Identifying manual changes to generated code: Experiences from the industrial automation domain

Robbert Jongeling*, Sachin Bhatambrekar§, Anders Lofberg§, Antonio Cicchetti*, Federico Ciccozzi*, Jan Carlson*
*Mälardalen University
Västerås, Sweden
{robbert.jongeling, antonio.cicchetti, federico.ciccozzi, jan.carlson}@mdh.se
§Tetra Pak
Lund, Sweden
{Sachin.Bhatambrekar, Anders.Lofberg}@tetrapak.com

***Abstract*—In this paper, we report on a case study in an industrial setting where code is generated from models, and, for various reasons, that generated code is then manually modified. To enhance the maintainability of both models and code, consistency between them is imperative. A first step towards establishing that consistency is to identify the manual changes that were made to the code after it was generated and deployed. Identifying the delta is not straightforward and requires pre-processing of the artifacts. The main mechanics driving our solution are higher-order transformations, which make the implementation scalable and robust to small changes in the modeling language. We describe the specific industrial setting of the problem, as well as the experiences and lessons learned from developing, implementing, and validating our solution together with our industrial partner.**

***Index Terms*—Model-based development, round-trip engineering, higher-order transformations, domain-specific modeling languages, industrial case study.**

I. Introduction

There, but what about back again? This question summarizes the context of the problem studied in this paper. We study a setting where currently, code is generated from a model, and then for various reasons that code is manually modified. Those code changes are currently not propagated back to the model, causing the model and code to be inconsistent. Nevertheless, for the maintainability of the model and code, both need to be synchronized. Hence, the context of this work is how to go back again, from the code to the model. We study the first part of this challenge, to identify the manual changes that have been made to generated code deployed on PLCs.

The context of this study is a concrete model-based development (MBD) setting at a company operating in the industrial automation domain. Our work is part of an industry-academia collaboration in Software Center[1], which involves 5 universities and 15 companies working together to accelerate the adoption of novel approaches to Software Engineering. The remainder of this paper reports on our collaboration with this company and hence is specific to this concrete setting.

MBD can refer to many forms of software development, but the common denominator is that they all involve models as core development artifacts [1]. In one of its forms, MBD brings the benefit of complete code generation from models. In a one-way code generation process, changes to the model have the consequence of the code being re-generated, overwriting the existing one. There have been relaxations of this principle to preserve manual changes to generated code in future generations: notably, protecting regions to prevent them being overwritten when code is re-generated is done in (among others) the Epsilon Generation Language [2]. More interestingly, code generation can be part of a round-trip process, where not only code is generated from a model, but also changes to that code are reflected in the model to keep them consistent.

Model-code synchronization in the context of round-trip engineering is a classic MBD problem [3]. Moreover, specifically in the industrial automation domain, back-propagation of manual code changes to the model has been identified as an open research problem [4]. Generated code can be modified for several reasons, for example, to be completed (in case of generated skeleton code), to be fixed after code-level verification, or to be optimized (in cases where performance is crucial). We consider another option, where the generated code is modified to customize the software for managing variants of the deployment domain. We study a setting in the industrial automation domain where modifications are made to generated code after it has been deployed on target PLC devices. In this setting, consistency between the model and generated code is lost and should be restored to improve the maintainability of the software as well as the management of different software variants.

A first step to restoring that consistency is identifying what manual changes were made to the generated and deployed code. It turns out to require a significant pre-processing of code files to allow their comparison. We provide a generic solution for this process based on higher-order transformations.

The remainder of this paper is organized as follows. We provide a detailed description of the industrial setting and the studied problem in Section II. We then discuss works related to the problem domain in Section III. The developed approach and implementation are then described in Section IV

[1] https://www.software-center.se

and Section V respectively. Experiences from evaluating the approach in the industrial setting are included in Section VI. A discussion of modeling lessons learned is included in Section VII before the paper is concluded in Section VIII.

II. Experienced problem

Overall research goal:

- Our overall goal is to identify the manual changes made to the generated code after deployment;
- The context of this change identification is to eventually enable a model-code round-trip in the studied industrial setting;
- To justify further investment in automating back-propagation, a proof-of-concept is required for the identification of changes made to deployed code.

We study an instance of the model-code round-trip problem in an industrial setting. The current round-trip consists of four steps: (i) creating (or changing) a model, (ii) generating code from the model and deploying it on target devices, (iii) making changes to the generated code, and (iv) reflecting those changes in the model. In our setting, the first three steps were in place, i.e. a model is created, code is automatically generated from the model, the code is deployed on PLCs, and then the code is manually changed. The fourth step is also in place, but it is done manually and is therefore very labor-intensive and error-prone. Therefore, we focus on providing automated support for reflecting code changes to the model, thereby synchronizing model and code.

Back-propagation of code changes to the model can be separated into two phases, first identifying the code changes and then reflecting those changes in the model too. In our setting, significant engineering effort is required for the second phase, and therefore, it is important to first study the feasibility of the first phase. Therefore, we limit our scope in this first step to providing engineers with an overview of changes between the manually changed code and the model, thereby facilitating the (for now) manual back-propagation of these changes and later allowing for this back-propagation to be further automated.

A. Industrial setting

Motivation summary:

- Engineers model the desired behavior using a DSML in an in-house tool;
- Code generated from the model is complemented with libraries to create executable programs;
- These executable programs are deployed on PLCs;
- After deployment, on-site engineers may customize code to support particular product variants;
- To improve the software's maintainability, code and model must be synchronized;
- To do so, we provide a way to compare manual changes to generated code with the model.

We worked in close collaboration with a global food and packaging company in the industrial automation domain. We describe a typical setting encountered in their development of software for parts of packaging systems. In our setting, models are created in a custom domain-specific modeling language (DSML) to design programs for programmable logic controllers (PLCs). The PLC code is IEC61131 [5]. The DSML is developed in C# and is tightly integrated with an in-house tool, which is used to create the models and also includes functionality to generate PLC code from these models. Models describe, e.g., the behavior of control modules. Several different PLC types can be targeted using a single model, using different code generation logic for each targeted PLC type. Although different models exist for different factory layouts, the unique characteristics of each factory may (and typically do) require on-site modifications to the PLC code.

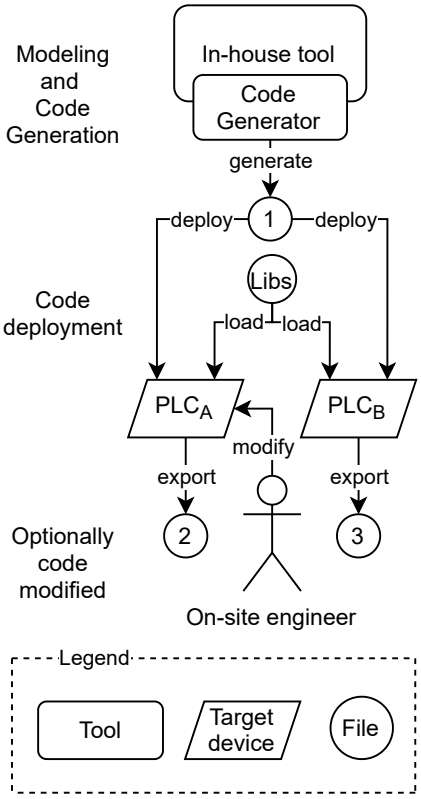

Fig. 1. Overview of industrial setting.

Figure 1 schematically shows the industrial setting under study. Models are defined in the in-house tool and code is generated from them. The generated code ① does not constitute a complete executable program. Instead, it is supplemented during deployment with data types and standard routines, packed in common libraries to enable their re-use. Together, the generated code and library code form programs that are executed on targeted PLCs. In Figure 1, we show two examples of PLCs, of potentially many PLCs of a few different types. After deployment, on-site engineers may modify programs directly on the PLCs, to customize programs for the specifics

of a particular factory. The running programs can be obtained by exporting them from the PLCs, yielding exported files ② and ③. From now on, throughout the remainder of the paper, we consider the example of the modified file ②, since it illustrates the workflow of interest: generating code, deploying it on the PLC, and modifying it. We study how to identify the manual changes made in ②.

It is natural to wonder why the generated code is being modified at all instead of the model being modified and the code regenerated from it. The current company process is to customize the systems on-demand, on-site, by engineers directly on the PLCs, who do not have access to the models. The engineers devising the general models do not have the required knowledge about the on-site characteristics that require tweaks to the generated code. Conversely, on-site engineers do not have access to the model to make the required changes. Moreover, this approach is aiming not only to be used in future settings but also to perform this differencing approach for many systems currently operating in the field, and to which already manual changes were made. Our approach needs to integrate with this way of working.

To summarize, we classify the setting according to the three-dimensional taxonomy for bidirectional transformation by Diksin et al [6]. The overall setting can be classified as organizational asymmetric, since we are interested in propagating changes only in one direction, from the code to the model. In the current way of working, there is in total a single round-trip. The model is created, the code is generated, the code is modified, and then the model is updated to be an accurate view of the code. The code is a refinement of the model, the code contains at least all the information that is also in the model (and more, since it is supplemented with libraries and code generation artifacts). Hence, there is an info-asymmetry between the model and code. Finally, the approach requires incrementality, since we do not want to recreate a model from scratch after changes to the code but instead want to propagate the code changes to the model.

B. Identifying code changes

In our setting, changes are being made to the generated and deployed code to customize the programs for specific purposes. Such changes include, e.g., adding instructions, renaming parameters, and configuring different values for control modules. Changes to sequence logic, parameters, and control modules in the PLC are the most relevant for back-propagation, but we do not distinguish in our approach between them. Synchronizing code and model after these types of changes is not trivial, for the following main reasons.

a) Differencing is challenging: We must first identify what code changes have been made. A first challenge is that directly calculating the difference between the model in the in-house tool and changed code (② in Figure 1) is not possible due to their different syntax. To address the syntax misalignment, an intermediate format can be chosen, where the changed code ② is compared to the unchanged (generated) code ①. Both these artifacts are XML files, albeit conforming to different XML *schemas* (metamodels). Hence, comparing these files is still not an easy task.

Consider an unmodified exported code file ③ from a PLC. During code generation and deployment, the original ① is supplemented with additional information from the common libraries, and any other changes induced by the PLC export. Hence, the code file ③ eventually contains more information than what was in the model, and a diff between ① and ③ would be non-empty, even though we did not make any manual modifications to the code. A comparison of file sizes (a line-based comparison is rather meaningless given the completely different structure of the files) shows that the PLC export can be up to 2.5 times as large as the non-deployed generated code (2.6MB vs 6.4MB). In terms of XML nodes, the difference is, the PLC export file contains about 2.8 times as many nodes (21283 vs 59551). So, comparing ① and ③ directly does not make sense since they are conforming to different metamodels. Moreover, ③ does not even contain the manual changes made to the PLC code, as present in ②, yet. Hence, to obtain a meaningful difference between the generated code ① and the exported code ② or ③, we must first prepare the files such that both sides of the comparison are conforming to the same XSD schema. Specifically, the exported files must be filtered to avoid marking irrelevant information as manually made changes.

> We target an approach that allows comparison of ①, generated code, and ②, code exported from the PLC, such that this comparison reveals only the manually made modifications to the generated code.

b) Code-level and model-level must be bridged: A second challenge is that, once differences are identified at the code level, they must be lifted to the model level. This is especially challenging due to the freedom of changes in the code. Essentially, any change can be made in the generated code, but not all changes may have sensible ways to be reflected in the model, such as a re-ordering of instructions in generated code, or added instructions to code imported from the common libraries. In general, only those code changes that are directly related to model elements can be propagated back to the model.

Another challenge with lifting identified changes to model-level is related to interpreting the calculated delta between the code files. Since we only have access to the result of code generation and PLC exports (that are made once in a while), any information on how the manual changes are made is not available. Hence, instantaneous synchronization approaches cannot be applied. Instead, we compare the code before and after deployment to the PLC and possibly undergone modifications. Consequently, there is no way to distinguish between e.g. a rename and a deletion-addition of an element. Nevertheless, at the model level, this distinction can be very relevant.

Moreover, several changes spread throughout the code might be a single change on the model level. In a general case, it

is very challenging to group the changes on the code level to represent meaningful changes at the model level. In our setting, what can at most be obtained from the code is a fully qualified path to each changed model element and the change that was made to it. Therefore, we approach this issue by considering such atomic code-level changes and mapping them each to changes at the model level, thereby accepting that we may miss more meaningful changes at the model level.

C. Motivation for automation

In our setting, the described back-propagation is currently done manually. It takes a considerable effort to first identify the changes in the code and then to implement them in the model. Furthermore, this effort needs to be repeated for each of several dozens of projects, and this number is expected to only grow in the future (since old projects must still be maintained). Automation of the back-propagation work would thus save a significant amount of manual effort.

III. Related work

A. Various synchronization approaches

Famously, FUJABA provides a model-code-model round-trip between UML and Java [7]. Also, further work has provided an approach for the generation of IEC61131 code from UML [8]. However, our work focuses not on UML but rather on a round-trip between models in a DSML, integrated with an in-house tool at the company.

We are establishing a reverse engineering process from code to model. Several approaches have been created for similar purposes, such as template-based reverse engineering approaches [9]. Crucially, in our setting the code changes are freely performed, they are not necessarily limited to specific portions of the code, or particular modification actions.

In their position paper, Sendall and Küster remark already that round-trip engineering is fundamentally about ensuring consistency between the model and code, rather than just transforming one artifact into the other [10]. In our case, we in particular aim to synchronize model and code by bringing changes from the code level to the model level. Given the industrial setting, this synchronization is rather sporadic, and hence continuous synchronization is not applicable. Instead, we consider on-demand propagation of these changes, similar to Antkiewicz and Czarnecki, who propose a category of DSMLs that can intrinsically provide model round-trip support due to essentially backward and forward model transformations [11]. As we will see in Section IV, our main effort is not so much in transforming the code to the model, but rather in cleaning the code such that it can be compared.

Giese and Wagner noted the similarity between model synchronization and inconsistency resolution and utilize triple graph grammars for establishing bidirectional transformations in a model synchronization setting [12]. In our synchronization approach, we aim to only consider the delta (changed part of the code) for back-propagation, rather than instantiating a complete (new) model from the code. We aim to reflect manual code changes in the model so that later the code generated from that model again contains those manual changes. This is a different concept from preserving manual code changes upon model evolutions and next code generations, as in e.g. [13]. The way of working is hence to first complete the round-trip and then possibly make changes to the model before re-deploying the code.

As we will discuss in Section IV, we base our approach on higher-order model transformations implemented in XSLT. Two other approaches using XSLT are created are mentioned in an overview paper of usages of higher-order model transformations [14]. Most approaches in that paper (from 2009) are based on the ATL transformation language.

Other research has focused on reverse engineering for related efforts. For example, Fleurey et al. [15] migrate a legacy codebase to a new language. Later, MoDisco was developed as a more general framework for that type of reverse engineering [16]. There are many efforts related to obtaining diagrams from code, e.g. reverse engineering a UML model from Java code. Such approaches have been adopted into commercial tools now too, e.g. IBM Rational Rhapsody and Softeam Modelio provide reverse engineering and model-code consistency features respectively. As we have discussed, the application domain of our work requires us to devise a different approach than these available ones.

B. Industrial case studies

The synchronization of manual changes to the generated code with the input model has been rarely addressed in the industrial automation domain and generation of IEC 61131-3 code [4]. Consistency management scenarios like ours have been studied in other domains too, many of them UML-based [17].

Several of these studies are related to inter-model consistency management, often utilizing triple-graph grammars to express bidirectional transformations between models [18]. Another study focuses on preserving extra-functional properties in model-code round-trips and evaluates the approach in an industrial setting [19]. Typically different industrial settings require a different focus of the round-trip approach. Ciccozzi et al. report on lessons learned in a round-trip setting with the focus of improving models and regenerating optimized code based on code execution monitoring [20]. In our setting, the focus of the round-trip is on restoring the consistency between the model and executed code. Similar challenges can be noted in metamodel-model co-evolution scenarios. Durisic et al. perform an industrial case study of an evolving domain-specific metamodel and its impact on related development artifacts [21].

IV. Developed approach

In this section, we detail our method to identify differences between the generated code and the manually changed deployed code (① and ② respectively in Figure 1). The first subsection details the assumptions derived from our industrial setting. The second subsection details the steps of

our approach. Our implementation and examples are included in Section V.

A. Assumptions and starting points

In the setting, the modeling tool and code generator already exist, both implemented in C#. Now, at a later time, it is decided to investigate the possibilities of reflecting the manual code changes in the models. To avoid duplicating the existing code generation functionality, we decided to not aim for a bidirectional transformation but instead focus on an approach that can immediately be applied within the existing way of working. While the eventual goal is to synchronize the model and code, an important side-effect of this synchronization should be for the company to get insight into what manual changes have been performed, especially those changes that were made a long time ago. Therefore, we do not aim to instantiate the model from the code but instead will consider an approach to identifying the changes by comparing the generated and deployed code.

We aim for an approach that is as generic as possible within the industrial setting as described in Section II. The approach needs to work for various types of targeted PLCs and needs to anticipate possible future evolution of the DSML. To preserve the genericity, we avoid assumptions about the DSML as much as possible. We do however base our approach on the syntactical format of the generated code and exported programs, i.e., XML. This is a pragmatic choice and inspired the direction of our implementation, but it is not necessary from a conceptual point of view. As described later, the mechanism to allow for comparisons of generated and manually changed code can in principle be applied to code with other textual syntactical representations.

To better classify the setting and type of solution, we refer to the framework by Anjorin et al. [22]. Our application scenario is initial-state-based, since we are comparing two states of artifacts, we do not have a delta as input. Within this scenario, we aim for a backward consistency restoration, that can propagate the manual changes from the code back to the model by first identifying the delta that was applied to the code.

B. Proposed solution

The task at hand is to compare the generated code from the model, and a version of this code exported from the PLC, containing possible manual modifications. We aim to identify the manual modifications and not any other differences between these artifacts introduced during the process of code generation, deployment, and exporting of the files. This means that before any comparison between these two files, some filtering is required to ensure that we are indeed comparing models conforming to the same metamodel, here, the same XSD schema. Moreover, in addition to manual modifications, there are other artifacts that are not relevant for back-propagation because their source is from the code generator or from libraries, this information should also be filtered before comparison of the files.

A schematic overview of the mechanisms we employ to achieve this filtering and comparison is shown in Figure 2. Our overall plan consists of the three steps A, B, and C. Step A (comprised of A_1 and A_2) and Step B (comprised of B_1, B_2, and B_3) prune from the generated code ① and the manually modified code ② all those elements that are not relevant for the differencing at the model level. For consistency, these ① and ② in Figure 2 correspond to ① and ② respectively as in Figure 1. Step C compares the pruned files to identify the differences between the two files, resulting in a delta (Δ). The changes listed in this delta should be propagated back.

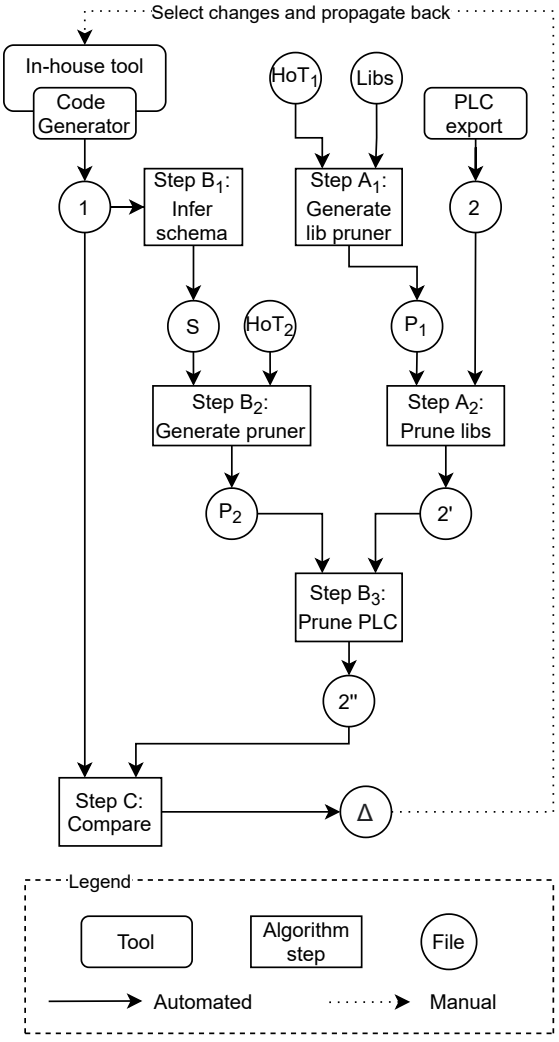

Fig. 2. Overview of our approach for comparing generated code ① and manually changed code as exported from the PLC ②, by first filtering ② to obtain ②' such that it conforms to the same XML schema as ① and therefore can be compared.

a) Steps A_1 and A_2: Prune common libraries: To express the generated code ① and the manually modified code ② in the same form, the first step is to remove from ② all information added by the common libraries during loading of the program onto the PLC.

In Step A_1, we propose to utilize a first higher-order

transformation (HoT), i.e., a transformation that itself outputs a transformation. The choice of an HoT for this step is based on the flexible formats of ① and ②. The exact formats can differ across different PLC types, and even for the same PLC type, over time there can be different formats of ①. Therefore we did not want to assume too much about the metamodels and instead rely on the model instances to derive the required filtering mechanism. A survey on HoTs classifies our type of HoT as transformation synthesis since we output a model transformation and the input to the HoT is not a model transformation [14].

HoT_1 is applied on a file containing the common libraries in the same XML format as encountered in the code files. Given these inputs, a transformation can be created that keeps all XML nodes except those occurring in the common libraries. The result is a transformation, here referred to as pruner, p_1. Pruner p_1 is used as input for Step A_2, where it is applied on ②, to obtain a first pruned version of this file, ②′.

b) Steps B_1, B_2, and B_3: Prune PLC-export: In addition to pruning away information added by the common libraries, pruning of PLC-export-specific elements is required for ②. As a general way of finding out what elements should be kept and which should not, we base our pruner on the generated code ①. We reason that any element of interest to be back-propagated to the model must at least be present in ①. And so, if any new type of element occurs in ②, then that element type may not be known in the model and should therefore be pruned because it can not be back-propagated. There is a small risk here that a small or incomplete program can not be extended because it does not contain any of the to-be-added types, but in practice, this does not happen. Any deployed code on the PLC includes the complete set of types of elements as known in the model. To summarize, we assert that all XML elements that do not fit the schema of XML file ① are not interesting for back-propagation to the model.

Therefore, we start a three-step process (B_1, B_2, B_3) to prune away the PLC-export-specific information from ②. In Step B_1, we first infer the schema of the XML file ①, which is exported to an intermediate file Ⓢ. In Step B_2, we use a second HoT, Hot_2, and apply it on the Schema Ⓢ. This HoT_2 is designed to, given an XML schema as input, output a transformation that preserves all concepts as defined in that schema, and prunes away all additional XML elements, tags, or attributes. The result of Step B_2 is a second pruner, Pruner P_2, that removes from any input XML all elements not fitting Ⓢ. Step B_3 consists of applying Pruner P_2 to ②′, to obtain ②″. The latter is an XML document from which all additional information related to code generation, PLC deployment, and PLC export has been pruned away.

c) Step C: Compare and diff: Now, we have removed all elements from ② that were due to libraries and otherwise do not conform to the schema ⓢ and have obtained ②″ which is in the same format as ① (they conform to the same XSD schema, allowing their comparison). When we compare these two files ① and ②″, we expect to detect only those changes that were made manually to the deployed code.

This comparison is done on XML files and hence a line-based comparison is inadequate. Merely a line-based diff could highlight changes to children and attribute orders, inserted comments, and other irrelevant syntactical changes that do not change the semantics of each of the programs. To find changes related to the semantic difference between the files, the applied differencing mechanism must be able to ignore these irrelevant changes, in other words, it should be XML-aware.

After the comparison, we export the detected differences in two ways. The first is an HTML file providing a side-by-side overview of the files, highlighting their differences. This file is primarily useful for manual inspection of the result. The second is an XML file defining the change operations as applied on the XML elements, which is particularly useful as a summary of the changes to be back-propagated. Furthermore, the XML format of this file allows for it to be parsed and interpreted by the in-house tool for eventual automation of the back-propagation. The result of the comparison is indicated as Δ in Figure 2.

Approach summary:

- Comparing the two code files requires superfluous information to be pruned from the deployed code;
- We provide an approach relying on higher-order transformations that generate these pruners based on the structure of the generated code, and of the added libraries;
- Once in the same format, an XML-aware diff is performed to find the differences between the code files, which are the manual modifications made to the generated code.

V. Implementation

In this section, we provide details on the implementation of all the steps introduced in Section IV. Furthermore, we highlight cases where the implementation deviates from the proposed method.

All transformations are implemented in XSLT [23]. XSLT is a language for the manipulation of XML documents. Since XSLT is itself conforming to XML, the output of an XSLT transformation can also be an XSLT transformation, hence it can be used to create HoTs. Besides XSLT, the rest of the implementation is done in C# to align with the codebase already in place at the company. Furthermore, implementing in C# allows the use of Microsoft `System.Xml` and `XmlDiffView` libraries for the eventual comparison of the pruned files. Finally, staying very close to the XML representation allows us to deal with potentially not well-formed files.

To illustrate the implementation in action, we run it on two files, provided by the company, that correspond to ① and ② in Figure 2. The sizes of these files are 7.8MB and 8.8MB.

Before any pruning actions, ① contains 68k XML nodes and ② contains 88k XML nodes (29% more than ①).

a) Steps A_1 and A_2: Prune common libraries: We started our implementation of the library pruner by creating an HoT. The input to this HoT is an XML export of the library and the output is a transformation that copies all nodes, except for those that are in the library export, in an XML file. This conceptually works fine for matching all library elements, however, in practice, we noticed some shortcomings that required manual modifications to the generated transformation.

The structure of the library pruner transformation is as follows. For each type of element that should be pruned, we create an `xsl:template` that ignores matching elements. For example:

```
<xsl:template match="Path/To/Filtered/Element" />
```

The result of applying this template is that all matched elements are pruned because they are not copied to the output.

The opposite can also be applied, a template can match all elements that should be kept and then in its body simply copy those elements. For example:

```
<xsl:template match="Path/To/Kept/Element">
    <xsl:apply-templates select="@*|node()"/>
</xsl:template>
```

Here, all matched elements are copied rather than pruned.

We incrementally improved the generated transformation by identifying all elements that should be pruned. The implementation thus deviates from the proposed method and is less generic than we had hoped. A clear definition of which elements are introduced during code generation and PLC export could help in separating again the pruners, leading to a more generic solution.

Examples of pruned elements are those starting with certain prefixes, that indicate their origin from the shared libraries. In addition, changes to elements that did not originate from the libraries, but rather from exporting code from the PLC, are not considered for back-propagation either. For example, we pruned all trailing ".0" values from numbers. The third category of elements was decided to be pruned after inspections of early diffs revealed them as useless, such as empty descriptions of elements.

A final interesting detail to mention is that, before running the transformation described above, we run the identity transformation on all XML files, to get rid of CDATA tags, which do not contain any XML.

After Steps A_1 and A_2, the size of the PLC export file has decreased from 88k XML nodes to 81k XML nodes. Still, there are many nodes left to prune that are related to the PLC export itself.

b) Steps B_1, B_2, and B_3: Prune PLC-export: For Step B_1, we use `InferSchema` from `System.Xml.Schema`. If the code is well-formed, the function returns a single XSD containing the schema Ⓢ of the XML file ①. This schema is then used as input for Step B_2.

Also for this purpose, we created an HoT; the aim is to transform the XML file as exported from the PLC, such that it conforms to the same XSD schema as the XML exported from the in-house tool. The HoT creates an `xsl:template` for each encountered element in the input schema Ⓢ. As an identifier to match these templates, it constructs the fully qualified name (FQN) of each XML element in the schema. In that way, there cannot be any accidental duplicate matches for nodes with the same name in different places of the hierarchy. Listing 1 shows a snippet of this pattern. The HoT also makes sure to include all attributes, children nodes, and values.

```
<xsl:template match="xs:element">
    <!--the "axsl" tag is used to output an "xsl" tag,
        allowing for the transformation to itself output a
        transformation -->
    <axsl:template>
        <!-- This creates a "fully qualified name" of all
            ancestor xs:element names, separated with '/',
            followed by a '/' and then the name of the
            current node. -->
        <xsl:variable name="ancestors" select="ancestor::xs
            :element"/>

        <!-- fqn will contain the ancestor names, separated
            by, and ending with a '/' -->
        <xsl:variable name="fqn">
            <xsl:for-each select="$ancestors">
                <xsl:value-of select="@name"/>/<!-- Comment
                    prevents line breaks in output -->
            </xsl:for-each>
        </xsl:variable>
        <!-- Appending the name of this node itself to
            finalize the match attribute -->
        <xsl:attribute name="match">
            <xsl:value-of select="$fqn"/><xsl:value-of
                select="@name"/>
        </xsl:attribute>
```

Listing 1. Snippet of HoT, creating `xsl:template` for each element in input schema

The HoT is rather generic, but it is inspired by the structure of the schemas present in our industrial setting. For example, it assumes a single tree structure for each XML file. Furthermore, it copies the non-tag content between tags only if the element does not contain any children. This assumption is derived from studying several examples and may not hold in the general case.

The overall implementation is very compact due to the powerful expressions of HoTs. Less than one hundred lines of XSL were needed to express the HoT implementing the second pruner. Another powerful benefit of this HoT is that it likely will not need to be updated when the DSML in the in-house tool evolves.

After Steps B_1, B_2, and B_3, are carried out, the size of the PLC export file has decreased from 81k to 73k XML nodes. Upon manual inspection, it became clear that more empty nodes could still be pruned away. These were not pruned in earlier runs of the Steps A_1 and A_2, because the nodes were emptied during Step B_3. Therefore, we extended our implementation to prune this file further by applying again the pruner from the Steps A_1 and A_2. The result is a further reduction of the XML file from 73k to 68k XML nodes.

c) Step C: Compare and diff: In this step, we rely on Microsoft's `XMLDiffView` library. For calculating the differences between the two XML files, we select various options to ensure finding only relevant semantic changes and exclude purely line-based changes. Notably, we ignore

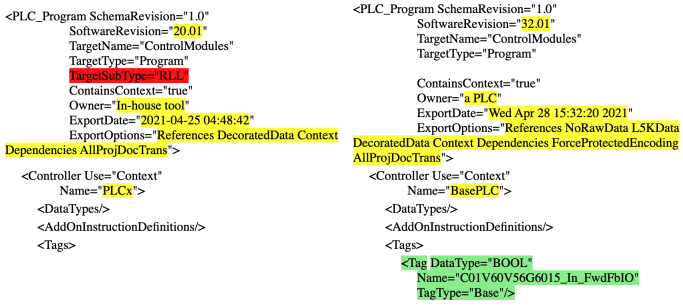

Fig. 3. Side-by-side display of XML documents with highlighted differences. Color legend: yellow indicates a change, red a deleted element, and green an added element.

the order among children XML nodes. Note that in some situations, the children's order is relevant, for example in the case of XML nodes representing rungs in ladder logic. However, we also have a sequential numbering of those rungs in our code, so we can rely on those for deriving the ordering and ignore their actual order of appearance in the XML file. We also ignore some other XML artifacts that are irrelevant for our comparison, in particular comments, white spaces, namespaces, and prefixes.

We create the output in two different formats, due to the `XMLDiffView` library. The first is a side-by-side overview of the compared XML files, in which changes are highlighted in various colors to indicate the action that caused the difference (change, add, delete, move). We added a style-sheet to the generated HTML to enhance the viewing ease, see Figure 3 for an example snippet of the resulting difference view. Understandably, the engineers responsible for back-propagating the changes typically prefer not to read XML, but to be presented with a more human-readable list of the changed elements. Such a list can be obtained by parsing the second difference view created, which is an XML list containing only the changed elements. A snippet of this view is shown in Listing 2, depicting the file portion as in Figure 3. While the size of the complete HTML file grows considerably (66.6MB) due to it containing the entire program twice, together with markup information, the XML list of changed elements is smaller: 245KB. The total number of added XML nodes was 1k, whereas about 2.8k of XML nodes were changed. Note that this difference view creates an added node for each added child element. Hence, the example addition in Figure 3 is denoted as four separate additions in the XML list, one for each attribute and one for the Tag itself.

While the HTML file is mostly useful for checking the results focusing on human readability, the XML list is aimed at future work on automating back propagation more. To lift the identified code-level changes to the model-level is relevant for manual inspection, but less so for automated back-propagation. An automated solution benefits from the explicated hierarchy in this difference view. It can translate the code changes one-by-one into actions to be done at model level, using the information provided by the full path to each node.

```
<PLCProgram SchemaRevision="1.0"
  SoftwareRevision="xd_ChangeFrom('20.01')To('32.01')"
  TargetName="ControlModules"
  TargetType="Program"
  TargetSubType="RLL"
  ContainsContext="true"
    Owner="xd_ChangeFrom('In-house tool')To('a PLC')"
    ExportDate="xd_ChangeFrom('2021-02-01 04:42:42')To('Wed
        Feb 03 15:32:20 2021')"
    ExportOptions="xd_ChangeFrom('References DecoratedData
        Context Dependencies AllProjDocTrans')To('
        References NoRawData L5KData DecoratedData Context
        Dependencies ForceProtectedEncoding AllProjDocTrans
        ')">
    <Controller Use="Context"
      Name="xd_ChangeFrom('PLCx')To('BasePLC')">
      <Tags>
        <xd_='Add(node)'Tag DataType="BOOL"
          Name="C01V60V56G6015_In_FwdFbIO"
          TagType="Base"/>
```

Listing 2. Snippet of XML file containing modified nodes, contains same content as Figure 3

Implementation summary:

- Following our proposed approach, we constructed higher-order transformations to generate pruners;
- To deal with specifics of the problem, we manually customized one of the generated pruners;
- We perform an XML-aware diff and obtain two views, one meant for human inspection and one meant to be parsed by future tooling to automate back-propagation.

VI. Iterative validation

The collaboration on this research project was performed in iterations, allowing for the required agility of adjusting the solution to the real industrial needs. Hence, we have iteratively improved our solution based on our experiences during the development of the approach, its implementation, and testing.

To assess the precision and recall of the pruner with respect to the required changes to be identified, manual checks were performed by the industrial partner. We found that the HTML difference view helped for this purpose, particularly after introducing some small improvements such as limiting the page width to the screen, to avoid horizontal scrolling. Future enhancements of the human readability of the HTML view include reducing the marked differences to the minimum scope, marking e.g. not an entire text node as a difference, but limiting the marking to only the specific characters that are different. Moreover, a mini-map on the page will make it is easier to navigate to highlighted differences.

The point of the iterative and manual validations is that we need to be certain that both the precision (does the pruner only filter out things that should be filtered out) and the recall (does the pruner filter out everything that should be filtered out) must be high for the approach to be eventually useful for automated back-propagation. False positives may not be possible to be propagated back to the model (you might identify changes in the code that do not have a counterpart in the model), but false negatives could be much more harmful since the model is expected to be synchronized with the code

after back-propagation. If the identified changes missed some manual changes, then these are not represented in the model and therefore at risk of being lost. Our validation effort is therefore aimed at iteratively including more pruning elements and fine-tuning the pruner accordingly.

Each iteration of the implementation was followed by a test run on a controlled data sample (i.e. a piece of code with purposefully made changes). We have performed these iterations until a predefined set of elements was pruned correctly (a set of elements to be back-propagated was selected beforehand to ensure a well-scoped proof-of-concept implementation).

In summary, the current version of the pruner removes all required elements of this initial set and misses no changes that had to be identified. The main challenge for the algorithm is in calculating the differences. The XML-aware diff tooling has difficulties in discerning between changes versus addition-deletion of elements. In a general case, no diff algorithm can distinguish these two cases, since they are equivalent when looking at the before-state and the after-state of a file. In our setting, this state-based approach is the best we can do, since there is no record of the operations performed during the manual code changes. For back-propagation to the model, we focus on reflecting the differences in the model by considering atomic code-level changes.

A. Performance evaluation

We tested our implementation on files that were provided by the company and were qualified by them as "small examples". The size of these files is in the order of 200k lines of XML, with file size 8.5MB. When two of these files are used as input, the solution produces an overview of the differences in the order of seconds. Since the approach is only executed sporadically, the run-time is not that important as in cases where models need to be synchronized continuously. The resulting HTML overview file can grow rather large due to it containing both input files and additional markup information. In our tests, the resulting HTML file was 66MB, which modern web browsers can load in a few seconds. Note that the resulting HTML file is not the main artifact to be obtained from the pruning step, it is merely meant as a means for manual inspection of the result. Eventually, the much more condensed XML list of changes will be used as input for the back-propagation (either manually or automatically).

> *Take-away message:*
> Our implementation correctly pruned the provided files in an acceptable time. The exact identification of differences remains a challenge for human inspection, but not for automatic back-propagation.

VII. Discussion

Our approach is guided by our industrial setting. When starting from scratch, one might devise a completely different approach to achieve back-propagation. We found that when working in industrial settings, it is more important to devise an approach that is aimed at improving a currently in place state-of-practice, rather than describing a perfect solution for a theoretical setting.

A. Is our solution generic?

We do not initially aim at generalizing the approach to other industrial settings, even though that might be possible. Instead, we consider how generic it is, in terms of its robustness against changes in the DSML and internal libraries. The robustness provided by our HoTs may be limited. In particular, as elaborated in Section V, the implementation does depend on the DSML definition, particularly because of the added special cases in the pruner. Nevertheless, changes to the DSML such as added or renamed attributes can be reflected with minimal modifications to the current pruner implementation. In contrast, larger changes to the DSML that change its internal structure, e.g. newly introduced top-level nodes (think: instructions, datatypes, etc.), would require changes to the HoTs that generate the pruners.

Another argument to call our approach generic is that we rely on the XML syntax for the programs, but not more. This allows us to run the approach with very little input and generate the pruners using higher-order transformations. Furthermore, since we eventually diff also based on this XML, we can be sure that the result is meaningful as long as both XMLs conform to the same schema, which is ensured by the pruners.

> *Take-away message:*
> Our approach is robust to small changes in the DSML and libraries, due to the usage of higher-order transformations to generate the pruners.

B. Future work

In this paper, we have limited our scope to identifying the differences between the generated and modified code, such that these differences can be manually back-propagated to the model. In future stages of this project, we will work towards automatic back-propagation. This requires interfacing with the existing in-house modeling tool to effectively automate the back-propagation process. In its architecture, it intended to cover only the code generation direction, not the other way around. Given the possibly significant required engineering effort for the modifications, it was important to start with a proof-of-concept implementation of the part of the round-trip presented in this paper. Our current results are being used as input for an investigation into modifying the tooling for allowing back-propagation.

For the back-propagation implementation, the first aspect to consider is the need to group code-level changes into model-level changes. For domain experts, it might be immediately obvious that two nearby changes as belonging to the same model element or not. Until now, we have approached this by considering atomic changes for back-propagation. Specifically, each identified change in the XML would be mapped to a

single change at the model level. The next step in the collaboration is to study the possibilities for a back-propagation mechanism given the identified changes from the approach presented in this paper.

> *Take-away message:*
> After seeing the developed proof-of-concept implementation for the comparison part of the round-trip, the company will move on with the next stage of the project and invest in the engineering effort required to allow automatic back-propagation of the identified changes.

C. Modeling-related lessons learned

HoTs proved to be a very powerful means in this setting to derive the needed transformations for pruning the input files. Nevertheless, to deal with the specifics of the setting, in the end, some customizations were needed for one of the transformations.

Our approach would probably be different in a different setting, but the type of research collaboration brings with it the need to adapt to many practical issues. In particular, we have in this paper considered a DSML and a model in that language as expressed in the in-house tool. Moreover, we have considered the code generator as a model transformation. In practice, the borders between these artifacts are less strict. It is for example not so easy to separately consider the DSML, in-house tool, and code generator since they are all integrated into the same application.

A final lesson learned was related to the effort of devising a partial solution versus the expected effort of a complete solution. Currently, we have implemented a solution that presents the identified changes to the engineers. This is the first step towards eventually establishing automated back-propagation. Since the engineering effort involved in this step is limited as compared to the complete automation of back-propagation, it was a good first step to quickly bring value to the company.

> *Take-away message:*
> Our approach is guided by the existing implementation, in which the DSML and code generator are tightly coupled in an existing in-house tool. Migrating in this setting to allow for bidirectionality required considerable rework.

VIII. Conclusion

In this paper, we reported on our experiences towards establishing round-trip engineering in an industrial setting. The described approach builds on top of existing infrastructure in place at the industrial partner and is, therefore, more about migrating to the new solution rather than inventing a perfect solution on a blank canvas. We have devised a general method, then implemented a specific instance of it and shared some experiences from using it in industry. Findings from applying our method in an industrial setting indicate the usefulness and usability of the approach. Our solution relies on higher-order transformations, which are a powerful modeling concept. We have however also encountered some challenges, in particular related to instantiating the general method in our particular industrial setting, leading to some modeling lessons learned. In particular, it is important to realize that this problem can be described as a modeling problem, but in practice involves much less neatly separated artifacts and manipulations of those artifacts.

Acknowledgment

This work is supported by Software Center.[2]

References

[1] D. C. Schmidt, "https://doi.org/10.1109/MC.2006.58 Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, p. 25, 2006.

[2] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack, "https://doi.org/10.1007/978-3-540-69100-6_1 The epsilon generation language," in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2008, pp. 1–16.

[3] R. F. Paige, N. Matragkas, and L. M. Rose, "https://doi.org/10.1016/j.jss.2015.08.047 Evolving models in model-driven engineering: State-of-the-art and future challenges," *Journal of Systems and Software*, vol. 111, pp. 272–280, 2016.

[4] H. Koziolek, A. Burger, M. Platenius-Mohr, and R. Jetley, "https://doi.org/10.1016/j.jss.2020.110575 A classification framework for automated control code generation in industrial automation," *Journal of Systems and Software*, vol. 166, p. 110575, 2020.

[5] M. Tiegelkamp and K.-H. John, *IEC 61131-3: Programming industrial automation systems*. Springer, 2010.

[6] Z. Diskin, H. Gholizadeh, A. Wider, and K. Czarnecki, "https://doi.org/10.1016/j.jss.2015.06.003 A three-dimensional taxonomy for bidirectional model synchronization," *Journal of Systems and Software*, vol. 111, pp. 298–322, 2016.

[7] U. Nickel, J. Niere, and A. Zündorf, "https://doi.org/10.1145/337180.337620 The fujaba environment," in *Proceedings of the 22nd international conference on Software engineering*, 2000, pp. 742–745.

[8] B. Vogel-Heuser, D. Witsch, and U. Katzke, "https://doi.org/10.1109/ICCA.2005.1528274 Automatic code generation from a UML model to IEC 61131-3 and system configuration tools," in *2005 International Conference on Control and Automation*, vol. 2. IEEE, 2005, pp. 1034–1039.

[9] M. Bork, L. Geiger, C. Schneider, and A. Zündorf, "https://doi.org/10.1007/978-3-540-69100-6_3 Towards roundtrip engineering-a template-based reverse engineering approach," in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2008, pp. 33–47.

[10] S. Sendall and J. Küster, "https://doi.org/10.1.1.94.7515 Taming model round-trip engineering," in *Proceedings of Workshop on Best Practices for Model-Driven Software Development*, vol. 1. Citeseer, 2004.

[11] M. Antkiewicz and K. Czarnecki, "https://doi.org/10.1.1.94.7752 Framework-specific modeling languages with round-trip engineering," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 692–706.

[12] H. Giese and R. Wagner, "https://doi.org/10.1007/s10270-008-0089-9 From model transformation to incremental bidirectional model synchronization," *Software & Systems Modeling*, vol. 8, no. 1, pp. 21–43, 2009.

[13] L. Angyal, L. Lengyel, and H. Charaf, "https://doi.org/10.1109/ECBS.2008.33 A synchronizing technique for syntactic model-code round-trip engineering," in *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2008)*. IEEE, 2008, pp. 463–472.

[14] M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin, "https://doi.org/10.1007/978-3-642-02674-4_3 On the use of higher-order model transformations," in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2009, pp. 18–33.

[2] https://www.software-center.se

[15] F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jézéquel, "https://doi.org/10.1007/978-3-540-75209-7_33 Model-driven engineering for software migration in a large industrial context," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2007, pp. 482–497.

[16] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "https://doi.org/10.1016/j.infsof.2014.04.007 Modisco: a generic and extensible framework for model driven reverse engineering," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 173–174.

[17] F. J. Lucas, F. Molina, and A. Toval, "https://doi.org/10.1016/j.infsof.2009.04.009 A systematic review of uml model consistency management," *Information and Software technology*, vol. 51, no. 12, pp. 1631–1645, 2009.

[18] A. Anjorin, E. Yigitbas, E. Leblebici, A. Schürr, M. Lauder, and M. Witte, "https://doi.org/10.22152/programming-journal.org/2018/2/7 Description languages for consistency management scenarios based on examples from the industry automation domain," *CoRR*, vol. abs/1803.10831, 2018. [Online]. Available: http://arxiv.org/abs/1803.10831

[19] F. Ciccozzi, A. Cicchetti, and M. Sjödin, "https://doi.org/10.1016/j.infsof.2012.07.014 Round-trip support for extra-functional property management in model-driven engineering of embedded systems," *Information and Software Technology*, vol. 55, no. 6, pp. 1085–1100, 2013.

[20] F. Ciccozzi, T. Seceleanu, D. Corcoran, and D. Scholle, "Uml-based development of embedded real-time software on multi-core in practice: lessons learned and future perspectives," *IEEE Access*, vol. 4, pp. 6528–6540, 2016.

[21] D. Durisic, M. Staron, M. Tichy, and J. Hansson, "https://doi.org/10.1007/s10270-017-0601-1 Assessing the impact of meta-model evolution: a measure and its automotive application," *Software & Systems Modeling*, vol. 18, no. 2, pp. 1419–1445, 2019.

[22] A. Anjorin, T. Buchmann, B. Westfechtel, Z. Diskin, H.-S. Ko, R. Eramo, G. Hinkel, L. Samimi-Dehkordi, and A. Zündorf, "https://doi.org/10.1007/s10270-019-00752-x Benchmarking bidirectional transformations: theory, implementation, application, and assessment," *Software and Systems Modeling*, pp. 1–45, 2019.

[23] "XSL Transformations (XSLT) Version 3.0," June, 8 2017. [Online]. Available: https://www.w3.org/TR/2017/REC-xslt-30-20170608/