IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 20 September 2022, accepted 23 November 2022, date of publication 30 November 2022,
date of current version 6 December 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3225736

RESEARCH ARTICLE

Transformation of GRAFCET Into GAL for Verification Purposes Based on a Detailed Meta-Model

ROBIN MROSS[1], ARON SCHNAKENBECK[2], MARCUS VÖLKER[1], ALEXANDER FAY[2], (Senior Member, IEEE), AND STEFAN KOWALEWSKI[1]

[1]Lehrstuhl Informatik 11, RWTH Aachen University, 52074 Aachen, Germany
[2]Institut für Automatisierungstechnik, Helmut-Schmidt-Universität, 22043 Hamburg, Germany

Corresponding author: Robin Mross (mross@embedded.rwth-aachen.de)

This work was supported by the Deutsche Forschungsgemeinschaft through the project Analyse von GRAFCET-Spezifikationen zur Erkennung von Entwurfsfehlern (AGRAFE).

ABSTRACT The graphical modeling language GRAFCET is used as a formal specification language in industrial control design. To use these formal specifications for model-driven development of control code it is beneficial to ensure their syntactical and semantic correctness. Therefore in this paper, a detailed meta-model for GRAFCET is presented, which takes so-called terms into account, i.e. logical and arithmetic expressions in conditions and assignments. The meta-model and additionally proposed invariants allow the creation of syntactically correct GRAFCET instances. Based on this, a translation of GRAFCET to Guarded Action Language (GAL) is presented. The resulting transition system in GAL forms the basis for a semantic analysis of the GRAFCET instances by means of model checking in future research. Finally, the models are then employed for automatic code generation in Structured Text.

INDEX TERMS Industry automation, formal model, formal verification, model checking, model-driven engineering, GRAFCET.

I. INTRODUCTION

An important part of the engineering process of cyber-physical systems is the specification of the control code. Formal methods can avoid the error-prone human interpretation of informal specifications and increase the quality of control code by automatic code generation [1]. However, formal control design has not gained much acceptance in practice. One reason given is that existing concepts and means of description from the field of software development are not suitable for control design in industrial automation [2].

GRAFCET is an internationally standardized graphical specification language according to IEC 60848 [3] (the term Grafcet refers to an instance of GRAFCET), which makes it possible to describe hierarchical control structures. If the behavior of the control system is described with a Grafcet, it can be used as a specification in the formal control design. Since the cost of correcting errors in software systems increases exponentially as the development phase progresses [4], it is beneficial to verify the specification as early as possible. Therefore, this paper will present a basis for verifying modeled control behavior, i.e. the formal specification instead of the actual implementation. The formal specification in form of a Grafcet is transformed into control code via model-driven code generation only after successful verification and, if necessary, iterative revisions.

The goal of this paper is to propose a groundwork for model checking of GRAFCET integrated in the model-driven development process of control code. The development process presented in this paper consists of three steps: (I) Using means to create syntactically correct Grafcets that (II) can be transformed into a transition system used as input language for a model checker. After the verification process an (III) automatic transformation of GRAFCET into control code ensures semantic equivalence of the verified specification and the implementation.

The associate editor coordinating the review of this manuscript and approving it for publication was Yang Liu.

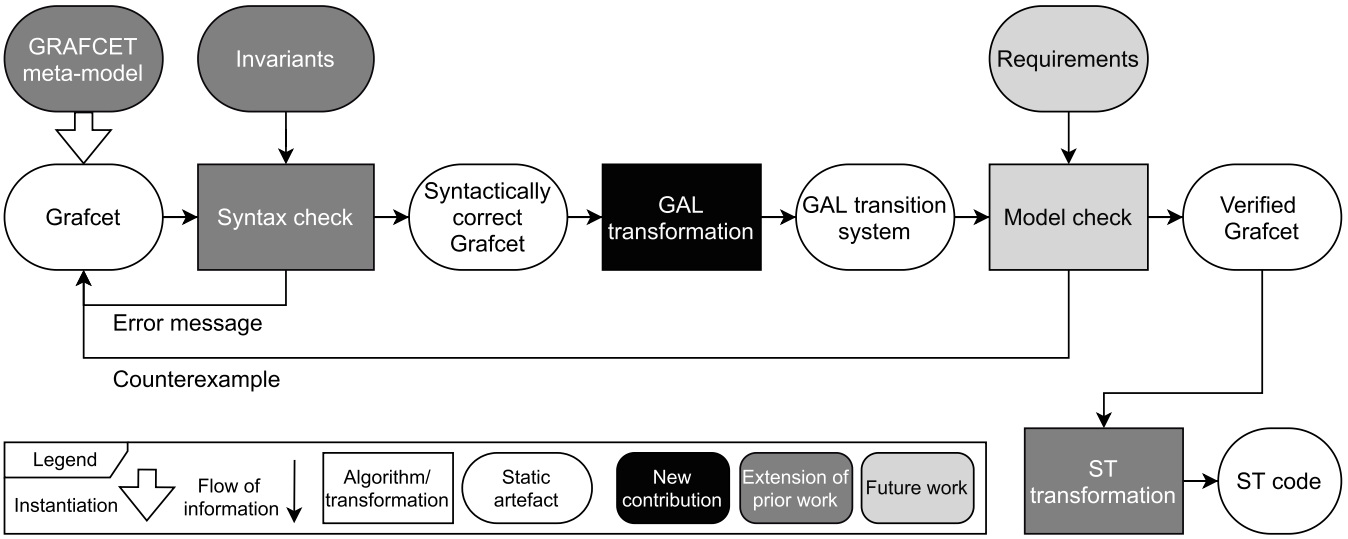

FIGURE 1. Model-driven development of control code using GRAFCET and formal verification.

An overview of this process is given in Fig. 1. After discussing related work in Section II we give some preliminaries in Section III which are used in our main contributions described in Section IV. There, the first subject is to completely define the abstract syntax of the used specification language GRAFCET in Section IV-A. This is done by building on the work of Julius et al. [1], via a meta-model (Fig. 1: GRAFCET meta-model) including the terms in conditions (e.g. a transition condition *A AND B*) and assignments which are not considered in research so far. Besides the meta-model, invariants are necessary to formulate additional syntactical rules, which we investigate in Section IV-B. After establishing syntactical correctness, a transformation to Guarded Action Language (GAL) [5] is presented in Section IV-C. GAL is a language for modeling transition systems and is suitable for model checking. Thus, the transformed GAL transition system can be used as an intermediate representation for verification of Grafcets using model checking in a subsequent work. Additionally, a transformation of the verified Grafcets, based on Julius et al. [6], into Structured Text (ST) [7] is presented in Section IV-D. We evaluate this approach in Section V and draw conclusions in Section VI.

II. RELATED WORK

Although the GRAFCET formalism provides a considerable number of additional modeling mechanisms, in its basic core it is in many ways similar to Petri nets: Transitions and steps, connected alternately by arcs, form a bipartite graph and transitions can only fire when all preceding steps are active, to name two shared concepts. Petri net variants themselves are powerful specification means for event driven systems that have been thoroughly studied, in particular with respect to verification.

A notable class form Signal Interpreted Petri Nets (SIPN) [8] and Control Interpreted Petri Nets (CIPN) [9], which are well suited for the specification of logic controllers. They provide deterministic behavior and introduce input and output signals associated with transitions and steps for modeling interaction with the environment. These and similar variants have been subject to research with respect to model checking in e.g. [10], [11], [12] and lately [13] as well as code generation for PLC in e.g. [14] and lately [15].

David and Alla [9] as well as El Rhalibi et al. [16] provide a comparison of CIPN and GRAFCET concluding that CIPN can be translated into GRAFCET without semantic loss. The other way around only a subset of GRAFCET, so called *sound Grafcets*, can be translated into CIPN. Sound Grafcets are restricted to *basic* Grafcet elements, i.e., Grafcets without hierarchical concepts, and also do not allow the usage of special constructs such as source transitions. However, these hierarchical concepts proposed by the GRAFCET standard allow for compact modeling of complex systems. They include macro steps, enclosing steps and forcing orders [3]. Based on [9] Schumacher et al. provide a translation of GRAFCET into CIPN by first normalizing hierarchical structures [17]. A drawback of this normalization technique is the loss of hierarchical information in the process, making it difficult to use directly for model checking, as finding the error source in the original Grafcet might become challenging. Furthermore the resulting normalized Grafcets are not sound and therefore most of the well known structural verification techniques for Petri nets [18] are not applicable. Specifying hierarchical behaviour of logic controllers motivates the need for GRAFCET-specific verification approaches before the Grafcets can be transformed into control code.

Since the standard is used in several industrial domains, for example railway transport and the manufacturing industry [19], GRAFCET is already widely known in the respective areas, making it potentially easier to advance the usage of

model-based engineering and verification using GRAFCET in the industry.

A first step toward that goal is to define a meta-model. An overview of the existing meta-modeling approaches of GRAFCET is presented by Julius et al. [1]. Furthermore, they propose a meta-model suitable for model-driven development in PLC programming. The proposed meta-model of IEC 60848 GRAFCET contains all basic elements, synchronizations, time and event conditions and hierarchical structures. However, the formalization of terms, i.e., logical and arithmetic expressions in conditions and assignments, has not been considered in [1]. They are only represented as strings in the target language syntax, which hinders formal verification. Schumacher and Fay [20] use the Petri Net Markup Language (PNML) as exchange format for GRAFCET which is defined in ISO/IEC 15909-2 [21]. The standard uses meta-models according to the Meta Object Facility (MOF) standard to define the structural concepts of Petri nets and to provide an exchange format. One part of the meta-model refers to the terms of the Petri nets. For syntactical correctness, related invariants are provided. However, PNML covers all Petri net formalisms and therefore results in a large amount of specific elements not needed for GRAFCET. Approaches to cover terms, left out by Julius et al., are proposed by Arnold et al. [22] and Nzebop et al. [23], both of which use a grammar and a parser. However, Arnold et al. do not provide a meta-model and therefore their approach can not be adapted to different target languages. Nzebop et al. parse the terms into a meta-model based on the Eclipse Modeling Framework (EMF) and provide additional invariants to ensure the correct syntax. However, the integration of the meta-model of terms into a GRAFCET meta-model (e.g. proposed by Julius et al.) as well as additional invariants which cover this integration are not presented.

To facilitate model checking of GRAFCET, previous work has transformed IEC 60848 into different formal models. In [24], Provost et al. discuss a translation to Mealy machines. Their concept includes that of a stable location automaton, which describes all stable (i.e., not transient) situations of the underlying Grafcet as states. The authors also introduce a mathematical formalization of the standard which Julius et al. expand on [1]. They use this formalization to rigorously describe behavioral aspects of the standard. However, the authors of [24] neither discuss hierarchical concepts such as enclosed steps nor timed elements. A translation to timed automata is given by Cassez in [25], who considers a restriction of the standard to fundamental components such as steps, transitions and timed conditions. However, it includes neither forcing orders nor actions associated with steps. The authors of [26], [27], and [28] provide a time Petri net model for GRAFCET, that also considers timed aspects of the standard, but neither rising and falling edges on internal Boolean variables nor explicit source transitions. Their approach models environmental variables structurally by elements of the Petri nets, rather than by employing SIPN or CIPN. More recently, GRAFCET has been utilized in model checking distributed reconfiguration scenarios in [29]. In their approach, the GRAFCET instances are first transformed into a programming language in order to be verified, such that model checking is not employed on specification level.

There are approaches that consider hierarchical concepts in other specification languages. Frey [30] verifies a hierarchical variant of SIPN. However, the hierarchical concepts proposed in hierarchical SIPN are only comparable to macro steps of the GRAFCET standard and unable to represent enclosures or forcing orders. Klotz et al. [31] verify UML state-charts. They normalize hierarchical concepts using priorities assigned to transitions. This approach can not be adapted to GRAFCET, because priorities of transitions are not supported due to the synchronous execution semantics enforced by the fourth evolution rule [3].

III. PRELIMINARIES

In the following sections, a brief overview of our preliminary work for formalizing GRAFCET and the basic concept of GAL is given.

A. FORMALIZING GRAFCET

Provost et al. [19], among others, have worked towards formalizing IEC 60848 to establish a rigorous framework upon which details of algorithms and transformations can be formulated precisely. Due to the semi-formal nature of the standard itself, different authors have done so in different ways. However, hierarchical concepts are often disregarded. An extension of the work of Provost et al. [19] is given by Julius et al. [32], which also includes the formalization of hierarchical aspects, and serves as a basis for this chapter.

Macro steps are intentionally left out in this work since they do not increase expressivity: Each Grafcet with macro steps can be turned into one without macro steps while staying semantically equivalent. This is done by simply replacing every macro step with their respective macro step expansion chart.

In the following, a Grafcet $G = (V_{in}, V_{int}, V_{out}, C)$ is a model that comprises a set of charts $C \neq \emptyset$ with globally available sets of input variables V_{in}, internal variables V_{int} and output variables V_{out}. Input, internal and output variables can either be Boolean or integral, i.e. v is assigned a value of $\mathbb{Z}$ for all $v \in V_{in} \cup V_{int} \cup V_{out}$ with Boolean variables being limited to the set $\{0, 1\}$. We denote by $V_{inB} \subseteq V_{in}$ ($V_{intB} \subseteq V_{int}$, $V_{outB} \subseteq V_{out}$) the set of Boolean input (internal, output) variables. Given these variables we are now able to construct Boolean expressions with usual relational symbols (such as $=$ and $\leq$) and Boolean operators (such as disjunction $\vee$ and negation $\neg$). A variable may change values caused by an event. In particular, any Boolean expression x may change from 0 to 1 or vice versa. As a short hand notation for this, we use $\uparrow x$ ($\downarrow x$) for such a *rising* (*falling*) *edge* event. When such an event occurs, these variables are set only for the first subsequent evolution when dealing with transient situations. By *CND* we denote the set of all Boolean

expressions over variables in G, while $CND_E \subseteq CND$ denotes all such expressions that require an event to evaluate to true. For example, given $a, b \in V_{in}$, $a \vee \uparrow b \in CND \setminus CND_E$ and $a \wedge \uparrow b \in CND_E$. Let $CND_{NE} \subseteq CND$ be the set of all conditions that never depend on a rising or falling edge event.

Every chart $c \in C$ is a 6-tuple $c = (S, I, E, M, T, A)$, where

- S is a finite set of steps, each of which is either active or inactive at any given time,
- $I \subseteq S$ is the set of initial steps,
- $E \subseteq S \times C$ is the set of enclosing steps,
- $M \subseteq S$ is the set of marked steps,
- $T \subseteq \mathcal{P}(S) \times \mathcal{P}(S) \times CND$ is the set of transitions,
- A is a set of actions.

We use the notation S_c, I_c, E_c, M_c, T_c, A_c to refer to the respective sets of a given chart $c \in C$. The set M_c describes the steps that are activated by the enclosing step, which are graphically denoted by an asterisk next to the step. Every $e \in E_c$ describes an enclosing step, which translates formally to $e = (s, c_{enc})$ for a step $s \in S_c$ and a chart $c_{enc} \in C$. If an enclosing step becomes active, it activates all steps $m \in M_{c_{enc}}$. If an enclosing step becomes inactive, it deactivates all steps $s \in S_{c_{enc}}$. We say that c is *enclosed* iff $M_c \neq \emptyset$. Further we have disjoint sets of steps, that is $S_c \cap S_{c'} = \emptyset$ for every $c' \in C$ with $c \neq c'$. Every $s \in S_c$ induces a new Boolean variable x_s which indicates the activation status of s and is true iff the step is active in the current situation. These variables can be used in Boolean expressions (CND).

A transition $t \in T_c$ is a triple $t = (U, D, b)$, where

- $U \subseteq S_c$ is the set of immediately preceding steps,
- $D \subseteq S_c$ is the set of immediately succeeding steps,
- $U \neq \emptyset \vee D \neq \emptyset$,
- $b \in CND$ is the transition condition.

We also call U the *upstream* and D the *downstream* of t. Note that if $b \in CND \setminus CND_E$ the transition may cause a transient situation. We say that t is *enabled* if x_s is true for every $s \in U$. We say that t can *fire*, if it is enabled and b is true. Finally, we formalize the set of actions A_c. The standard defines different types of actions: continuous actions (A_{cont}), forcing orders (A_{fo}) and stored actions, of which the latter is subdivded into the three categories of action on event (A_e), action on activation (A_{act}) and action on deactivation (A_{deact}). These sets are assumed to be disjoint. Let $A_c = \{A_{cont}, A_{fo}, A_e, A_{act}, A_{deact}\}$. Every element of A_{cont} is a triple (s, v, b), where

- $s \in S_c$ is the associated step,
- $v \in V_{outB}$ is a Boolean output variable,
- $b \in CND_{NE}$ is the action condition.

We say that a continuous action is *active* if x_s and b are true. Several charts in G may employ continuous actions on the same output variable v. In this case, v is set to true if at least one of these continuous actions is active. Note that v can not be used by any stored action.

Every element of A_e is a tuple (s, v, val, b), where

- $s \in S_c$ is the associated step,
- $v \in V_{int} \cup V_{out}$ is an internal or output variable,
- val is an expression yielding a value in the respective domain, e.g. $val \in \mathbb{Z}$,
- $b \in CND_E$ is the action condition.

An action on event sets v to val if x_s and b are true.

Every element $a \in A_{act} \cup A_{deact}$ is a tuple (s, v, val), where

- $s \in S_c$ is the associated step,
- $v \in V_{int} \cup V_{out}$ is an internal or output variable,
- val is an expression yielding a value in the respective domain, e.g. $val \in \mathbb{Z}$.

Such an action a assigns v to val if x_s changes value from false to true (true to false) if $a \in A_{act}$ ($a \in A_{deact}$). This corresponds to the rising (falling) edge semantics discussed earlier.

Finally, every element of A_{fo} is a tuple (s, c_{forced}, S), where

- $s \in S_c$ is the associated step,
- $c_{forced} \in C$ is the chart which is to be forced,
- $S \in (\mathcal{P}(S_{c_{forced}}) \cup \{*, init\})$.

A forcing order action is regarded as a special kind of continuous action. It is active while x_s is true and forces c_{forced} into the situation specified by S. If $S = *$, then the current situation in c_{forced} is retained for as long as s is active. If $S = init$ then c_{forced} is set to its initial situation. Otherwise it is set to the specified situation (element of the power set $\mathcal{P}(S_{c_{forced}})$). Note that $init$ always represents a situation which is part of $\mathcal{P}(S_{c_{forced}})$. No transition of $T_{c_{forced}}$ can fire while x_s is true.

The presented formalization will be used in Section IV-C in which we describe the translation.

B. GAL

Guarded Action Language (GAL) is a modeling language which allows for a compact description of a transition system. It is specifically designed to be a target in a model-to-model transformation process. Before we describe the transformation from IEC 60848 to GAL in Section IV-C, we provide a brief introduction to GAL [5] itself. GAL is the backend language of the model checker ITS-Tools [33]. A GAL system comprises variables, arrays and transitions, where the only variable type available is an integer, limited to the range $[-2^{31}, 2^{31} - 1]$. The transition system is then made up of the states that are defined by the valuations of the variables. Variables are initialized with the value 0, if not specified otherwise. This then describes the unique initial state. Transitions describe how variables are updated and lead to new states. To this end, a transition in GAL comprises two parts: a guard and a list of actions. The guard is a Boolean expression over the variables, together with constants in $[-2^{31}, 2^{31} - 1]$ and relational symbols such as $=$ or $>$. Boolean expressions can be combined with usual Boolean connectives to form new Boolean expressions. A transition starts at every state that meets this guard and ends in another target state. That target state is defined by the valuation in the current state together with the action list that indicates how variables are updated. The action list allows for control structures such as the usual

if, *then*, *else*. A transition can optionally bear a label. If a transition has a label, it in itself does not automatically induce any transition in the resulting transition system. However, that transition can be called from within the action list of another transition, much like a function in programming languages. The action list of a transition, including calling other transitions, is handled atomically, such that intermediate results are not visible in the state space.

Before we present the translation of a Grafcet to GAL we concern ourselves with syntactical aspects in Sections IV-A and IV-B.

IV. CONTRIBUTION

In [1], a meta-model for GRAFCET is presented which defines an abstract syntax that forms the basis for model-driven software development in [6]. However, these concepts do not consider the verification of GRAFCET instances. Furthermore, the meta-model presented in [1] does not take into account the modeling of Boolean, logical, and arithmetic expressions which occur in GRAFCET in conditions and value assignments and are called terms in this work. For verification, the modeling of such terms is necessary to allow a more accurate analysis of execution behavior.

Therefore, in this work, the abstract syntax of terms in GRAFCET is defined in Section IV-A using a meta-model (i.e. a model used for instantiation of Grafcets and its terms) designed according to the MOF standard [34]. The GRAFCET meta-model then uses these expressions. In addition, the syntactic rules for the formation of terms that are not covered by the meta-model will be described using invariants (Section IV-B). Next, we provide a description of the transformation process of a Grafcet into GAL in Section IV-C. We elaborate on how this model simulates the behavior of the transformed Grafcet. Finally, we provide a transformation of GRAFCET into IEC 61131-3 ST code in Section IV-D.

A. META-MODEL OF TERMS

The meta-model of terms (Fig. 2) is based on the standard of High Level Petri Nets [21]. However, it has to be adapted to represent GRAFCET-specific elements such as step variables and events.

As shown in Fig. 2, terms can be of the type variable or operator. Operators can have sub-terms allowing a recursive structure. Variables refer to a variable declaration which describes the variable. Modeling the sorts allows to verify the syntax of terms using invariants (cf. Section IV-B). Therefore, note that the correct references to the sorts are not necessarily required for following transformations. Sorts of terms can be derived, either from the variable declaration or from the type of the operator. An operator contains its output sort and references its input sorts, depending on the number of sub-terms. E.g., the *AND* operator in the term *A AND B* references an instance of the class `Bool` as `output`, meaning the operator returns a value of the type Boolean. It has two additional children (`Variable` instances *A* and *B*) each stored as `subterm`. The sorts of the sub-terms are referenced by the *AND* operator via `input`.

To use instances from the meta-model of terms in Grafcets, the GRAFCET meta-model imports it. Fig. 3 shows an excerpt of the GRAFCET meta-model presented in [1] (dashed lines indicate the neglected parts of the model). The root of the model is the Grafcet itself. The Grafcet has an instance of the `VariableDeclarations` class, containing the variable declarations, i.e. of input, output and internal variables. A transition has a condition, containing a term. An action controls the variable `variable`. A stored action has a value which will be assigned to the corresponding variable when the action is performed. A stored action on event can be modeled using a stored action instance with a condition. The class `EventCondition` proposed in [1] is replaced by the rising and falling edge operators in the terms meta-model.

B. FORMALIZING INVARIANTS

In the following, invariants are presented to ensure additional syntactical rules of Grafcets. The rules can be divided into two categories. First, the consistency of the terms independently of the usage in GRAFCET has to be ensured. Most of these rules are already formalized in [21] and are omitted in this work. Second, we formalize GRAFCET specific rules mentioned in the IEC 60848 standard [3]. In this work as well as in [21], the Object Constraint Language (OCL) is used to formalize these rules as invariants. OCL is a formal language independent of a programming language to describe expressions on UML models. In general, OCL expressions are written in the context of a specific model instance, to which the keyword `self` refers. The operator "." is referring to an attribute. This can result in a single attribute or a set, called collection. Navigation from a collection is done with the operator "->".

```
1  context Operator Inv:
2  self.subterm->collect(subterm|subterm
       .sort) = self.input->asSequence()
```

Listing 1. Consistency of sorts.

Additionally to the rules defined in [21], the OCL invariant in Listing 1 ensures the consistency of the sorts of the sub-terms of an operator. The invariant ensures for every instance of the class `Operator` that `sort` of the sub-term(s) refers to the same sort as `input` of the operator. Therefore, the sorts of the sub-terms are collected using the `collect()` operator. The `asSequence()` operator is used to cast into the correct type of collection.

The remaining rules are described in the following shortly and the related listings can be found in Appendix A.

If a variable is declared as step variable it has to be of type Boolean and `step` has to be set (Listing 3).

The term of a condition has to be a logical expression returning a Boolean (Listing 4).

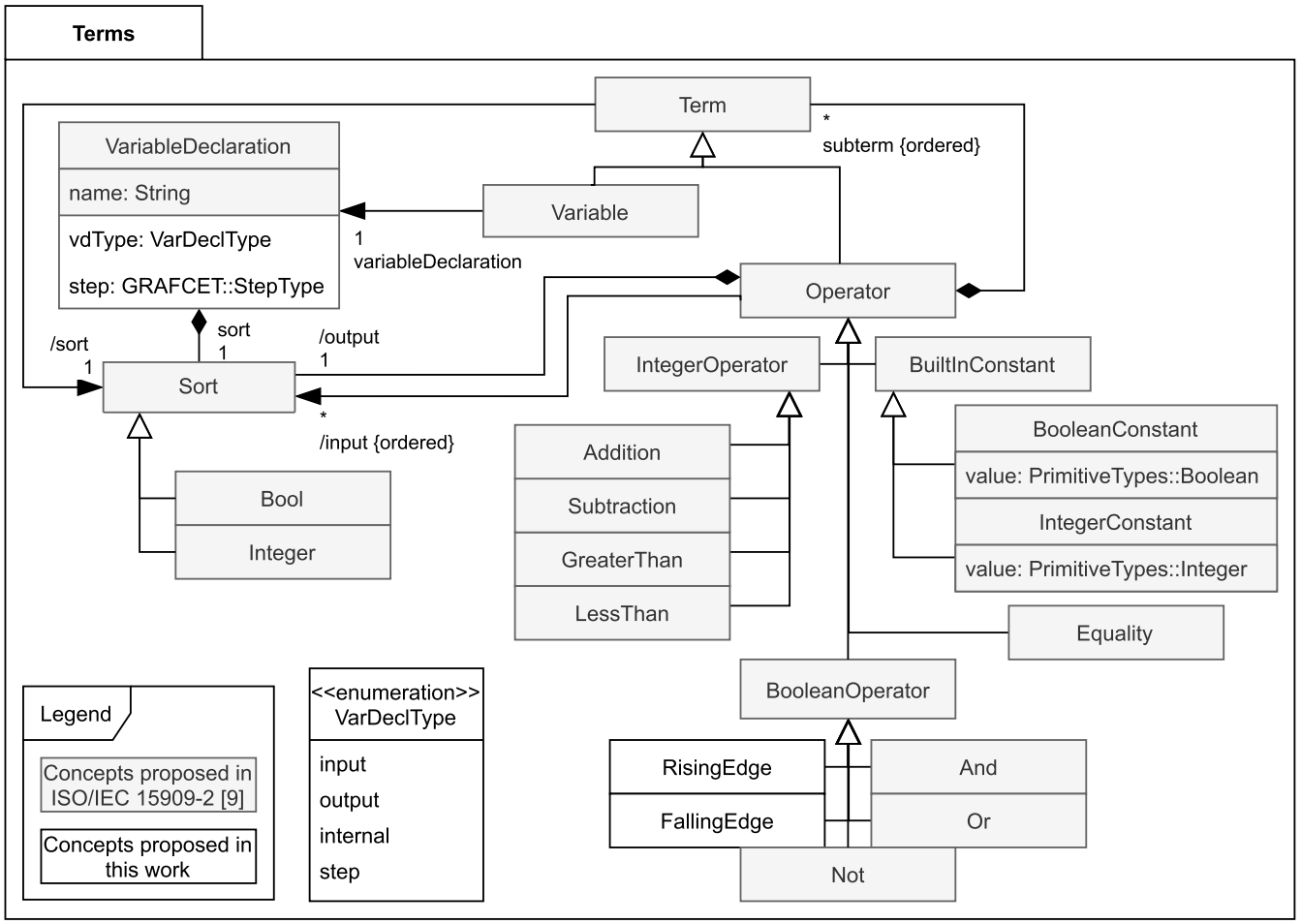

FIGURE 2. Meta-model of terms.

According to the standard, output variables must not be read (Listing 5) and input variables must not be written (Listing 6) [3]. Therefore, instances of the class `Condition` can only be an expression from input, step or internal variables, i.e. no output variable should occur in the term. Hence, it has to be checked if the corresponding `term` is either one of the correct variable types, or an instance of the `Operator` class containing only the correct variable types.

On the other hand to ensure no input variables are written, `variable` of an action has to be an output variable (for continuous and stored actions) or an internal variable (only allowed for stored actions) [3].

The remaining rules ensure the correct sort of variables and condition types of continuous (Listing 7 and 8) and stored actions (Listing 9 and 10). According to the standard, continuous actions can only perform actions on Boolean variables [3]. Furthermore, potential conditions of continuous actions (so called assignation conditions) must not include an event, i.e. a `RisingEdge` or `FallingEdge` instance [3].

Stored actions on the other hand can assign both logical or arithmetic expressions to variables [3]. However, it has to be ensured that variable and assigned term have the same sort. Similar to continuous actions, according to the standard, conditions on stored actions must contain an event [3]. However, checking if the appropriate term contains a `RisingEdge` or `FallingEdge` instance does not completely ensure that the condition is an event. E.g. the term *A OR RisingEdge(B)* is true in the case of *A* is true without an event occurring.

The usage of the presented meta model and the formalized invariants ensures a syntactically correct Grafcet. We now turn to questions concerning the semantics of a given Grafcet, for which we describe a transformation in the next Section IV-C. The result is a verifiable model which is suitable for verification techniques, in particular model checking.

C. GAL-TRANSFORMATION

We now illustrate how a model in GAL can be constructed from a given Grafcet *G*. The goal is to have a transition system which describes the behavior of the system that is modeled in *G*, based on the formalization provided in Section III-A. The resulting model roughly consists of two parts:

- The parts that are generated from the elements of *G*.

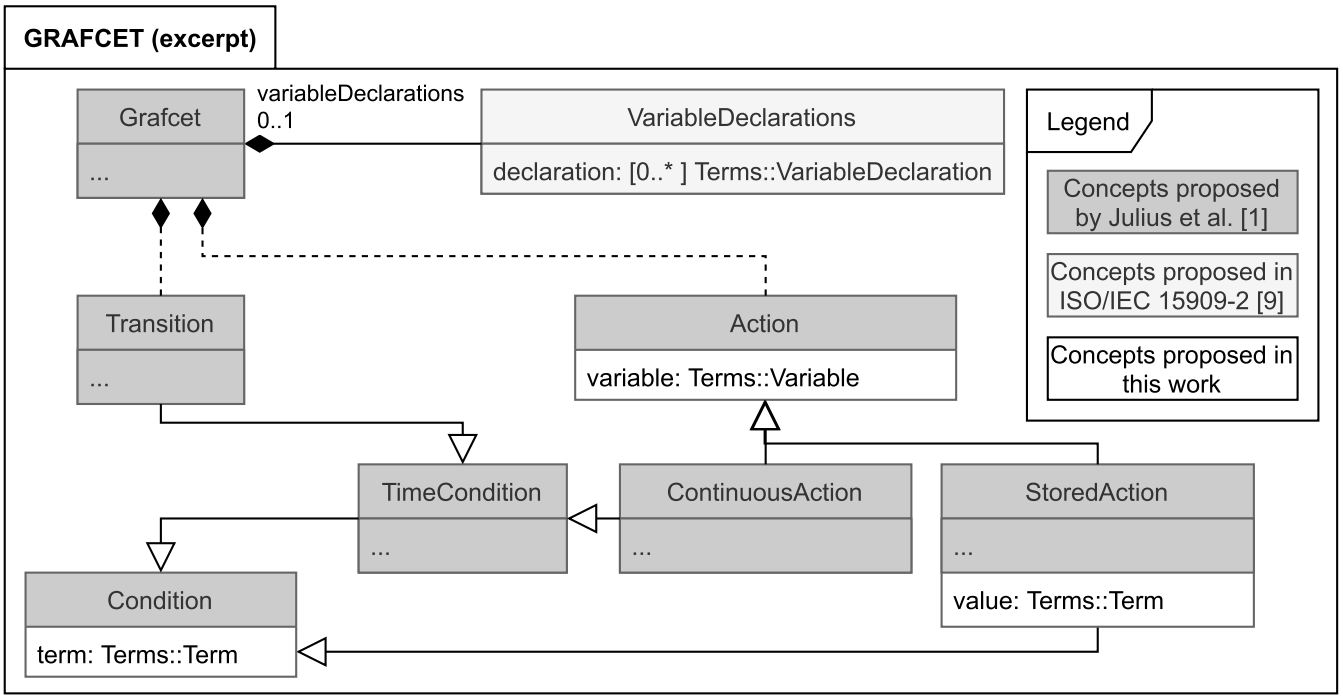

FIGURE 3. Excerpt of GRAFCET meta-model.

- The evolution manager, which is GAL code that is part of every G transformed by this method. It is structurally fixed and coordinates how evolutions, execution of actions, input changes and transient situations are dealt with.

We begin with the former. The most fundamental elements in GRAFCET arguably are the variables. Since GAL only allows for integers, Booleans form a special case with the usual interpretation that 0 corresponds to false and 1 to true. Boolean variables in G are therefore translated the same way as integer variables. All these variables in the resulting model are initialized to 0, in accordance with the standard. These variables are globally available to all charts of G.

Next, we consider a chart $c \in C$. For every step $s \in S_c$ we create three variables: The step activity variable x_s, a variable *MarkActivate* and one *MarkDeactivate*, each of which is Boolean. All variables are initialized to 0, except for $s \in I_c$. In that case, the corresponding *MarkActivate* is initially set to 1. They ensure the compliance with the evolution rules four and five of the standard, which state that all transitions that can be simultaneously fired do so and that a step remains active if it is to be activated and deactivated at the same time. This decouples the evaluation of transitions from the actual state changes of the steps. Accordingly, three labeled transitions are generated for every $s \in S_c$. One deals with the activation of the step. It sets x_s to true and also executes all actions on step activation that are associated with this step, that is for all $(s, v, val) \in A_{act}$ an assignment $v = val$ is generated. Another transition that deals with the step deactivation is constructed analogously. The final labeled transition is concerned with the evaluation of the marker variables. It checks if the system is to call the step's (de)activation transitions or not. More precisely, if *MarkActivate* for s is true and x_s is false, the step's activation transition is called. Else, if *MarkDeactivate* is true, *MarkActivate* is false and x_s is true, the step's deactivation transition is called. In all other cases, the state of this step does not change and in any case, both marker variables are reset to false.

So far, the marker variables play an important role in determining how the situation changes, but they are not yet set. Usually, these markers are set by the transitions of G, which we describe next. For every $t \in T_c$ a labeled GAL transition is defined which checks if t is fireable. As described in the standard, $t = (U, D, b)$ is fireable if x_s is true for all $s \in U$ and b is true. If fired, all $s \in U$ are marked for deactivation, while all $s' \in D$ are marked for activation. If in any chart $c_{forcing} \in C$ there is a forcing order action $a \in A_{fo}$ with $A_{fo} \in A_{c_{forcing}}$ such that $a = (s, c, S)$, then the guard is extended by the conjunction of $\neg x_s$ for all such forcing orders a. This ensures that the chart can not evolve while any forcing order is active. For example, the guard of a source transition within a chart that can be subject to a forcing order issued by steps s_{13} and s_{82}, while the transition itself has the condition $a \wedge b$, is $a \wedge b \wedge (\neg x_{13} \wedge \neg x_{82})$.

Actions on events can be caused by internal events such as input changes. Such an action can be fired if the condition is true while the associated step is active. Since the associated actions can potentially influence the evaluation of other conditions, we employ a similar system as with step activations to ensure that actions that can fire simultaneously will end up

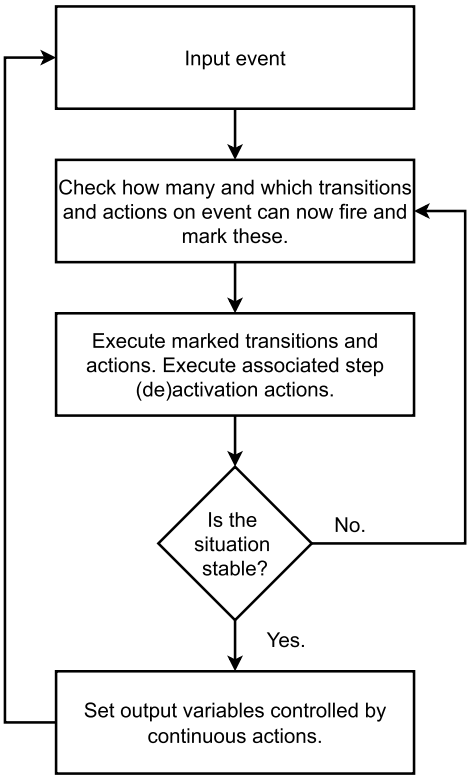

FIGURE 4. Overview of the evolution manager which simulates a given Grafcet.

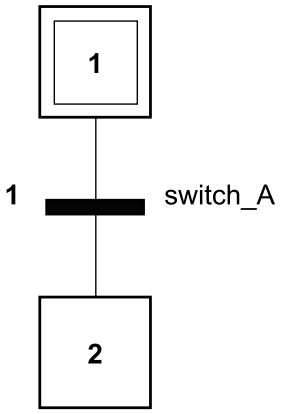

FIGURE 5. Simple Grafcet containing basic elements.

doing so. To this end, we add a Boolean variable that signals that the event is marked for execution. Every such action also creates two labeled transitions: One which marks the action for execution and one that executes the associated assignment. The former consists merely of a condition that checks whether the associated condition is true while the linked x_s is true as well. If this is the case, the marker variable is set. The latter transition executes the assignment (*action*) if the marker variable is true and then also resets the marker to false. The last type of actions, other than continuous actions which are discussed later, are forcing orders. A forcing order comprises two parts: Upon activation, it enforces a specific situation in the underlying chart $c_{forced} \in C$, while it also disables all transitions in that chart, i.e., none of these transitions can fire. The behavior of the first aspect is similar to the behavior of a stored action on step activation. Therefore, if the step that performs the forcing order becomes active, we set the appropriate *MarkActivate* and *MarkDeactivate* variables for all steps in the forced chart to their respective value within the step activation transition introduced earlier. Note that if the forcing order is of type $*$, then none of such markers are set. We do not modify the step activity variables directly to ensure that associated actions on step (de)activation are also performed. Finally, in order to implement the freezing aspect of the forcing order, we ensure that for every $t \in T_{c_{forced}}$ the condition is extended to include the negated x_s for the forcing order issuing step $s \in S_c$ as a conjunct as described above. Enclosing steps are simulated in a similar way: Upon step activation, the desired situation is set in the enclosed chart $c_{enc} \in C$. The desired situation is described by the set of steps that are marked, i.e., $M_{c_{enc}}$. Upon deactivation, all steps are deactivated in c_{enc}. We again set the appropriate marker variables instead of modifying the step activity variables directly. Note that enclosing steps do not include any freezing aspects, unlike forcing orders.

So far we have provided translations for the different elements of the standard. We need some mechanism in the target model that describes how G evolves, when a situation is stable and how the different types of variables are updated, i.e., we need the aforementioned *evolution manager*. An overview of the manager is given in Fig. 4. The overall flow of action of this manager can be summarized as follows: If the situation is stable, simulate an event, e.g. a change of a Boolean input variable. Next, evaluate which transitions and actions can now be executed and mark these respectively. Subsequently, the marked transitions are fired and the marked actions executed. This usually results in a change of situation and internal or output variables. The new situation may lead to other transitions being enabled, which may be fireable due to the current variable assignments. We therefore have to check again if any transition or action can be executed, i.e., we repeat the previous steps. If this is the case then the current situation is transient. Assuming that a stable situation is reached, we now have to properly evaluate the continuous actions, as their values are not set during transient situations. For each output variable $v \in V_{outB}$ that is used in the context of such an action, a labeled transition is generated which checks if at least one of the steps that have a continuous action on v is active. If that is the case, v is set to true, otherwise to false. Finally, we go back to the first step. Note that an infinite transient loop can still be modeled with this transformation. In fact, this model can be used to check for the existence of such an usually undesired loop by means of model checking. The manager is implemented mostly in a single transition, which calls the labeled transitions introduced earlier, in order to, e.g., check whether a transition is fireable and to perform step (de)activations. The resulting GAL code of the example in Fig. 5 is given in Appendix B. This generated model can be

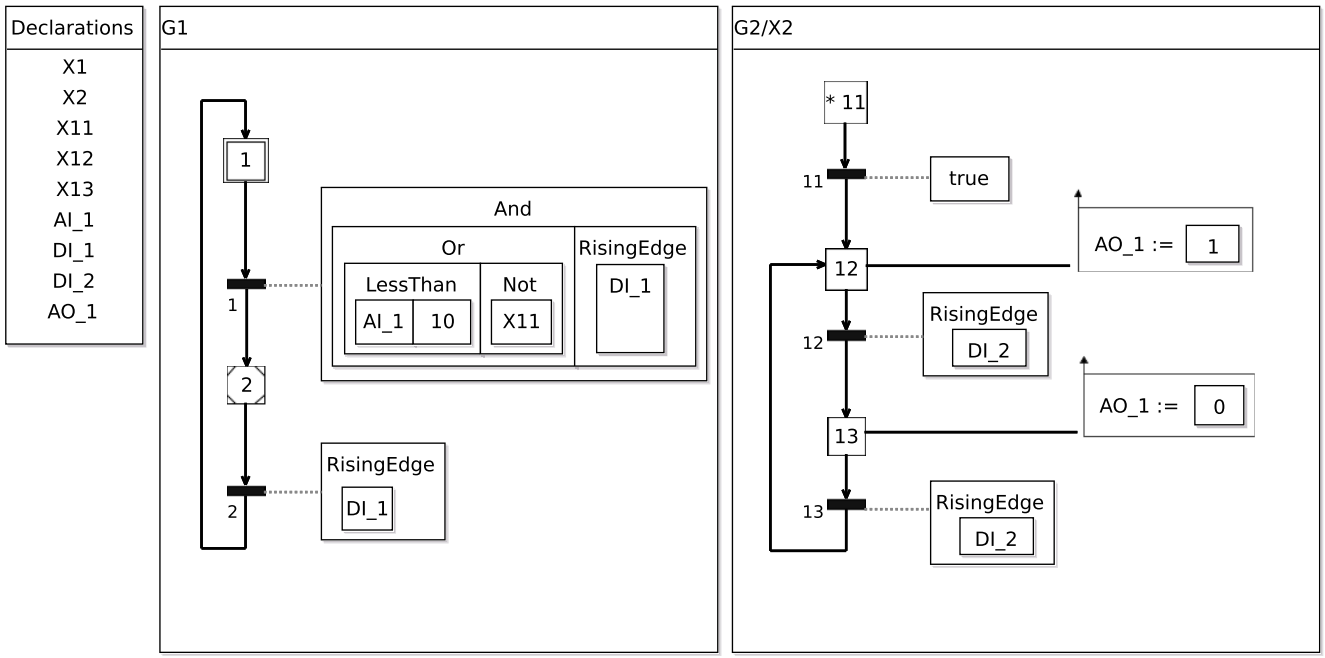

FIGURE 6. Example Grafcet consisting of two Grafcet charts.

used for verification, in particular model checking, as shown in Fig. 1.

D. TRANSFORMATION IN ST-CODE

After verification of the Grafcet by means of model checking, it can be transformed into control code, as shown in Fig. 1. The target language is one of the IEC 61131-3 languages as it is the most common standard in PLC programming. As discussed in [6], ST is the most suitable for the requirements. IEC 61131-3 proposes three different kinds of so called Program Organization Units: Functions, Function Blocks (FB) and Programs. The authors of [6] also discuss that FB are the most suitable Program Organization Units because they can store states and can call each other. In the transformation concept, each GRAFCET chart corresponds to a FB and the communication between charts takes place via global variables. This has the advantage of keeping the Grafcet's hierarchical structure. An example of an application would be distributed control code. Local variables would have the advantage of more precise read and write permissions. However, the code is more readable and less complex without the cascading transfer of local values between the GRAFCET charts.

The main concept of the transformation is already described in [6]. To support the presented model of the terms, the transformation has to be adjusted. In the ST code, global variables for input, output and internal variables have to be declared. Step variables used in the terms have to be mapped to existing step variables proposed by [6]. Finally a recursive algorithm generates a string in ST syntax from the term objects. Thereby, for rising (falling) edges the standard ST Function Block R_TRIG (F_TRIG) [7] is used.

The proposed generation of PLC code forms the last step of the model-driven development process proposed in Fig. 1.

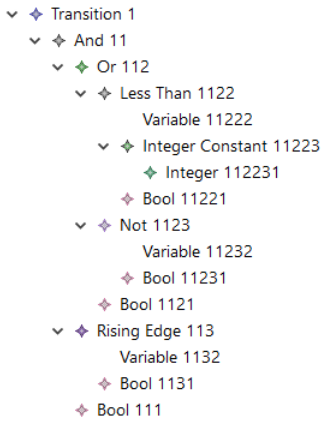

FIGURE 7. XMI representation of transition 1 of the example Grafcet in Fig. 6.

V. EVALUATION

To evaluate the proposed model-driven development process we first show the implementation of the tool chain before applying it on an example presented in Fig. 6.

A. IMPLEMENTATION

Two different tools have been implemented: An Eclipse Modeling Framework (EMF) [35] based graphical editor for the syntactical check and the transformation of GRAFCET to ST code as well as a C++ based tool to perform the transformation of GRAFCET to GAL. XMI is used as exchange format. The serialization to XMI is provided by EMF.

The meta-models are implemented as Ecore files. Sirius is used to design a graphical editor which allows to instantiate the GRAFCET meta-model. The terms are represented in a recursive tree-based manner. Thus, the expressions have a

variable as root element or an operator as a root element with additional operators in a recursive manner. This allows the creation of terms independent of the target language, as shown in Fig. 6 at transition 1. For derived references in the meta-model, a rule-based instantiation of the corresponding classes is possible and implemented using Sirius. E.g., to add a sub-term the user has to create the `Operator` instances manually using the editor. However, the `Sort` instance `output` is created automatically and the references `sort` and `input` are added.

The invariants written in OCL can be checked using Eclipse OCL. The invariant check provides feedback to the user on which class instance violates which invariant.

The implementation of the GAL transformation is capable of using the exported XMI file as input. The transformation can be configured in detail by flags that alter small nuances of the resulting code which is due to some unclear formulations of the standard with respect to the behavior.

For the transformation to ST code the EMF-API is used. The ST code output format is PLCOpenXML [36], a vendor neutral exchange format for IEC 61131-3 control code.

B. EXAMPLE

Fig. 6 shows a Grafcet composed of two GRAFCET charts G1 and G2, whereby G2 is controlled by the enclosing step 2 in G1. The condition at transition 1 shows the tree-like graphical structure of the terms. The corresponding visualization of the XMI file in the EMF editor is shown in Fig. 7. The referenced classes for the datatype are visible. This example has been transformed into GAL using the procedure described in Section IV-C and the resulting state space has a size of 169. In order to evaluate elements of the behavior of the provided Grafcet, the generated GAL code has been verified in the model checker ITS-Tools [33]. Several properties have been evaluated and the instance, for example, fulfills the property that it is impossible to reach a situation in which step 1 and step 12 are active simultaneously. This can be formulated by the Computation Tree Logic (CTL) expression $AG(\neg(x_1 \wedge x_{12}))$. This implementation is proof-of-concept and ongoing research is focused on making the approach tractable on larger systems, the details of which are the subject of future publications.

An excerpt of the transformed ST code regarding transition 1 from Fig. 6 is shown in Listing 2, which is divided into three sections. First, the global variables, used in the transition conditions, are declared (Lines 1-4). Second, the local variables for the FB corresponding to the GRAFCET chart are declared (Lines 5-7). From Line 8 on, the actual FB is shown. The transition is transformed into an `IF` statement starting at Line 13 and requires the transition condition as well as the Boolean variable `T_1`. `T_1` is true when transition 1 is enabled i.e. step variable `Step_1` of step 1 is true (Line 10). To detect the rising edge of the input variable `DI_1` the `R_TRIG` FB declared in Line 7 is called in Line 12 and

```
1  //declaration of global variables:
2  Step_11: BOOL := false;
3  DI_1: BOOL := false;
4  AI_1: INT := 0;
5  //declaration of local variables:
6  T_1: BOOL := false;
7  RTrig_T1_1: R_TRIG;
8  //FB of global grafcet:
9  ...
10 T_1 := (Step_1);
11 ...
12 RTrig_T1_1(CLK := DI_1);
13 IF ((((AI_1 < 10) OR ( NOT Step_11))
       AND (RTrig_T1_1.Q))AND(T_1)) THEN
14 ...
15 END_IF
16 ...
```

Listing 2. Excerpt of ST code showing transition 1.

the value is requested in Line 13. The omitted body of the `IF` statement (Line 14) would deactivate step 1 and activate step 2 according to the GRAFCET evolution rules.

The modeled Grafcet was transformed into ST code completely and the code was simulated in CODESYS [37], resulting in expected behavior.

VI. CONCLUSION

In this work a complete GRAFCET meta-model according to the standard [3] has been presented which can be used to guarantee syntactical correctness of a given Grafcet. It is based on the work of Julius et al. [1] and has extended their meta-model by adding a representation of terms. To this end, invariants formulated in OCL have been provided in detail, which describe syntactical rules specific to GRAFCET. Examples of this are that output variables are never read or that input variables are never written. Another example is a rule which ensures that continuous actions only write Boolean variables, as it is demanded by the standard. This work therefore provides the basis for static analysis methods. A formalization, based on the work of Provost et al. [19] and Julius et al. [32], has been provided which describes all elements of the standard except for timed aspects and macro steps. In this work, it has mainly been used to describe the transformation of GRAFCET to GAL. The translation creates a model that simulates the behavior of a given Grafcet. Unlike methods discussed in Section II our approach covers structural elements of the standard such as enclosures, forcing orders, source transitions and retains hierarchical information. The transformation has been evaluated on an example, and with the help of ITS-Tools [33] its behavior exemplarily tested. Details on the verification process itself using model checking in this tool chain have been left out of this work, but are the topic of on-going research. Existing contributions shown in Section II could be adapted to the verification of GRAFCET using this approach.

APPENDIX A
OCL LISTINGS

```
context VariableDeclaration Inv:
self.vdType = VarDeclType::step
implies self.sort.oclIsTypeOf(Bool)
    and step <> null
```

Listing 3. Sort of step variables.

```
context Condition Inv:
self.term <> null
implies self.term.sort.oclIsTypeOf(
    Terms::Bool)
```

Listing 4. Sort of conditions.

```
context Condition Inv:
self.term <> null
implies if self.term.oclIsTypeOf(
    Terms::Variable)
then self.term.oclAsType(Terms::
    Variable).variableDeclaration.
    vdType <> Terms::VarDeclType::
    output
else self.term.oclAsType(Terms::
    Operator)->closure(term: Terms::
    Term| term->selectByKind(Terms::
    Operator).subterm) ->selectByKind(
    Terms::Variable)->select(var|var.
    variableDeclaration.vdType = Terms
    ::VarDeclType::output)->size() = 0
endif
```

Listing 5. Variable types in conditions.

```
context Action Inv:
self.variable.variableDeclaration.
    vdType = terms::VarDeclType::
    output
or if self.oclIsTypeOf(StoredAction)
then self.variable.
    variableDeclaration.vdType = Terms
    ::VarDeclType::intern
else false endif
```

Listing 6. Variable types in actions.

```
context ContinuousAction Inv:
self.variable.variableDeclaration.
    sort.oclIsTypeOf(Terms::Bool)
```

Listing 7. Sort of variables in contiuous actions.

```
context ContinuousAction Inv:
self.term <> null and not (self.term.
    oclIsTypeOf(Terms::Variable))
implies self.term.oclAsType(Terms::
    Operator)->closure(term: Terms::
    Term| Term->selectByKind(Terms::
    Operator).subterm)->selectByKind(
    Terms::RisingEdge)->size()
+ self.term.oclAsType(Terms::Operator
    )->closure(term: Terms::Term| term
    ->selectByKind(Terms::Operator).
    subterm)->selectByKind(Terms::
    FallingEdge)->size() = 0
```

Listing 8. Event type of continuous actions.

```
context StoredAction Inv:
self.variable.variableDeclaration.
    sort.oclType() = self.value.sort.
    oclType()
```

Listing 9. Consistency of sorts in value assignments.

```
context StoredAction Inv:
self.saType = Grafcet::
    StoredActionType::event implies(
if self.term.oclIsTypeOf(Terms::
    Variable) then false
else self.term.oclAsType(Terms::
    Operator)->closure(term: Terms::
    Term| term->selectByKind(Terms::
    Operator).subterm)->select(
    operator|operator.oclIsTypeOf(
    Terms::RisingEdge))->size()
+ self.term.oclAsType(Terms::Operator
    )->closure(term: Terms::Term| term
    ->selectByKind(Terms::Operator).
    subterm)->select(operator|operator
    .oclIsTypeOf(Terms::FallingEdge))
    ->size() > 0
endif)
```

Listing 10. Event type of stored actions.

APPENDIX B
GAL CODE OF THE GRAFCET IN FIG. 5

```
gal GeneratedGrafcet {
//System Variable
int situationIsStable = -2;
int transitionSystemState = 0;
int aComponentIsFireable = 0;
int xG1 = 1;

//Input Variable
```

```
int switch_A = 0;
typedef randswitch_A= 0..1;

//Step Variable
int xs1 = 0;
int xs1MarkActivate = 1;
int xs1MarkDeactivate = 0;
int xs2 = 0;
int xs2MarkActivate = 0;
int xs2MarkDeactivate = 0;

//System Transition
transition transitionStateChange_0[
    true] label "checkTransitions" {
transitionSystemState=0;
situationIsStable=0;
}

transition initialTransitionCheck[
    transitionSystemState==-2] {
self."stepEvaluateMarkerss1";
self."stepEvaluateMarkerss2";
self."checkTransitions";
}

transition
    checkIfComponentIsFireable [true
    ] label "
    checkIfComponentIsFireable" {
if((xs1 == 1 && (switch_A)) ){
aComponentIsFireable = 1;
} else{
aComponentIsFireable = 0;
}
}

transition setGrafcetVars [true]
    label "setGrafcetVars" {
xG1 = xs1 + xs2;
if(xG1 > 0){
xG1 = 1;
}
}

//Input Transition
transition inputVariableChanges (
    randswitch_A $switch_A) [
    situationIsStable == 1]{
switch_A = $switch_A;
self."checkTransitions";
}

//Transition Transition
transition checkIfFireable_t1 [true]
     label "t1check" {
if(xs1 == 1 && (switch_A)) {
xs1MarkDeactivate = 1;
xs2MarkActivate = 1;
}
}

//Step Transition
transition stepActivation_s1 [true]
    label "sAs1" {
xs1 = 1;
}

transition stepDeactivation_s1 [true
    ] label "sDs1" {
xs1 = 0;
}

transition stepEvaluateMarkers_s1 [
    true] label "
    stepEvaluateMarkerss1" {
if (xs1MarkActivate == 1 && xs1 ==
    0) {
self."sAs1";
} else{
if xs1MarkActivate == 0 &&
    xs1MarkDeactivate == 1 && xs1 ==
     1) {
self."sDs1";
}
}
xs1MarkActivate = 0;
xs1MarkDeactivate = 0;
}

transition stepActivation_s2 [true]
    label "sAs2" {
xs2 = 1;
}

transition stepDeactivation_s2 [true
    ] label "sDs2" {
xs2 = 0;
}

transition stepEvaluateMarkers_s2 [
    true] label "
    stepEvaluateMarkerss2" {
if (xs2MarkActivate == 1 && xs2 ==
    0) {
self."sAs2";
} else{
if (xs2MarkActivate == 0 &&
    xs2MarkDeactivate == 1 && xs2 ==
     1) {
self."sDs2";
}
}
```

```
xs2MarkActivate = 0;
xs2MarkDeactivate = 0;
}

transition transitionStateChange_0_1
    [transitionSystemState==0] {
situationIsStable = 0;
self."t1check";

self."stepEvaluateMarkersss1";
self."stepEvaluateMarkersss2";

aComponentIsFireable = 0;
self."checkIfComponentIsFireable";
if(aComponentIsFireable==1) {
aComponentIsFireable = 0;
situationIsStable = 0;
transitionSystemState = 0;
} else{
situationIsStable = 1;
transitionSystemState = -1;
}
self."setGrafcetVars";
}
}
```

REFERENCES

[1] R. Julius, T. Trenner, A. Fay, J. Neidig, and X. L. Hoang, "A meta-model based environment for GRAFCET specifications," in *Proc. IEEE Int. Syst. Conf. (SysCon)*, Piscataway, NJ, USA, Apr. 2019, pp. 1–7.

[2] B. Vogel-Heuser, A. Fay, I. Schäefer, and M. Tichy, "Evolution of software in automated production systems: Challenges and research directions," *J. Syst. Softw.*, vol. 110, pp. 54–84, Dec. 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215001818

[3] *GRAFCET Specification Language for Sequential Function Charts*, Standard IEC 60848, 2013.

[4] B. W. Boehm, *Software Engineering Economics* (Advances in Computing Science and Technology Series). Englewood Cliffs, NJ, USA: Prentice-Hall, 1981.

[5] Y. T. Mieg, "From symbolic verification to domain specific languages: Formal languages and automata theory," Ph.D. dissertation, Dept. Laboratoire d'informatique de Paris, Sorbonne Université UPMC, Paris, France, 2016.

[6] R. Julius, T. Trenner, J. Neidig, and A. Fay, "A model-driven approach for transforming GRAFCET specification into PLC code including hierarchical structures," *IFAC-PapersOnLine*, vol. 52, no. 13, pp. 1767–1772, 2019.

[7] *Programmable Controllers—Part 3: Programming Languages*, Standard IEC 61131–3, 2013.

[8] G. Frey, *Design and Formal Analysis of Petri Net Based Logic Control Algorithms*. Aachen, Germany: Shaker, 2002.

[9] R. David and H. Alla, *Discrete, Continuous, Hybrid Petri Nets*, 2nd ed. Heidelberg, Germany: Springer-Verlag, 2010.

[10] X. Weng and L. Litz, "Model checking of signal interpreted Petri nets," in *Proc. IEEE Int. Conf. Syst., Man Cybern. e-Syst. e-Man Cybern. Cyberspace*, Oct. 2001, pp. 2748–2752.

[11] X. Weng and L. Litz, "Verification of logic control design using SIPN and model checking: Methods and case study," in *Proc. Amer. Control Conf. (ACC)*, Jun. 2000, pp. 4072–4076.

[12] P. Pawlewski, *Petri Nets—Manufacturing and Computer Science*, 2012, pp. 177–192.

[13] J.-P. Barros and L. Gomes, "Towards an integrated tool support for the analysis of IOPT nets using the spin model checker," in *Proc. IEEE 31st Int. Symp. Ind. Electron. (ISIE)*, Jun. 2022, pp. 239–244.

[14] S. Klein, X. Weng, G. Frey, J.-J. Lesage, and L. Litz, "Controller design for an FMS using signal interpreted Petri nets and SFC: Validation of both descriptions via model-checking," in *Proc. Amer. Control Conf.*, vol. 5, May 2002, pp. 4141–4146.

[15] I. Azkarate, M. Ayani, J. C. Mugarza, and L. Eciolaza, "Petri net-based semi-compiled code generation for programmable logic controllers," *Appl. Sci.*, vol. 11, no. 15, p. 7161, Aug. 2021, doi: 10.3390/app11157161.

[16] A. El Rhalibi, F. Prunet, and C. Durante, "Analysis of function charts for control systems using Petri nets," in *Proc. IEEE Symp. Emerg. Technol. Factory Automat. (SEIKEN) Symposium)-Novel Disciplines Next Century Proc.*, Piscataway, NJ, USA, Nov. 1994, pp. 365–372.

[17] F. Schumacher, S. Schröck, and A. Fay, "Transforming hierarchical concepts of GRAFCET into a suitable Petri net formalism," *IFAC Proc. Volumes*, vol. 46, no. 9, pp. 295–300, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667016343014

[18] M. Silva and R. Valette, "Petri nets and flexible manufacturing," in *Advances in Petri Nets* (Lecture Notes in Computer Science), vol. 424. G. Rozenberg, Ed. Berlin, Germany: Springer-Verlag, 1990, pp. 374–417.

[19] J. Provost, J.-M. Roussel, and J.-M. Faure, "A formal semantics for Grafcet specifications," in *Proc. IEEE Int. Conf. Autom. Sci. Eng.*, Piscataway, NJ, USA, Aug. 2011, pp. 488–494.

[20] F. Schumacher and A. Fay, "Formal representation of GRAFCET to automatically generate control code," *Control Eng. Pract.*, vol. 33, pp. 84–93, Dec. 2014.

[21] *Systems and Software Engineering—High-Level Petri Nets—Part 2: Transfer Format*, Standard IEC 15909–2, ISO/IEC, 2011.

[22] G. V. Arnold, P. R. Henriques, and J. Fonseca, "A graphical interface based on grafcet for programming industrial robots off-line," in *Proc. 2nd Int. Conf. Informat. Control, Automatics Robot.*, 2005, pp. 113–118. [Online]. Available: https://repositorium.sdum.uminho.pt/handle/1822/3360

[23] G. Nzebop, E. Simeu, and M. Tchuente, "Langage et sémantique des expressions pour la synthèse de modèle Grafcet dans un environnement IDM," in *Proc. Conférence de Recherche en Informatique (CRI Yaoundé)*, 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02558838v4

[24] J. Provost, J.-M. Roussel, and J.-M. Faure, "Translating grafcet specifications into mealy machines for conformance test purposes," *Control Eng. Pract.*, vol. 19, no. 9, pp. 947–957, Sep. 2011.

[25] F. Cassez, "Formal semantics for reactive GRAFCET," *J. Europeen des Systemes Automatises*, vol. 31, pp. 581–603, 1997.

[26] M. Sogbohossou, R. Yehouessi, T. Djara, T. Aballo, and A. Vianou, "SE-LTL model-checking on timed GRAFCETS via ϵ-TPN," *Current J. Appl. Sci. Technol.*, vol. 38, no. 6, pp. 1–12, Dec. 2019.

[27] M. Sogbohossou and A. Vianou, "Formal modeling of grafcets with time Petri nets," *IEEE Trans. Control Syst. Technol.*, vol. 23, no. 5, pp. 1978–1985, Sep. 2015.

[28] M. Sogbohossou and A. Vianou. (Sep. 2020). *Translation of Hierarchical GRAFCET Charts Into Time Petri Nets*. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02934113

[29] I. Tahiri, A. Philippot, V. Carré-Ménétrier, and A. Tajer, "A fault-tolerant and a reconfigurable control framework: Application to a real manufacturing system," *Processes*, vol. 10, no. 7, p. 1266, Jun. 2022. [Online]. Available: https://www.mdpi.com/2227-9717/10/7/1266

[30] G. Frey, "Hierarchical design of logic controllers using signal interpreted Petri nets," *IFAC Proc. Volumes*, vol. 36, no. 6, pp. 361–366, Jun. 2003.

[31] T. Klotz, E. Fordran, B. Straube, and J. Haufe, "Formal verification of UML-modeled machine controls," in *Proc. IEEE Conf. Emerg. Technol. Factory Automat.*, Sep. 2009, 2009, pp. 1–7.

[32] R. Julius, M. Schürenberg, F. Schumacher, and A. Fay, "Transformation of GRAFCET to PLC code including hierarchical structures," *Control Eng. Pract.*, vol. 64, pp. 173–194, Jul. 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0967066117300709

[33] Y. T. Mieg, "Symbolic model-checking using ITS-tools," in *Tools and Algorithms for the Construction and Analysis of Systems* (Lecture Notes in Computer Science), vol. 9035. London, U.K.: Springer, Apr. 2015, pp. 231–237. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02104373

[34] *Meta Object Facility (MOF) Core Specification*, Object Manag. Group, Milford, MA, USA, 2016.

[35] (Mar. 3, 2022). Eclipse Foundation. *Eclipse Modeling Project | The Eclipse Foundation*. [Online]. Available: https://www.eclipse.org/modeling/emf/

[36] *XML Formats for IEC 61131-3: Version 2.01—Official Release*, PLCopen Tech. Committtee, Gorinchem, The Netherlands, 2009. [Online]. Available: https://plcopen.org/legal-notice

[37] (Mar. 3, 2022). *CODESYS*. [Online]. Available: https://www.codesys.com/

ROBIN MROSS received the B.Sc. and M.Sc. degrees in computer science from RWTH Aachen University, Aachen, Germany. He is currently a Research Associate at Informatik 11–Embedded Software, RWTH Aachen University. His research interest includes verification of graphical specification languages.

ARON SCHNAKENBECK received the B.Sc. and M.Sc. degrees in industrial engineering from Universität Hamburg, Germany. He is currently a Research Associate with the Institute of Automation, Helmut Schmidt University, Hamburg, Germany. His research interest includes automatic ways to verify formal specifications of control code design in manufacturing automation using means of static analysis.

MARCUS VÖLKER received the B.Sc. and M.Sc. degrees in computer science from RWTH Aachen University, Aachen, Germany. He is currently a Research Associate at Informatik 11–Embedded Software, RWTH Aachen University, where he leads the Formal Methods for Reactive Systems Research Group. His research interest includes various formal methods to analyze and verify reactive systems.

ALEXANDER FAY (Senior Member, IEEE) is a Full Professor with the Institute of Automation, Helmut Schmidt University, Hamburg, Germany. His main research interests include engineering methods, models, and tools for automated systems, with focus on decentralized and autonomous control. He is a member of IES, CSS, and RAS.

STEFAN KOWALEWSKI is a Professor of embedded software at RWTH Aachen University, Germany. His research interests include model-based and formal analysis and design methods for embedded software with applications in automotive, medical technology, and industry automation.