2018 25th International Conference on Mechatronics and Machine Vision in Practice (M2VIP)

Template-Based Generation of PLC Software from Plant Models Using Graph Representation

Yurii Pavlovskyi
Fraunhofer IFF
Magdeburg, Germany
yurii.pavlovskyi@iff.fraunhofer.de

Matthias Kennel
Fraunhofer IFF
Magdeburg, Germany
matthias.kennel@iff.fraunhofer.de

Ulrich Schmucker
Fraunhofer IFF
Magdeburg, Germany
ulrich.schmucker@iff.fraunhofer.de

***Abstract*—Digital planning of manufacturing processes becomes standard industrial practice. It implies the creation of detailed digital plant models, providing opportunities for the automated transition from the digital model of a system to a software implementation. This enhances the development efficiency and software quality, helps enforce programming standards, and facilitates reuse of the information from the design phase. The paper introduces a novel technique of software generation using declarative metaprogramming in the template-based approach that interprets both the model and the software as graph structures. It is applied to generate software in the graphical and textual languages of IEC 61131-3 from a plant model and templates that are developed in the native development environment of a programmable logic controller.**

***Index Terms*—Programmable logic controllers, code generation, metaprogramming, graphs, IEC 61131-3**

I. INTRODUCTION

Digital planning of manufacturing processes becomes standard industrial practice. The motivation of this paper is to smooth the transition from the digital planning to a working plant by automatically translating the digital model into a functioning software.

Automatic code generation helps to enforce design patterns and company standards. It also facilitates reuse of the data collected in the design phase directly in the implementation phase [1].

A practical method for the automated generation of the programmable logic controller (PLC) software directly from the design data reduces manual work and, consequently, reduces the number of defects, minimizes required effort, and increases code quality.

This work proposes a novel original technique based on principles of the declarative metaprogramming [2] and template-based code generation [3] for automatic generation of PLC software reflecting the model structure in all languages of IEC 61131-3 [4] from templates in native integrated development environment (IDE) without using additional visual editing software. To the best of knowledge, no existing methods are able to provide such possibility. The technique is implemented in commercially available software and evaluated on various projects.

This work was supported by the Ministry of Economy, Science and Digitalisation of Sachsen-Anhalt and European Regional Development Fund.

VINCENT[1] developed by Fraunhofer IFF, allows reusing the data generated in the planning phase of an automation project during complete engineering cycle, including virtual commissioning and operation of a digital twin. VINCENT supports the development of kinematic 3D models for mechatronic plants and a description of the plant behavior using a flowchart utilizing abstract kinematic notions for axes and sensors. Without specifying engineering or programming details, a functioning model of the machine can be created through teach-in (the motion sequence is recorded by example) and verified on the 3D model. Motion sequences can be specified directly in the model using axis movements, instead of implementing them in the software with actor commands and combined with logic to produce complex behavior description.

VINCENT provides abstract-level plant description in the machine-readable eXtensible Markup Language (XML) [5] format. TIA Portal is used as a code generation target platform. The Openness [6] API supports import and export of the SIMATIC STEP7 PLC software projects using an own XML format or source code.

Section II considers related work; section III describes the applied methods and techniques; section IV describes the evaluation of the proposed technique; section V presents the discussion and future work.

II. RELATED WORK

Code generation, in general, is well-established as a programming tool for accelerating mundane tasks. It is applied to create a design of desktop application graphically and automatically translate it to code (for example, NetBeans [7] and WindowBuilder [8]). The developer is then able to concentrate on the logic of the application. Other tools automatically generate code from XML descriptions of data structures and services (e.g, Apache Axis2 [9]). Parser generators [10] produce code from declarative language definitions. These are only a few of the notable applications.

An *overview* of the software engineering applications in the industrial automation domain is available in [11] and [3]. The work [3] identified three approaches to the code configuration of the PLC software: the modular approach, the parameter-based approach, and the template-based approach. The latter being most flexible of them.

[1]https://www.vincent.engineering/

978-1-5386-7544-1/18/$31.00 ©2018 IEEE

The existing *template-based* approaches include [12], [13], [14] and [15]. In [16] the authors propose a *parameter-based* software configuration method. In [17] the authors present *modular* approach. In [1] the authors present a *case study* describing application of a commercial tool that is capable of template-based and parameter-based code generation.

Software synthesis was demonstrated for example in [18], [19], and [20]. The synthesis problem requires generating software satisfying the high-level requirements. The synthesis might still be used to generate plant model in combination with the proposed technique for PLC code generation, but it is a formidable challenge by itself and is therefore out of the scope of this work.

Further, we will present a short comparison of our technique to a selection of existing methods.

In [12] the authors propose an approach to automatic code generation using XSLT technology. Profound knowledge of this technology would be required to achieve similar results as in our case.

The methods proposed in [12], [16], and [15] use code generation templates prepared in the external text editor. In this work, the PLC project designed in native IDE serves as a template. This ensures that all usual syntactic checks and productivity tools are available during the template development, and that template definition for visual languages is easily possible.

The techniques of the model-driven engineering, including [13], [14], consider multi-level modeling down to the program organization units. This allows automatic generation of tests, requirements tracing, and validation. The current technique, however, does not require to include the complete implementation details in the model, sacrificing some guarantees to achieve flexibility and simplicity.

Similar to [1], the proposed technique exploits the hierarchical data structure for model and software. A notable difference is that it creates cross-references between program elements automatically, whereas in [1] the connections are specified in special visual editing software.

The recent works [15] and [17] have some parallels to our technique. The work [15] proposes using embedded language in templates and the dependency tree for model specification. It allows including logic in the templates, which is only possible to some degree with our technique. However, application for the graphical languages as well as the possibility of template development in native IDE was not considered. The work [17] also utilizes TIA Portal Openness interface, modular machine and software definition using graph representation, but does not allow for finer granularity in the contents of the modules allowed by template-based approaches.

Several methods listed above also consider automatic hardware configuration, which will not be considered in this work. There are also some commercial and open-source tools available for code and hardware configuration (for instance [21], [15]).

Existing methods are often overlooked [3], partly, because companies have prescribed hardware and software standards (e.g., in German automotive industry often the use of the graphical languages is required) which imposes significant flexibility requirement on the code generation method. Summing up, the existing methods either require knowledge of the high-level programming languages to define the code generation logic in a template, support only text-based languages, or otherwise are not fully automatic due to inherent code generation architecture. The motivation for this work is addressing these challenges.

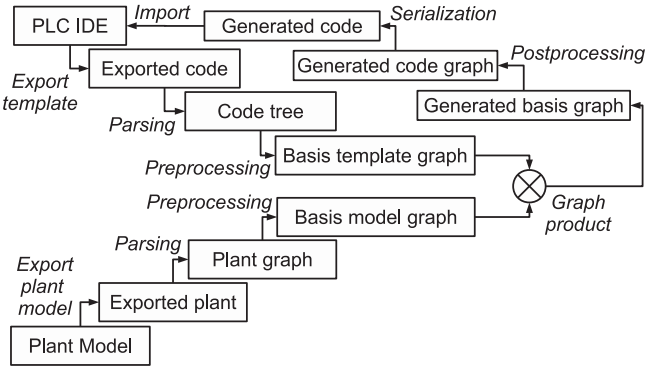

Fig. 1. Technique overview.

III. METHODS

A. Overview

The goals of the proposed technique are: to produce the program written in IEC 61131-3 according to an arbitrary software architecture defined in the code template developed in native PLC IDE and reflecting the model structure; and to generate the control logic consistent with the software architecture from the behavior model. The very definition of the software architecture suggests that a graph is an appropriate way of its description [22], therefore it is used for both the model and the template. Fig. 1 shows an overview of the technique.

In the *first step*, the template and plant model are accessed via a programmatic or file-based interface. In the *second step*, both the template and the plant model are parsed into the graph data structures.

We propose the notion and structures for *basis graphs*, which possess certain properties described in section III-C and are used as intermediate representation. The graph transformations from parsed graphs to basis graphs are made in the *third step* to prepare the data structures for the code generation.

Then the proposed *graph product transformation* (GPT) as described in section III-D is applied. The proposed name relates to the similarity of the algorithm output to the product of two graphs [23]: the template graph and the model graph.

The produced basis graph is then transformed back into the original graph structure, the code is generated from the produced graph structure, and the generated code is imported back into the IDE.

B. Preliminaries

A graph is a tuple $G = (V_G, E_G)$ [23] with a set of vertices or nodes $V_G = \{v_1 \dots v_n\}$ and a set of edges $E_G = \{e_1 \dots e_m\} \subseteq (V_G \times V_G)$. Further, a node v_j of a directed graph is called a direct successor of a node v_i, and v_i is called a direct predecessor of a node v_j, $i, j \in 1 \dots n$ if and only if $(v_i, v_j) \in E$. The vertices reachable from a node v_i are called its successors. A tree is a connected acyclic graph. A spanning tree $G^{(t)} = (V, E^{(t)})$ of a graph $G = (V, E)$, $E^{(t)} \subseteq E$, is a subgraph with the same vertex set as G that is a tree. A connected graph may have multiple spanning trees.

In further, the typed attribute graph is defined informally, inspired by the [24]. A typed graph is defined by a tuple $TDG = (TG, G, d)$, where $TG = (D, ETG)$ is a type graph with type nodes D and type edges ETG, and $G = (V_G, E_G)$ is an instance graph. The typed graph has following properties: a) for all vertices V_G there is a mapping $d : V_G \to D$ from the instance graph nodes to the type graph nodes; b) if there is an edge $e = (v_i, v_j) \in E_G$, then there is a corresponding edge in the type graph: $(d(v_i), d(v_j)) \in ETG$.

An attribute graph is a tuple $AG = (G, attr)$, where G is a graph, S is the set of all strings of letters and $attr : V_G \times S \to S$ is a labeled mapping from graph nodes to attribute values.

A data graph is defined here as a typed attribute graph with a root node:

$$G = (V, E, v_0, D, ETG, d, A, attr, id),$$

where the nodes V are connected by the edge relationships $E \subseteq V \times V$ and $v_0 \in V$ is a root node. A node possesses a class, defined by the mapping $d : V \to D$ that determines item type – for instance, axis, sensor, etc. Other properties of the typed graph also hold: $(d(v_i), d(v_j)) \in ETG \subseteq D \times D$. The function $attr : V \times A \to S$ defines attribute values of a node for each attribute $a_i \in A \subseteq D \times S$ defined in the type graph. A function $id : (D \to S) \subseteq A$ provides a name of an attribute that has semantic meaning of a node identifier – e.g., $attr(v_i, id(d(v_i)))$ may be a variable identifier, name, or title of a node v_i.

For example, a machine axis a with the name "M1" may be represented by the node with functions of the data graph defined as $d(a) := Axis$, $id(Axis) := "name"$ and $attr(a, "name") := "M1"$.

C. Basis Graph

We define a basis graph as a data graph with the following properties:

a The direct successor nodes of a node can be described by "belongs to," "referenced by," or "is part of" semantic relationship.

b The graph transformation from a parsed graph to a basis graph is deterministic and semantically reversible – that is it produces same basis graphs for two parsed graphs if and only if they have the same semantics.

c The nodes that correspond to multiple nodes in the generated graph have string identifier attributes derived from the parsed graph.

Additionally, for the template basis graph:

d A spanning tree is specified.

e All semantic dependencies of the nodes that correspond to multiple nodes in the generated graph to other nodes are represented explicitly by the edges or implicitly by the equality of an attribute of the referencing node and the identifier of the referenced node.

Regarding the property "d", a graph with its spanning tree effectively define a bare bigraph [25] which was applied previously for modeling of the spatially distributed systems. This separation was also performed in the graph package system framework [26].

Regarding the property "e", the template basis graph nodes are not allowed, for instance, to have numeric identifiers which reference other nodes by the identity of attributes.

D. Algorithm

1) Input Data: The GPT algorithm takes a template graph and a model graph and produces a target graph. The template graph G_t with the nodes V_t is a basis data graph that serves as a template for the transformation. The model graph G_s with the nodes V_s is a basis data graph representation of the model that is mapped onto the architecture specified by the template graph during transformation. In following, the subscripts t, s, and r always refer to the template, model, and target graph respectively.

The proposed algorithm performs recursion on the template graph. The template graph is not a tree in the general case. Therefore, it is required that a spanning tree for the template graph is specified by providing its edges $E_t^{(t)} \subseteq E_t$. The edges included in the selected tree must have the semantic meaning "contains" and graph edges that are not included – "references."

The nodes of the model graph may have tags. The tags are employed to generate different code for the nodes of the same type. For instance, the specific PLC code to control an axis depends on the chosen hardware. The function $tags : V_s \to \mathcal{P}(T)$ returns the set of node tags, where $T \subseteq S$, and $\mathcal{P}(X) = \{x \mid x \subseteq X\}$ denotes a power set.

Typically, in the programming language, the semantics is specified by means of the string identifiers or symbols. The program components with the same identifiers inside the same scope are deemed to be referring to the same object.

The GPT is based on the special strings of letters, called markers, which are syntactically allowed as substrings in the identifiers of the target language. The metalanguage syntax is defined by a mapping from the template graph attribute value to the marker list: $markers : S \to \mathcal{P}(M)$, where M is the set of all markers, which finds the marker substrings. A marker refers to a specific model graph node by its type defined by a mapping $d_m : M \to D_s$.

Important part of the algorithm is the notion of context. A context is a subset of nodes from the model graph $c \subseteq V_s$. The context contains maximum one node of a given type

$$\nexists v_{si}, v_{sj} \in c : d_s(v_{si}) = d_s(v_{sj}).$$

The function $rewrite(s, c) : S \times \mathcal{P}(V_s) \rightarrow S$ replaces the markers in the string of letters s with the current context-based values from the context c. It has following properties: a) $\forall\, c,\ s : markers(rewrite(s, c)) = \emptyset$ – the resulting string produced by $rewrite$ function does not contain any markers; b) two rewritten result strings produced by $rewrite(s, c_1)$ and $rewrite(s, c_2)$ should be equal even when $c_1 \neq c_2$ if all objects of the types referenced by markers are same in both contexts. This ensures that the semantic relationship is established in the target graph.

2) Markers: The markers are classified into the structural, attribute, assignment, value, and definition markers, which form a partition of the set M.

The structural markers $M_s \subseteq M$ are mainly used to define relationships between the template graph structure and the model graph structure. The structural marker selects a subset of model graph nodes depending on the context using some predicate. A structural marker $m_s \in M_s$ also has a tag $t(m_s) \in T$. The marker is replaced with the identifier attribute value of the referenced node by the $rewrite$ function.

The attribute, value, and assignment markers only affect the generated node and do not change the context. The attribute markers are rewritten with a value of a predefined attribute of the model node from the current context with a predefined type. The assignment markers set the predefined generated node attribute(s) to the value of the predefined model node attribute(s) from the current context - e.g., the axis velocity assignment marker sets the initial value of the variable to the axis velocity limit in the generated code. The assignment markers are rewritten with an empty string. The value markers replace the subtree of the template spanning tree consisting of the node and its descendants with another subgraph.

The definition markers allow specifying the semantic references in the subgraphs generated by the value markers. They create entries in the symbol table maintained by the transformation algorithm, which are then used by the value markers. The definition markers are rewritten with an empty string.

3) GPT: The recursive algorithm, shown in Fig. 2, traverses the template tree top-down depth-first. The algorithm consists of the three stages, which are delimited by the comments inside curly brackets in Fig. 2. The first stage selects model graph nodes, the second stage iterates the selection results producing subtrees in the target graph and the last stage reconstructs subgraphs from the subtrees.

If the template graph does not contain markers it is simply copied to the target graph. The structural markers change a number of the produced copies of the template subtree and their contexts, and the markers in attributes of the copied nodes are replaced using these contexts.

a) Context expansion: The algorithm is called with a context c and a template graph node v_t. The first stage processes only structural markers found in the identifier of v_t and iterates them in the order they were found in the string, maintaining the current context list C after expansion.

Input: Data graphs G_t, G_s, template node $v_t \in V_t$, context $c \subseteq V_s$, the spanning tree edges $E_t^{(t)}$
Output: The subgraph of target graph G_r produced from subtree of v_t, target node to template node mapping $g \subseteq V_t \times V_r$, target node to context mapping $c_r : V_r \rightarrow \mathcal{P}(V_s)$, unresolved upward edges $up \subseteq E_s$, nodes V_{tt} of a subtree
$C \leftarrow \{c\}, g \leftarrow \{\}, c_r \leftarrow \{\}, up \leftarrow \{\}$, $G_r \leftarrow$ empty graph, $V_{tt} \leftarrow \{\}$
{Context expansion}
for $m_i \in marker(attr(v_t, id(d_t(v_t))))$ **do**
  if $m_i \in M_s \wedge \nexists v \in c :\ d_s(v) = d_m(m_i)$ **then**
    $C' \leftarrow \{\}$
    for $c_i \in C$, $v' \in V_s$: $P_m(m_i, c, v')$ **do**
      $C' \leftarrow C' \cup \{c_i \cup \{v'\}\}$
    end for
    $C \leftarrow C'$
  end if
end for
{Node rewriting}
for $c_i \in C$ **do**
  if Node has value marker **then**
    $v_i' \leftarrow$ rewrite value marker and get root node
    Add v_i' to V_r
  else
    $v_i' \leftarrow$ new node
    $attr_r(v_i', s) \leftarrow rewrite(attr_t(v_t, s), c_i) \forall s \in S : (d_t(v_t), s) \in A_t$
    Apply assignment markers
    Add v_i' to V_r
    for $v_d \in V_t : (v_t, v_d) \in E_t^{(t)}$ **do**
      $(G_r', g', c_r', up', V_{tt}') \leftarrow$ recursion with v_d and c_i
      Add $\{(v_i', v_d') \mid (v, v_d') \in g' \wedge v = v_d\}$ to E_r
      $G_r \leftarrow G_r \cup G_r'$, $g \leftarrow g \cup g'$, $V_{tt} \leftarrow V_{tt} \cup V_{tt}'$
      $c_r \leftarrow c_r \cup c_r'$, $up \leftarrow up \cup up'$
    end for
  end if
  Add (v_i', c_i) to c_r
end for
$up \leftarrow up \cup \{(v, v_i) \in E_t \setminus E_t^{(t)} \mid v = v_t\}$
$g \leftarrow g \cup \{v_t\} \times \{v_1', .., v_i'\}$, $V_{tt} \leftarrow V_{tt} \cup \{v_t\}$
{Edge resolution}
for $e = (v_a, v_b) \in up : v_b \in V_{tt}$ **do**
  for $((v, v_a') \in g : v = v_a), ((v, v_b') \in g : v = v_b)$ **do**
    if $\forall (v_{c1}, v_{c2}) \in c_{int}(c_r(v_a'), c_r(v_b')) : v_{c1} = v_{c2}$ **then**
      Add (v_a', v_b') to E_r
    end if
  end for
  $up \leftarrow up \setminus \{e\}$
end for

Fig. 2. Graph product transformation algorithm.

For each structural marker m_i and every context $c_i \in C$ the operation of the structural marker depends on the c_i. If the node of the type filtered by the marker is not already included in the context $\nexists v \in c_i : d_s(v) = d_m(m_i)$, then the common successors of all nodes in the c_i (preferring direct successors) are filtered using the predicate $P_m(m_i, c, v') = [\bigwedge_{v_s \in c}(((v_s, v') \in E_s \wedge (d_s(v_s), d_s(v')) \in ETG_s) \vee (v' \text{ is reachable from } v_s \wedge (d_s(v_s), d_s(v')) \notin ETG_s))] \wedge d_s(v') = d_m(m_i) \wedge t(m_i) \in tags(v')$ and for each filtered node a new context is derived by including the node in the new context. The *negated predicate* may be used for some markers $P_m(m_i, c, v') = [\bigwedge_{v_s \in c}(((v_s, v') \notin E_s \wedge (d_s(v_s), d_s(v')) \in ETG_s) \vee (v' \text{ is reachable from } v_s \wedge (d_s(v_s), d_s(v')) \notin ETG_s))] \wedge d_s(v') = d_m(m_i) \wedge t(m_i) \in tags(v')$.

b) Node rewriting: The second stage creates a new node v'_i in the target graph for each context c_i in C based on the node v_t by rewriting its attributes using the context c_i. The value, assignment and definition markers are applied at this point. Then, for each created node the algorithm is called recursively for every descendant node v_d in the template graph spanning tree and the result is merged into the current state. The edges from v'_i to the nodes created during the recursive call based on the node v_d are appended into the target graph. The context c_i is saved as the one used to produce v'_i.

The edges that are not a part of the spanning tree $E_t \setminus E_t^{(t)}$ are called upward edges henceforth because they are built by traversing the tree bottom-up. The upward edges outgoing from the node v_t and all nodes that were created from the node v_t are saved in the current state.

c) Edge resolution: The resolution of upward edges is performed after the complete subtree of v_t was built in the target graph. At this point, the up contains all the upward edges that could not be reconstructed in the descendant subtrees. The upward edge $e = (v_a, v_b) \in up$ is reconstructed in the target graph as soon as the node v_b was processed in the subtree (the node v_a always is) at the closest common ancestor node of v_a and v_b in the template spanning tree.

Initially, the edges outgoing from all nodes created from v_a to all nodes created from v_b are considered. They are filtered semantically, analogous to how the identifier rewriting works in the algorithm. This is done by computing the common set of two contexts c_1 and c_2, which is defined as the set of pairs nodes of the same type from both contexts, $c_{int}(c_1, c_2) := \{(v_{c1}, v_{c2}) \in c_1 \times c_2 \mid d_s(v_{c1}) = d_s(v_{c2})\}$. Only when $\forall (v_{c1}, v_{c2}) \in c_{int}(c_1, c_2) : v_{c1} = v_{c2}$ for the contexts that were used to generate the two nodes, the upward edge is created in the target graph.

E. Example

Fig. 3 shows a highly simplified, but motivating example. The goal is to generate the code for every machine axis and every operation and set the variable `Var_Move` of the axis referenced by the operation to `TRUE` (e.g., to start an operation or activate a frequency inverter of a motor). The tags have intuitive meaning in this case, they may define motion

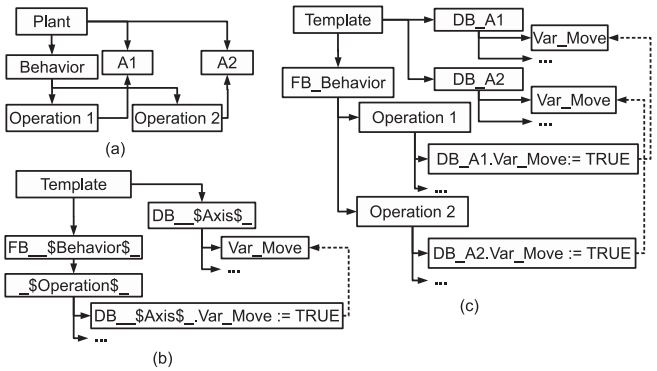

Fig. 3. GPT example. (a) Model graph. (b) Template graph. (c) Target graph.

system vendor, intended axis use (e.g., constant velocity or absolute positioning), and other parameters to choose different implementation.

The plant graph has two axes `A1`, `A2`, and two operations `Operation 1`, `Operation 2`. The `_$x$_` is a structural marker with the type $d_m(\texttt{\_\$x\$\_}) = x$. The marker set is $M = M_s = \{\texttt{\_\$Axis\$\_}, \texttt{\_\$Behavior\$\_}, \texttt{\_\$Operation\$\_}\}$ and function $markers$ returns the found marker substrings. The function $rewrite$ replaces the markers with the corresponding model node identifiers, which are shown as node labels in Fig. 3. The dashed lines indicate upward edges. The template graph nodes may have descendant subtrees without upward edges (denoted by "..."), which do not affect the result.

1) The algorithm is called for the root node `Template` with an empty context $c_{Template} = \{\}$ and copies it without changes as it does not contain any markers.
2) The algorithm is called recursively for the descendant node in the template tree `FB__$Behavior$_` from step 1). The context expansion produces context $c_{Behavior} = \{\texttt{Behavior}\}$. The identifier attribute is rewritten $rewrite(\texttt{FB\_\_\$Behavior\$\_}, \{\texttt{Behavior}\}) = \texttt{FB\_Behavior}$.
3) The algorithm is called for the node `_$Operation$_` from 2) and the context expansion produces two contexts for each operation. The node rewriting stage produces nodes `Operation 1` and `Operation 2`.
4) The algorithm is called from 3) for `DB__$Axis$_.Var_Move := TRUE` with the context $\{\texttt{Behavior}, \texttt{Operation 1}\}$ that does not contain an axis. The only axis node reachable from `Operation 1` is `A1`, which is added to the single context after context expansion. The template graph contains upward edge outgoing from the node. The up set is extended with the edge (`DB__$Axis$_.Var_Move := TRUE`, `Var_Move`), and the context of the node `DB_A1.Var_Move := TRUE` is saved as $\{\texttt{Behavior}, \texttt{Operation 1}, \texttt{A1}\}$
5) Similarly the node `DB_A2.Var_Move := TRUE` is

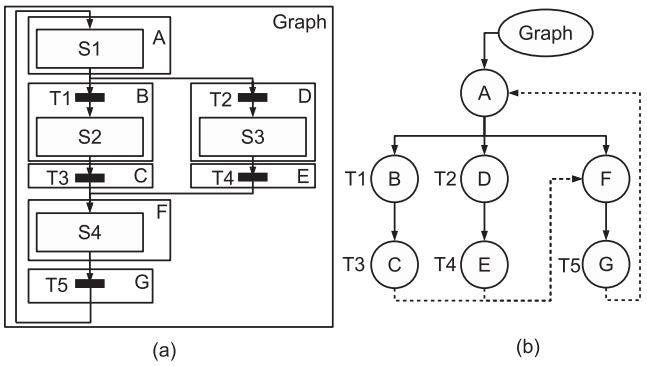

Fig. 4. SFC transformation. a) SFC. b) Corresponding basis graph.

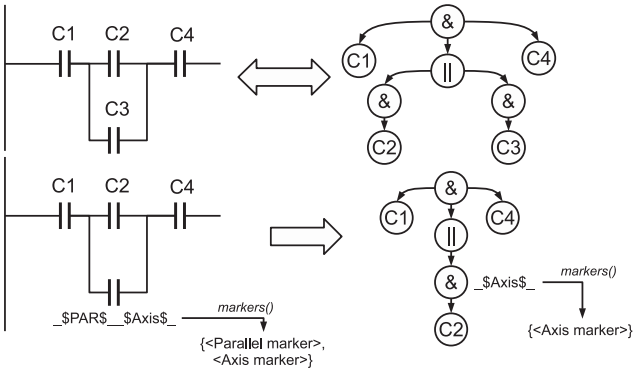

Fig. 5. LD basis graph transformation.

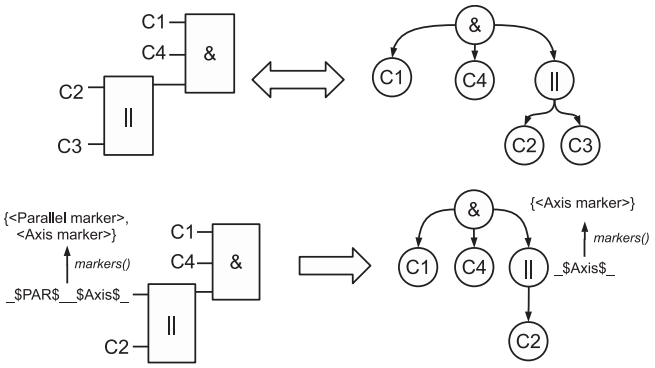

Fig. 6. FBD basis graph transformation.

created.

6) The algorithm is called from step 1) for the node `DB__$Axis$_` with the same context. The context does not contain an axis node. The filtered nodes of the model graph are `A1` and `A2`. The contexts produced during context expansion are $c_{DB\_A1} = \{\texttt{A1}\}$ and $c_{DB\_A2} = \{\texttt{A2}\}$, and for each context a subtree is created in the target graph.
7) In the edge resolution phase of the `Template` node, the upward edge is resolved with an edge between `DB_A1.Var_Move := TRUE` and `Var_Move` node of `DB_A1` with the common set $\{(\texttt{A1}, \texttt{A1})\}$. Similarly, the second edge is created.

This example also shows that the upward edges resemble the resolution of the variables in the target programming language.

F. Template Basis Graphs

An IEC 61131-3 program [4] consists of the program organization units (POUs): functions (FC), function blocks (FB), and programs (PRG). The blocks may be written in the two graphical languages: function block diagram (FBD), ladder diagram (LD) or two textual languages: instruction list (IL), structured text (ST). The FBs and PRGs can be structured as sequential function charts (SFC). We further propose and illustrate informally the structures of the template basis graphs with properties defined in III-C.

1) SFC: Nodes of the basis graph (Fig. 4) contain at most one transition followed by at most one step, but at least one of both. The transition names are used as the node identifiers. If there is no transition in a node, then it has an empty identifier attribute value and, therefore, does not contain markers.

The basis graph is built starting from the top (or initial) step, the first child of the root node, downwards for each transition and step pair. When an end of a branch is encountered, the following elements start a new basis graph branch from the common ancestor node of all nodes immediately preceding the branch end. The processing ends at the transitions to already processed steps. The links connecting the steps and transitions that are not a part of the built tree are added in the graph as upward edges. The step-step and transition-transition links (e.g., "A - F") are not generated in reverse transformation. The parallel and alternative branches can be unambiguously represented by the proposed basis graph.

2) LD: LD is transformed into an abstract syntax tree structure consisting of the disjunction (||) and conjunction (&) operations (Fig. 5). Variable names in contacts are used as the identifiers. A contact is then repeated if the context expansion produced multiple contexts, implementing conjunction operation.

The disjunction is possible using an additional marker available in the LD – the *parallel marker*. Parallel markers are processed during the transformation to the template basis graph. A single contact branch with the contact identifier containing the parallel marker is deleted and its identifier is assigned to all conjunction nodes on the same level.

3) FBD: FBD is transformed (Fig. 6) analogous to LD. The labeled nodes are part inputs and repeated by GPT. The parallel marker input is deleted and its label is assigned to the part. This allows repeating the part.

4) Textual Languages: ST and IL code may be transformed into the tree structure by including markers within comments (Fig. 7).

G. Value Markers

1) SFC Behavior: The behavior is generated from a plant model flowchart by using a special value marker in the tran-

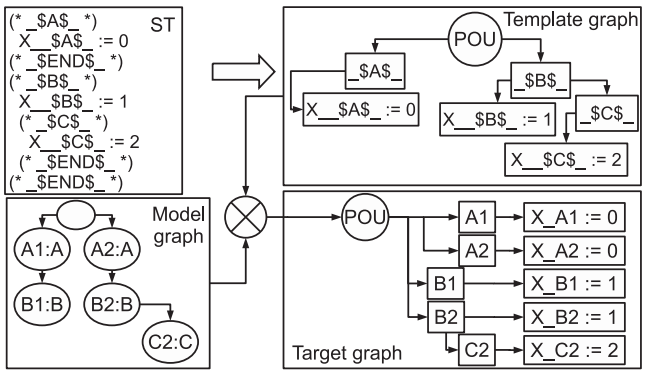

Fig. 7. Textual basis graph transformation.

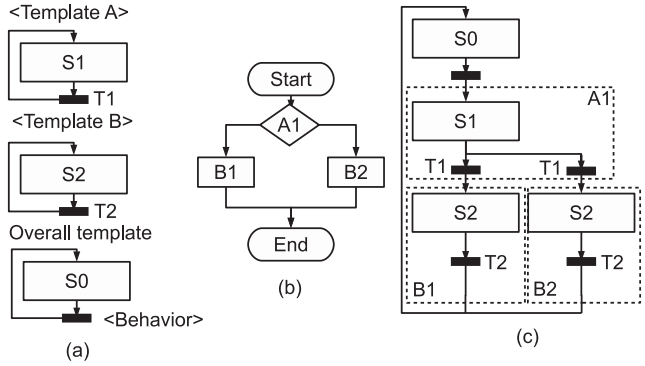

Fig. 8. SFC behavior generation. (a) Graph template. (b) Behavior model. (c) Result.

sition name. The template includes specially named networks that define templates for elementary flowchart operations, such as moving the axis or making a decision. The operation template networks start with a step and end with a jump transition back to the starting step. This jump transition is used to glue the operation templates together.

The flowchart is transformed by replacing its nodes with the graphs for elementary operations that are transformed by the GPT starting with the context that includes the flowchart operation node. The generated graph is then glued into the target graph replacing the original node.

Fig. 8 shows an example. The code template (Fig. 8a) defines two operation templates and the overall template. The behavior flowchart (Fig. 8b) makes use of these operations. Fig. 8c shows the produced graph.

2) LD and FBD Expressions: The node and its descendants are replaced by an expression tree that is obtained from an abstract syntax tree (AST) of a plant model formula (e.g., a condition in the decision block of the flowchart). As it can be intuitively seen from Fig. 5 and Fig. 6, the branch node of the LD and FBD tree is a part or a component performing some operation and therefore is a concrete implementation of an AST operation node. The leaves are the variable identifiers of the operands and are obtained from the AST by replacing the model variables with references to the nodes marked by definition markers.

3) ST Expressions: Expressions are generated using the context-free language translation [27]. There are three textual languages: IL, ST, and SFC actions.

IV. RESULTS

The code generation tool for TIA Portal V14SP1 / V15 PLC IDE was implemented and integrated as a part of the commercially available VINCENT software, including the basis graph transformations for S7-Graph (STEP7 SFC) and LD, the GPT algorithm, and the markers.

The code generator takes a model and a syntactically correct TIA Portal template project. Most attributes, including names, comments, and titles in the template project may contain markers. The markers have the form `_$<Marker>$_`. There were 37 marker types defined. For example, structural: `_$Axis_Motor$_` – finds all machine axes with the tag "Motor," value: `_$Expression$_` – is replaced by the current decision condition, assignment: `_$EQ_AxisPos$_` – assigns the axis position value as the initial value of the variable. The PLC project is accessed via TIA Portal Openness using an additionally developed tool implemented in C#, which exports and imports XML files for the blocks in FBD, LD, and S7-Graph (the STEP7 SFC), and source code for AWL (the STEP7 IL) and SCL (the STEP7 ST language).

The plant model was represented by a graph starting with the root plant node, which represents a graph of axes, sensors, control sequences, etc. An axis, for instance, has a list of positions, which are used as motion targets in the behavior description. A sequence has sequence variables and a flowchart. The blocks of a flowchart may contain machine motion definitions, which in turn, also reference axes, axes positions, and sensors. The nodes (such as axes and sensors) have names which serve as unique identifiers.

The template graph has a root project group node, containing other groups, blocks, variable tables, and user-defined types, which themselves are root nodes of the graphs obtained from the corresponding exported files.

The technique was successfully evaluated on projects with varying complexity and structure. The machine that sorts parcels between two conveyors inspired by [28] was modeled in VINCENT. The template project in LD and S7-Graph was created and the code generation tool was applied (execution time 300ms, CPU Intel Xeon E5-1630 v3, 3.70 GHz, generated 23 POUs, 82 contacts in LD, 43 steps, and 44 transitions in S7-Graph).

The hardware configuration, consisting of a technology object drive, 2 cylinders with end position sensors, and a light barrier sensor, was created manually with naming, that is consistent with the plant model and template. The generated software was imported into the project (import time 12.7s) and could be compiled without change. The resulting project was tested using virtual commissioning and was fully functional. This case was also evaluated experimentally from the usability standpoint by three subjects with PLC development background without previous knowledge of the proposed technique.

They all were able to complete this project from scratch in under 3 hours.

A more complex case is another machine with 19 axes: code generation produced 53 POUs, 1006 contacts, 557 steps, 663 transitions, code generation took 1144ms, import in TIA Portal 43.1s.

The obtained transformation is deterministic. When the process is repeated, the generated code is compared to the previous version and only the changed code blocks are required to be updated. If the appropriate modular architecture and design patterns (loose coupling, design for reusable components, etc.) are used to avoid the manual editing of the generated code, it is possible to simplify the iterative development process and improve maintainability.

V. DISCUSSION

The current technique does not explicitly include any mechanisms to verify the semantics and functionality of the generated program. But at the syntax level, sanity checks are done during transformation, strict XML validation is executed by TIA Portal Openness before import, and afterward, the absence of syntactic errors is ensured by IDE build process.

Future work includes a more thorough practical evaluation from maintainability and usability view; complete implementation of the technique for FBD, ST, and IL; research of other marker and upward edge resolution policies; extension of the formal approach; and research towards improving the maintainability aspect. We also plan to extend the technique to a more general solution supporting more PLC vendors. Significant effort is devoted to the standardization of the XML representation of the IEC 61131-3 code [29]. The IEC 61131-10 standard based on the PLCOpen XML format for languages defined in the IEC 61131-3 is expected to appear in the near future.

ACKNOWLEDGMENTS

This work has been supported by the project “Investigation of new production and control concepts for the automated pre-assembly of large components (autoVMon)” (Grant No. 1604/00084) of Ministry of Economy, Science and Digitalisation of Sachsen-Anhalt and European Regional Development Fund.

We would also like to thank the anonymous reviewers for the much-appreciated comments and suggestions.

REFERENCES

[1] P. Falkman, E. Helander, and M. Andersson, “Automatic generation: A way of ensuring PLC and HMI standards,” in *ETFA2011*. IEEE, Sep. 2011, pp. 1–4.

[2] K. Mens, I. Michiels, and R. Wuyts, “Supporting software development through declaratively codified programming patterns,” *Expert Systems with Applications*, vol. 23, no. 4, pp. 405–413, Nov. 2002.

[3] B. Vogel-Heuser, J. Fischer, S. Feldmann, S. Ulewicz, and S. Rösch, “Modularity and architecture of PLC-based software for automated production systems: An analysis in industrial companies,” *Journal of Systems and Software*, vol. 131, pp. 35–62, Sep. 2017.

[4] IEC, “IEC 61131-3 Standard - Programmable controllers - Part 3: Programming languages,” Jan. 2003.

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, “Extensible markup language (XML) 1.0 (fifth edition),” W3C, Tech. Rep., Nov. 2008, https://www.w3.org/TR/2008/REC-xml-20081126/.

[6] Siemens AG, “TIA Portal,” http://www.industry.siemens.com/topics/global/en/tia-portal/, 2018, accessed: 2018-10-10.

[7] Apache Software Foundation, “Apache NetBeans,” https://netbeans.apache.org/, accessed: 2018-10-09.

[8] Eclipse Foundation, “WindowBuilder,” https://www.eclipse.org/windowbuilder/, accessed: 2018-10-09.

[9] Apache Software Foundation, “Axis2,” https://axis.apache.org/, accessed: 2018-10-09.

[10] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.

[11] V. Vyatkin, “Software engineering in industrial automation: State-of-the-art review,” *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1234–1249, Aug. 2013.

[12] E. Estévez, M. Marcos, and D. Orive, “Automatic generation of PLC automation projects from component-based models,” *The International Journal of Advanced Manufacturing Technology*, vol. 35, no. 5-6, pp. 527–540, Jul. 2007.

[13] M. Jamro, “Automatic generation of implementation in SysML-based model-driven development for IEC 61131-3 control software,” in *2014 19th International Conference on Methods and Models in Automation and Robotics (MMAR)*. IEEE, Sep. 2014, pp. 468–473.

[14] E. Estevez, F. Perez, D. Orive, and M. Marcos, “A novel approach for flexible automation production systems,” in *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*. IEEE, jul 2017.

[15] G. Ulm, F. Bellorini, D. Brodrick, R. Fernandes, N. Levchenko, and D. Piso, “PLC Factory: Automating routine tasks in large-scale PLC software development,” *Proceedings of the 16th Int. Conf. on Accelerator and Large Experimental Control Systems, ICALEPCS2017, Barcelona, Spain*, 2018.

[16] J. Fischer, B. Vogel-Heuser, and D. Friedrich, “Configuration of PLC software for automated warehouses based on reusable components- an industrial case study,” in *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE, Sep. 2015, pp. 1–7.

[17] A. Armentia, E. Estevez, D. Orive, and M. Marcos, “A tool suite for automatic generation of modular machine automation projects,” in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. IEEE, jul 2018.

[18] C.-H. Cheng, E. A. Lee, and H. Ruess, “autoCode4: Structural controller synthesis,” in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2017, pp. 398–404.

[19] A. D. Vieira, E. A. P. Santos, M. H. de Queiroz, A. B. Leal, A. D. de Paula Neto, and J. E. R. Cury, “A method for PLC implementation of supervisory control of discrete event systems,” *IEEE Transactions on Control Systems Technology*, vol. 25, no. 1, pp. 175–191, jan 2017.

[20] D. Gritzner and J. Greenyer, “Synthesizing executable PLC code for robots from scenario-based GR(1) specifications,” in *Software Technologies: Applications and Foundations*. Springer International Publishing, 2018, pp. 247–262.

[21] EPLAN Software & Service GmbH & Co. KG, “EPLAN Engineering Configuration,” https://www.engineeringconfiguration.com/en/configuration/, accessed: 2018-09-10.

[22] S. M. Becker, T. Haase, and B. Westfechtel, “Model-based a-posteriori integration of engineering tools for incremental development processes,” *Software & Systems Modeling*, vol. 4, no. 2, pp. 123–140, May 2005.

[23] D. B. West, *Introduction to Graph Theory (2nd Edition)*, 2nd ed. Pearson, 2000, vol. 2.

[24] R. Heckel, “Graph transformation in a nutshell,” *Electronic Notes in Theoretical Computer Science*, vol. 148, no. 1, pp. 187–198, Feb. 2006.

[25] R. Milner, “Bigraphs and their algebra,” *Electronic Notes in Theoretical Computer Science*, vol. 209, pp. 5–19, Apr. 2008.

[26] G. Busatto, G. Engels, K. Mehner, and A. Wagner, “A framework for adding packages to graph transformation approaches,” in *Theory and Application of Graph Transformations*. Springer Berlin Heidelberg, 2000, pp. 352–367.

[27] A. Aho and J. Ullman, “Translations on a context free grammar,” *Information and Control*, vol. 19, no. 5, pp. 439–475, Dec. 1971.

[28] A. P. Estrada-Vargas, E. Lopez-Mellado, and J.-J. Lesage, “An identification method for PLC-based automated discrete event systems,” in *49th IEEE Conference on Decision and Control (CDC)*. IEEE, Dec. 2010.

[29] “Status of the PLCopen TC6,” http://www.plcopen.org/pages/tc6_xml/xml_intro/index.htm, 2018, Accessed: 2018-10-10.