents. The creation of such a model is out of scope of this contribution. However, as the creation of such a meta-model corresponds essentially to the design of a suitable data model or ontology, we feel that this does not pose a significant technological risk. Therefore, we decided to explore the generation of rather simple documents based on common data entities like invoices or timesheets. Nevertheless, this prototype implementation of the *Runtime Radiant* architecture does address a possible and important technological hurdle. As these documents need to incorporate runtime data from the live information systems, e.g., the actual details of the various invoices. Therefore, this proof of concept validates the expansion of artifacts based on runtime data from *any* information system expanded by the NST metaprogramming environment. In this way, the implementation can also serve as a validation for the expansion of other source artifacts based on live runtime data of information systems, such as marketing emails or sensor configuration files.

B. Declarative Control and Runtime Expansion

Consider two typical samples of a simplified data model for an administrative information system as presented in Figure 2.

- An *invoice* with some attributes, e.g., an invoice number and reference, containing a reference to a client, and consisting of several invoice lines.
- A *timesheet* with some attributes, e.g., the month and employee, containing a reference to a project, and consisting of several timesheet entries.

These data entities are expanded into *data elements*, collections of software classes as described in [7], by the NST metaprogramming environment, and incorporated in a web-based information system. The expanded data elements or collections of classes include:

- Reader and writer classes to read and write the XML data files, e.g., *InvoiceXmlReader* and *InvoiceXmlWriter*, *TimesheetXmlReader* and *TimesheetXmlWriter*.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

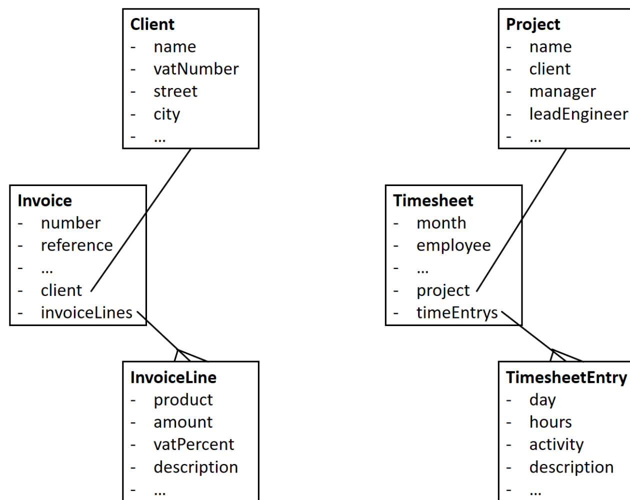

Figure 2. Samples of a simplified data model for an administrative system.

```
<expander name="TexInvoiceExpander"
  xmlns="http://normalizedsystems.org/expander">
  <packageName>net.palver.tex.invoice</packageName>
  <layerType name="ROOT"/>
  <technology name="COMMON"/>
  <sourceType name="TEX"/>
  <elementTypeName>Invoice</elementTypeName>
  <artifact>Invoice-$invoice.number$.tex</artifact>
  <artifactPath>$expansion.directory$/
                $artifactSubFolders$</artifactPath>
  <isApplicable>true</isApplicable>
  <active value="true"/>
</expander>
```

Figure 3. Declaration document of a *TexInvoiceExpander*.

- Model classes to represent and transfer the various entities, and to make them available as an object graph, e.g., *InvoiceDetails* and *InvoiceComposite*, *TimesheetDetails* and *TimesheetComposite*.
- View and control classes to perform *CRUDS (create, retrieve, update, delete, search)* operations in a generated table-based user interface.

In the same way that the instances of the NST meta-model data elements are read and made available as an object graph at the time of code generation, the instances of the data elements represented in Figure 2 can be made available as an object graph at runtime in a generated information system. Incorporating the core templating engine of the NST metaprogramming environment, as described in the following subsection, allows to evaluate the various attributes of the administrative data entities using *Object-Graph Navigation Language (OGNL)* expressions, and to feed them to the text templates, e.g., in LaTeX, that are used to create the invoice and timesheet documents.

As explained in [7], the expansion of artifacts, e.g., source code or document files, is based on a generic *ArtifactExpander* that uses declarative control to evaluate the model parameters and insert them into the source templates. Every individual expander generating a source artifact is defined in an *Expander*

```
<mapping
 xmlns="https://schemas.normalizedsystems.org/
               xsd/expanders/2021/0/0/mapping">
 <value name="info" eval="invoice.info"/>
 <value name="number" eval="invoice.number"/>
 <value name="client" eval="invoice.client.name"/>
 <value name="vatNr" eval="invoice.client.vatNr"/>
 <value name="street" eval="invoice.client.street"/>
 <value name="city" eval="invoice.client.city"/>
 <value name="isForeign"
  eval="!invoice.client.country.equals('Belgium')"/>
 <list name="invoiceLines"
       eval="invoice.invoiceLines"
       param="invoiceLine">
  <value name="info" eval="invoiceLine.info"/>
  <value name="product" eval="invoiceLine.product"/>
  <value name="amount" eval="invoiceLine.amount"/>
 </list>
</mapping>
```

Figure 4. Mapping document of a *TexInvoiceExpander*.

XML document. An example of the definition of such an individual expander to expand a LaTeX source file for an invoice is shown in Figure 3. It is quite similar to the declaration of an expander creating a Java source file during code generation, but has a `tex` source type, and uses for instance the runtime invoice number to construct the filename.

The evaluation of the various instance parameters or attributes is based on OGNL expressions [20] and defined in a separate *ExpanderMapping* XML document. This ensures the separation of content from format, as required by [10] to have reusable and evolvable documents. An example of the definition of such an individual mapping document for the invoice creation is shown in Figure 4. Besides simple OGNL expressions, it allows to evaluate logical expressions, e.g., whether the invoice client is foreign for VAT purposes, and to define lists of linked objects, e.g., invoice lines, and to loop through these lists and access the attribute values of the members of the list.

The values as defined in the expander mapping document are passed to the LaTeX templates. As described in [6], the NST environment uses the *StringTemplate (ST)* engine library [21]. This library supports the creation of a modular document structure by providing *subtemplate include* statements, enabling the document designers to adhere to the principles of single sourcing [12]. For instance, we share the declaration of various LaTeX packages and the definition of some basic commands through the use of the subtemplates `<basePackages()>` and `<baseCommands()>`. And the various invoice lines of an invoice (or timesheet entries of a timesheet) are created by instantiating a corresponding subtemplate for every list item through `<invoiceLines:invoiceTableLine()>` (or `<timesheetEntrys:timesheetTableLine()>`).

C. Integration in Normalized Systems Applications

A reduced version or kernel of the NST metaprogramming environment, incorporating the core templating engine, was packaged into a runtime installation that can be integrated

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

into every expanded normalized systems application. This runtime installation provides an interface to invoke expanders by passing instances of a tree of plain data classes, i.e., so-called *DTOs (Data Transfer Objects)*. The runtime kernel will *expand an artifact for every data tree instance that is passed to an expander (template)*.

The expansion of an artifact by the runtime kernel of the NST metaprogramming environment is schematically represented in Figure 5, both for expanding a Java software class for an *Invoice data entity* (left side), and for expanding a Latex document for an *actual instance of an invoice* (right side). In both cases, the expansion basically intertwines —the process is similar to a mathematical convolution— an instance of a data entity (tree) with a source template.

- An instance of an agent interface class for the *data element Invoice*, i.e., `InvoiceAgentIf`, is expanded by combining the instance data tree (including linked elements like *fields*) for the *data element* named `Invoice` with the template of the Java interface class `AgentIf`.
- An instance of a Latex invoice for the *invoice Inv-001*, i.e., `Inv-001TexInvoice`, is expanded by combining the instance data tree (including linked elements like *invoice lines*) for the *invoice* named `Inv-001` with the template of the Latex document `TexInvoice`.

Though the runtime expansion kernel can be used by any application that is able to provide (trees of) data instance classes, the integration in a normalized systems application is particularly straightforward and economical. First, the required (trees of) data transfer classes can be generated automatically by the NST metaprogramming environment. Second, an option has been introduced to generate a specific implementation class for a task element, that automatically feeds an instance of such a data class tree to a defined set of expanders. This expander set merely needs to be defined in a configuration file in the expansion resource. Such a resource is a standard library archive that contains the actual expanders, i.e., the triplets consisting of an expander definition, an OGNL mapping file, and a template. This task element can be invoked in a line of code, by a button in the user interface, but can also be embedded in a workflow driven by an NST flow element. In the latter case, the expansion of the document can be automatically performed for every instance of the target data entity for which a dedicated field has a specific value.

The integration of this runtime expansion kernel, the so-called *Runtime Radiant*, has been tested in a normalized systems business application that included (a more elaborated version of) the data elements represented in Figure 2. Based on live data from this operational production environment, many hundreds of LaTeX sources for invoices and timesheets were successfully generated through the use of the expander declarations and OGNL parameter evaluations as presented above. The processing of the Latex sources into *PDF* documents was integrated by embedding an additional task element in the workflow of the NST flow element.

It is clear that this expansion architecture allows information systems to create other type of document source artifacts based on live runtime data. Indeed, as the NST expansion environment is agnostic with respect to the source type, e.g., able to create LaTeX source documents in exactly the same way as Java source files, the generation of other types of document source modules is basically reduced to creating other types of templates and defining them in expander declarations. It is worth noting that such generated documents can have multiple remote impacts in our networked world. For instance, HTML documents generated by the runtime expansion kernel can be sent out immediately to large amounts of users through email or direct messaging. Or generated XML configuration documents can be uploaded automatically to remote Internet of Things (IoT) sensors or controllers, resulting in the flexible configuration of those devices.

V. Exploring Some Target Document Types for the Runtime Artifact Expansion

In this section, we explore the feasibility of several use case scenarios to apply and leverage the proposed architecture for the runtime expansion of various document types.

A. *Typical Information Systems Documents*

Though the automated creation of various small documents and reports is nearly omnipresent in contemporary information systems, we nevertheless believe that the additional possibilities and/or advantages of the proposed runtime expansion architecture can be significant.

First, the runtime expansion architecture brings a built-in capability for document creation to every piece of data of every information system that is able to provide (trees of) data transfer objects. Such a capability could enable end-users to define and author various classes of documents to be created, as opposed to the current situation where they have to either rely on specific types of documents that are supported by the information system, or use a dedicated application for document authoring and creation. While the former case could be limiting, the latter case could lead to consistency issues.

Second, the deep integration of the document expansion kernel into the information system, enables easy and structured access to possibly large areas of the data model. Consider for instance the generation of *Curriculum Vitae (CV)* documents. Though an enormous amount of applications exist to manage and generate employee CVs, the use of the runtime expansion architecture could give the document creation engine immediate access to a wide variety of personnel data, such as project involvement, completed training programs, and even performance appraisals. Moreover, it would intrinsically support the introduction of dedicated query logic, like the selection of the most relevant experiences for a target customer, based for instance on the industry sector and the project team size of past project involvements.

Third, the proposed architecture could provide a unified way to create a whole range of document artifacts. Besides

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

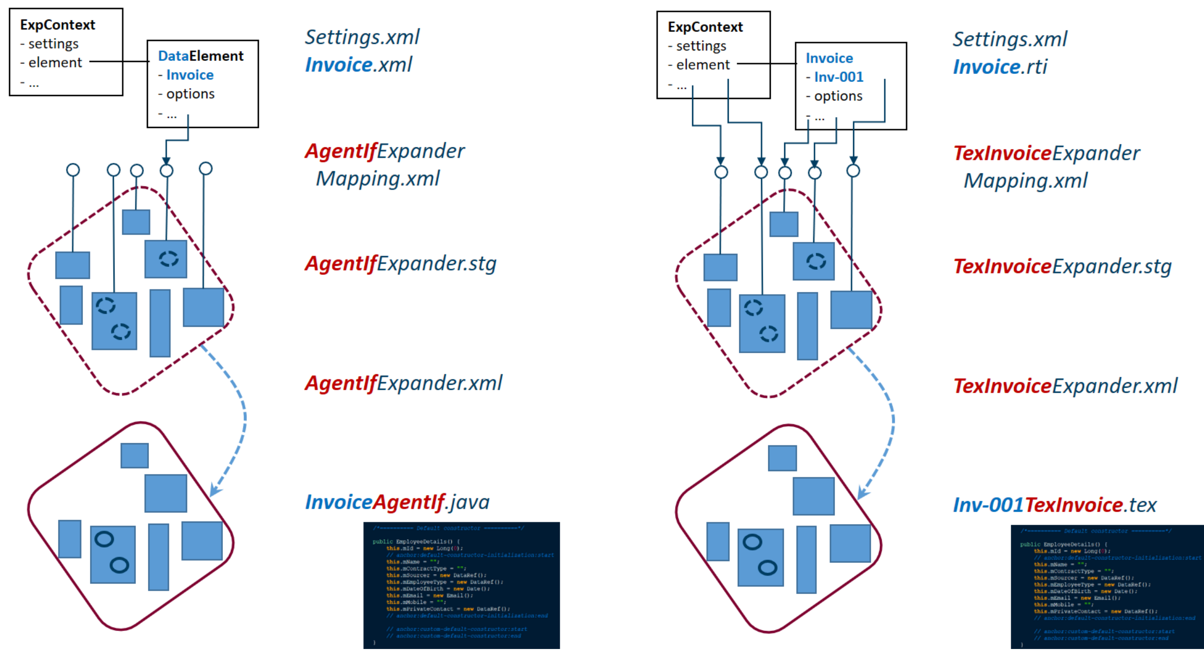

Figure 5. Schematic representation of an expansion of software class (left) and text document (right).

traditional documents like invoices and timesheets, we mention XML messages for financial reporting purposes or e-commerce transactions, and HTML documents for announcements and marketing campaigns. While these types of documents or messages are typically generated by other subsystems, a unified architecture could not only entail economies of scale, but would also contribute to consistency and single sourcing [2]. For instance, both the PDF invoice and the XML version based on the *Universal Business Language (UBL)* standard [22], could be based on the same data entries. Moreover, the proposed architecture could not only establish single sourcing across different types of content, but extend it to integration architectures between different information systems. Revisiting the CV example, one could for instance imagine to exchange CV data through machine readable structured document formats such as XML or JSON.

B. More Complex Hierarchical Documents

The advanced capabilities of a code generation environment with respect to hierarchical modular structures could be beneficial for certain types of more complex documents. The built-in hierarchical structure of the metaprogramming environment could enable the automated generation of documents with an elaborate and/or complex hierarchical structure, that require both different versions over time, and multiple variations for a given version, for instance depending on the profile of the target audience. In accordance with the detailed analysis of such documents with a complex hierarchical structure in [4], we mention the following examples.

- Detailed technical documentation of modular artifacts such as heat transformers or compressors, where different instances of the artifact may contain different (versions of) individual parts, and different descriptions and/or sections are appropriate depending on the target audience.
- Detailed security rules and guidelines for technical installations such as power plants, where guidelines often depend on regulations and legislation that evolve over time, and where specific instructions need to be tailored to individual installations and staff profiles.
- Elaborate assessment documents as required for self-assessments and external audits, where (versions of) various descriptions and/or sections, and even whether they need to be present at all, are in general dependent on the target audience and/or the scope of the assessment. A case related to such self-assessment documents for the evaluation of study programs is described in detail in [18].

For these types of documents, the proposed runtime expansion architecture could be an enabler for the structured hierarchical approach toward document creation described in [17][18].

C. Toward Integrating Multimedia Content

The integration of the document creation kernel in any information system could also facilitate the introduction of multimedia content. Having for instance such a system for the generation of CVs, would simply require the definition of one or more video clip data element(s) to enable the uploading of multiple types of personal message clips into the system. Such clips could both be bundled or integrated with the CV, and integrated through embedded links to the clips.

As discussed in a case study on a video learning channel [23], multimedia content itself could be hierarchically structured with different versions and variations. One could imagine

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

a runtime expansion environment to aggregate video lectures based on an hierarchical modular structure, selecting the appropriate content modules, language tracks and background themes. Provided the integration of some technical utilities, the runtime expansion system could combine the various modules and aspects into an integrated video lecture. Once again, this could foster collaboration between lecturers and content providers, while respecting the concept of single sourcing.

VI. Conclusion and Future Work

Many organizations are often required to produce large amounts of versions and variants of documents in areas like technical documentation and accreditation. At the same time, corporate information systems are producing massive amounts of relatively simple documents based on corporate data. The streamlined and possibly automatic generation of such documents has been associated with concepts like modularization and single sourcing, which are similar to techniques used in code generation software. As in software, dealing simultaneously with different versions and variants requires the design of document structures to deplete the rippling of changes in order to provide a desired level of evolvability.

In our previous work, we have presented a meta-circular implementation of a metaprogramming environment, and have argued that this architecture can be used for code generation based on different and even newly defined meta-models. In this contribution, we have investigated the use of this code generation environment within the generated information systems at runtime. More specifically, we have explored the creation of evolvable artifacts, such as simple administrative documents, where live runtime data of the generated software application is used to instantiate the artifacts.

To this purpose, we have packaged a reduced version or runtime kernel of the NST metaprogramming environment, and presented an architecture to integrate this expansion kernel into the runtime environment of every expanded information system. More specifically, we have shown how (trees of) instances of data transfer objects, containing runtime data, can be passed to source templates to generate artifacts such as document sources. The usage of this runtime expansion kernel in information systems to generate sources for administrative documents based on templates, only requires declarative definitions and OGNL evaluation mappings, and does not imply any dedicated software programming.

This paper provides different contributions. First, we validate that it is possible to use the NST metaprogramming environment to create another type of source code artifacts, e.g., document sources in some typesetting system. Moreover, we have explained that this implementation adheres to several fundamental concepts regarding modular and evolvable document creation, like CCM and single sourcing. Second, we validate that we can integrate a reduced kernel version of the NST metaprogramming environment into a runtime information system expanded by the NST metaprogramming environment, and to generate source artifacts from live data within this running information system.

Next to these contributions, it is clear that this paper is also subject to a number of limitations. First, we have only demonstrated the integration of the runtime expansion kernel into information systems generated by the NST metaprogramming environment. Second, the generated documents are quite simple, and in line with documents that are currently generated by mainstream information systems.

Nevertheless, this explorative proof of concept can be seen as an executable architecture, and we are planning future work to extend both the scope and the use of this environment in several ways. First, we intend to generate more types of document sources. We mention for instance XML-UBL documents for electronic invoices, *Fast Healthcare Interoperability Resources (FIHR)* for healthcare information exchange, and various XML documents to automatically configure sensors and controllers in energy monitoring and management systems. Second, we plan to significantly increase the use of this document generation environment. Besides the current production use for invoices and timesheets, we are considering its use for car policy documents, CVs, training certificates, meeting notes, et cetera. This usage would not only be confined to applications expanded by our NST metaprogramming environment, but will also be made available to other applications. Third, we intend to start addressing more complex hierarchical documents such as technical manuals and audit reports. To facilitate this use case, we have introduced in our tooling the possibility to define meta-models as ontologies, independent from our web application models. This will enable users to create document generation applications based on the structural models of their manual or reports, without having to deal with the technicalities of the traditional NST web information systems.

References

[1] H. Mannaert, G. Oorts, K. De Cock, and S. Gallant, “Exploring the use of code generation patterns for the creation of evolvable documents and runtime artifacts,” in Proceedings of the Thirteenth International Conference on Pervasive Patterns and Applications (PATTERNS), April 2021, pp. 17–22.

[2] R. Andersen and T. Batova, “The current state of component content management: An integrative literature review,” IEEE Transactions on Professional Communication, vol. 58, no. 3, 2015, pp. 247–270.

[3] S. Abel and R. A. Bailie, The Language of Content Strategy. Laguna Hills, CA, USA: XML Press, 2014.

[4] G. Oorts, Design of modular structures for evolvable and versatile document management based on normalized systems theory. Antwerp, Belgium: University of Antwerp, 2019.

[5] 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.

[6] 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.

[7] H. Mannaert, K. De Cock, P. Uhnák, and J. Verelst, “On the realization of meta-circular code generation and two-sided collaborative metaprogramming,” International journal on advances in software, vol. 13, no. 3-4, 2020, pp. 149–159.

[8] H. Simon, The Sciences of the Artificial. MIT Press, 1996.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[9] C. Y. Baldwin and K. B. Clark, Design Rules: The Power of Modularity. Cambridge, MA, USA: MIT Press, 2000.

[10] D. Clark, “Content management and the separation of presentation and content,” Technical Communication Quarterly, vol. 17, no. 1, 2007, pp. 35–60.

[11] F. Sapienza, “A rhetorical approach to single-sourcing via intertextuality,” Technical Communication Quarterly, vol. 16, no. 1, 2007, pp. 83–101.

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

[13] 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.

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

[15] 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.

[16] H. Mannaert, P. De Bruyn, and J. Verelst, “On the interconnection of cross-cutting concerns within hierarchical modular architectures,” IEEE Transactions on Engineering Management, vol. 69, no. 6, 2022, pp. 3276–3291.

[17] G. Oorts, H. Mannaert, and P. De Bruyn, “Exploring design aspects of modular and evolvable document management,” in Proceedings of the Seventh Enterprise Engineering Working Conference (EEWC), May 2017, pp. 126–140.

[18] G. Oorts, H. Mannaert, and I. Franquet, “Toward evolvable document management for study programs based on modular aggregation patterns,” in Proceedings of the Ninth International Conferences on Pervasive Patterns and Applications (PATTERNS), February 2017, pp. 34–39.

[19] H. Mannaert and K. De Cock, “Bootstrapping meta-circular and autogenous code generation,” in Proceedings of the Seventeenth International Conference on Software Engineering Advances (ICSEA), October 2022, pp. 87–92.

[20] “OGNL,” URL: https://en.wikipedia.org/wiki/OGNL, 2022, [accessed: 2022-06-15].

[21] “StringTemplate,” URL: https://www.stringtemplate.org/, 2022, [accessed: 2022-06-15].

[22] “Universal Business Language (UBL),” URL: https://en.wikipedia.org/wiki/Universal_Business_Language, 2023, [accessed: 2023-02-01].

[23] H. Mannaert, I. Franquet, C. Lippens, and K. Martens, “Exploring various aspects of a video learning channel : the educational case study of eclips,” International Journal On Advances in Intelligent Systems, vol. 12, no. 13-4, 2019, pp. 169–176.

2023, © Copyright by authors, Published under agreement with IARIA - www.iaria.org