From IEC 61131-3 Function Block Diagrams to Sequentially Constructive Statecharts

Marcel Christian Werner
University of Kaiserslautern
Kaiserslautern, Germany
marcel.christian.werner@gmail.com

Klaus Schneider
University of Kaiserslautern
Kaiserslautern, Germany
schneider@cs.uni-kl.de

***Abstract*—Function Block Diagrams (FBDs) are widely used for implementing the software of IEC 61131-3 based systems. In general, there is a risk that FBDs used in industry will become more and more complex during their life cycle, while at the same time strict specifications have to be met. On the other hand, a trend towards model-based design with standardized modeling tools can be observed in software engineering. While previous research focuses on translating existing FBDs to formal models for verification purposes, this paper presents two translations from existing FBDs to sequentially constructive statecharts, thus enabling an intuitive functional reuse for a model-based design. Besides a basic translation in the first approach, it is shown in the second approach that it is possible to improve the readability through code refactoring within the synchronous paradigm.**

***Index Terms*—model-driven development, programmable logic devices, software reusability, synchronous languages, system analysis and design**

I. Introduction

The third part of the IEC 61131 standard [1] considers the development of software for industrial real-time systems such as *Programmable Logic Controllers* (PLCs). A language widely used in the field of IEC 61131-3 based PLCs are graphical *Function Block Diagrams* (FBDs) where the elements such as function blocks or variables are connected with graphical lines following a data-flow notation.

Furthermore, over the last two decades, methods have been investigated to enhance software development of IEC 61131-3 based systems using modelling techniques such as the *Unified Modeling Language*[1] (UML) [2]. This trend is reflected in isolated PLC vendors[2] and engineering tools such as *Simulink*[3] [3]. For new projects, a model-based design, e.g., using UML should be used, while for existing plants already using IEC 61131-3 systems, a translation to languages used by model-based designs becomes desirable for at least two reasons: First, the software part of existing plants grows during their life cycle and can become chaotic and complex when it exceeds a certain limit. Second, a further aspect is the need for producing models for a formal verification. Related approaches using synchronous languages [4] follow the idea of a functional reuse similar to a model-based design in software engineering. Picking up the idea of a possible reuse of existing IEC 61131-3 software, this work is motivated by the opportunity to improve the readability of existing FBDs.

This paper presents two translations from IEC 61131-3 FBDs to *Sequentially Constructive Statecharts* (SCCharts) [5], a direct one and another one via the synchronous language *Quartz* [6]. In addition to the translation from FBDs to SCCharts in the first approach, the second one is based on refactoring the code for imperative *Quartz* models, which is intended to improve the readability. Both translation approaches lead to a textual description of the SCCharts, from which different graphical representations can be generated.

The outline of the paper is as follows: After introducing the different software models in Section II, a translation from FBD to SCChart is explained in Section III. In Section IV, a translation from FBD to *Quartz* and a code refactoring is introduced, and in Section V, a translation from *Quartz* to SCChart is described. Related work is discussed in Section VI.

II. Background

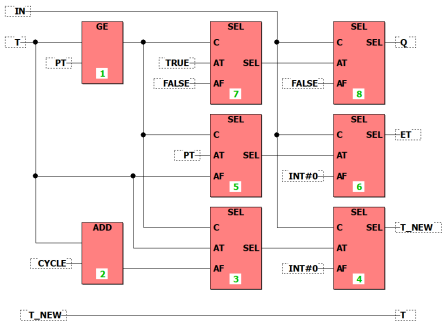
This section introduces the different software models illustrated in Fig. 1 using the example of a generic on-delay timer of an input `IN`. Fig. 1(a) shows a typical graphical IEC 61131-3 FBD, Fig. 1(b) a simple *Quartz* model without declaration section, and Fig. 1(c) a view as graphical data-flow-oriented SCChart, that was automatically generated based on a textual description.

A. The IEC 61131-3 Function Block Diagram

In general, the IEC 61131-3 software model [1] consists of various configuration and language elements. In this approach, we consider cyclically controlled software units, called *Program Organization Units* (POUs). In this model of computation, cyclically, first a so-called process image of the inputs is created, then the logic is executed, and finally the process image of the outputs is updated [7]. In this paper, we are mainly focusing on the modeling of the logic which is why the process images of the physical IEC 61131-3 system are neglected. The language elements in FBDs are represented as shown in Fig. 1(a), which are connected by graphical

[1]https://www.omg.org/spec/UML/2.5.1
[2]https://store.codesys.com/codesys-uml.html
[3]https://de.mathworks.com/products/simulink.html

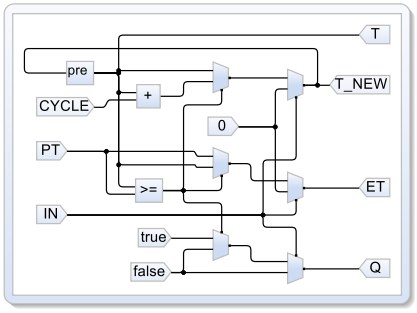
978-1-6654-7332-3/22/$31.00 ©2022 IEEE



(a) IEC 61131-3 FBD (manually implemented)

```
loop{
  if(IN){
    if(T >= PT){
      Q = true; ET = PT; T_NEW = T;}
    else{
      Q = false; ET = T;
      T_NEW = T + CYCLE; }}
  else{
      Q = false; ET = 0; T_NEW = 0;}
  next(T) = T_NEW; pause;
}
```

(b) Quartz Model (manually implemented)



(c) Data-flow-oriented SCChart (automatically generated view)

Fig. 1. Different software models for a generic on-delay timer

signal lines. Typically, not only variables are connected to each other by lines, but also the ports of graphical function blocks [1]. The function blocks can be user-implemented POUs or function blocks included via libraries. The execution of the function blocks is sequential and is identified with a number in the blocks by some PLC manufacturers.

B. The Synchronous Software Model

Synchronous languages are based on the general principle of perfect synchrony [6]. This means that a program can typically be divided into so-called reaction or macro steps which define different logical points of time in the program. Synchronous languages are becoming more and more attractive for the design and verification of reactive real-time systems and can be generally divided into imperative, data-flow, and graphical languages [6], [8], [9]. This approach focuses on the imperative language *Quartz* and the graphical language SCCharts.

1) Quartz and Averest: Averest[4] is a framework for the specification, verification and implementation of reactive systems. It expects the system to be described as a program in the textual language *Quartz* [6] as shown in Fig. 1(b). *Quartz* is derived from *Esterel* [10] and comes with typical statements for imperative languages like assignments, selection or bounded loops. Furthermore, *Quartz* allows the implementation of synchronous and asynchronous concurrency of threads, the explicit modeling of non-determinism and delayed assignments, among other things. Consumption of time must be explicitly programmed with special statements. An example for this is the `pause` statement. When the *Quartz* program is executed, the control flow stops at the `pause` statement and is resumed at this point in the next macro step, provided there is no surrounding suspension or abortion.

2) Sequentially Constructive Charts and the KIELER Framework: KIELER[5] is an open-source framework as part of a research project to improve graphical model-based design of complex systems. This framework can be used to generate graphical SCCharts, among other things. SCCharts was initially introduced for the specification of safety-critical reactive systems [5]. In addition to various visualization options, we will focus in the following on data-flow-oriented SCChart models and state-oriented SCChart models where the respective graphical representation is based on textual descriptions.

III. From IEC 61131-3 FBDs to SCCharts

This section describes the first approach introduced in this paper to translate existing IEC 61131-3 FBDs to SCCharts, whose idea is shown in Fig. 2. In order to translate FBDs to SCCharts automatically, they are first translated to suitable equivalent textual IEC 61131-3 *Structured Text* (ST) models [1], [11], assuming that no explicit function execution control is used within the FBDs, which would be realized via additional Boolean inputs at the used function blocks [7]. Some PLC development environments such as *Beremiz*[6] support auto-translation from FBD to ST.

However, it must be considered that the backward translation strategy [12] is followed. This algorithm starts from the last executed function block of an output to the first executed function block since FBDs typically consist of a series of function blocks that are connected sequentially, hierarchically, and compositionally [12]. Accordingly, for each output of a given FBD such as the FBD of the generic on-delay timer, the sequential execution of all affected function blocks is expressed as an equation. Due to the fact that the graphical function blocks can be interpreted as block calls, this results in a representation as ST model which also uses the operators as function block calls. For both variables, `T_NEW` and `T`, given in the FBD in Fig. 1(a), this results in the two shown equations within the IEC 61131-3 paradigm. In comparison to the backward strategy, the forward translation strategy [12] starts from the first executed function block and the results are stored in additional temporary variables which are used as inputs in subsequent statements. This would prevent a nesting of multiple function blocks, but leads to the fact that for each graphical data-flow signal line an additional variable is needed which can have a negative impact on the readability depending on the generated view. Independent of the chosen translation strategy, the sequentially constructive model of computation

[4]http://www.averest.org/

[5]https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Overview

[6]https://beremiz.org/doc

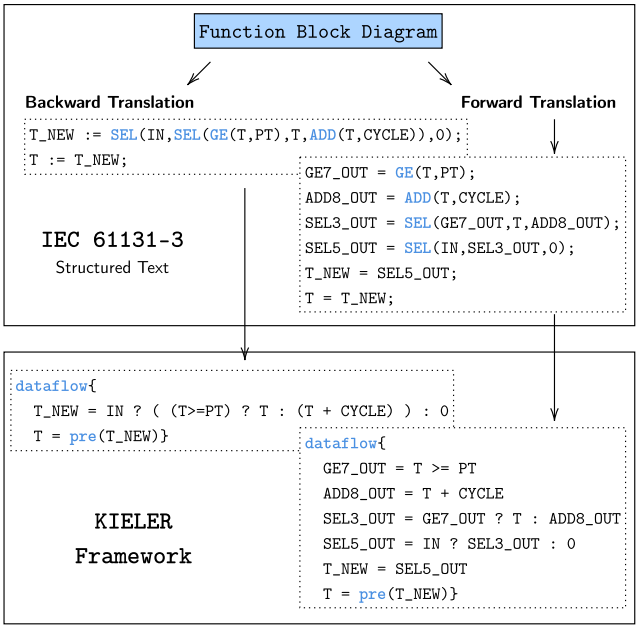
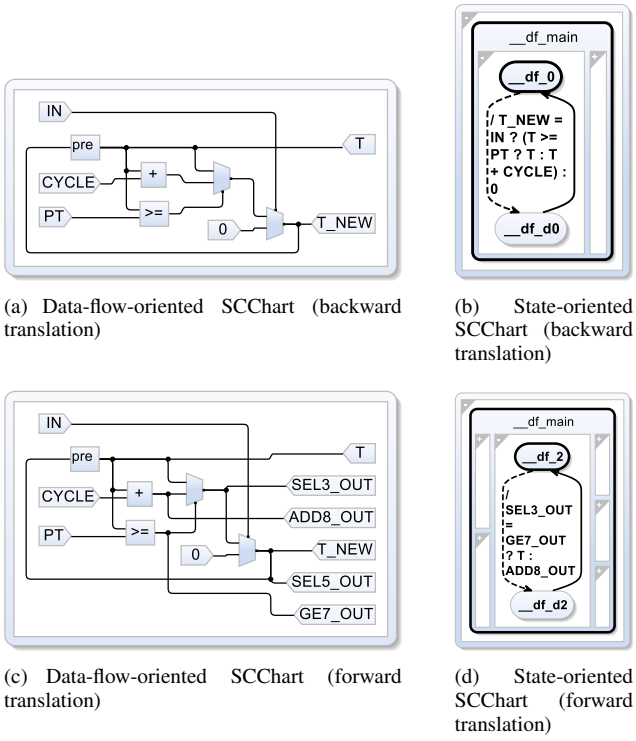


Fig. 2. Workflow to translate existing FBDs to SCCharts



(a) Data-flow-oriented SCChart (backward translation)

(b) State-oriented SCChart (backward translation)

(c) Data-flow-oriented SCChart (forward translation)

(d) State-oriented SCChart (forward translation)

Fig. 3. Impact of the translation strategy on the resulting graphical SCCharts related to the translation from FBD to ST

of SCCharts has to be considered. A good example of this is given by the two variables `T` and `T_NEW` in Fig. 2. Since the assignment to `T` is sequential after the assignment to `T_NEW`, a `pre(T)` statement is omitted in the assignment to `T_NEW`.

The difference in the subsequent representation is shown in Fig. 3. Fig. 3(a) and Fig. 3(b) show graphical SCCharts automatically generated by the *KIELER* framework as a result of a backward translation strategy and Fig. 3(c) and Fig. 3(d) show SCCharts as a result of a forward translation strategy. In both state-oriented SCCharts, only one variable is shown, but the relationship between the textual description in Fig. 2 and the transition is obvious.

There are two approaches for further translation of the resulting IEC 61131-3 ST model into the synchronous paradigm using *KIELER*: (1) The used function blocks are supported as operators or (2) are defined in a separate SCChart and instantiated in this place. We assume that the operators are supported by *KIELER* and focus on the first approach. Inline refactoring can then be used to generate the textual SCChart description as illustrated in Fig. 2 by the variables `T` and `T_NEW`, and thus generate the graphical data-flow-oriented SCChart introduced in Fig. 1(c).

A. Readability of FBD-based SCChart

Overall, it can be stated that this translation can be used to generate graphical data-flow-oriented SCCharts which have almost the same complexity as the underlying IEC 61131-3 FBDs which is confirmed by the graphical models in Fig. 1(a) and Fig. 3(a). Only delayed assignments, i.e., those that are only to be valid in the following cycle, must be handled specially due to the synchronous paradigm. In our opinion, the possible representation as equivalent state-oriented SCChart can significantly degrade the readability in case it is intended to be reused as a graphical model for software engineering purposes, since the nesting of the function blocks is directly reflected in the transitions. For complex algorithms, we expect that the logic can be difficult to understand. For instance, comparing the equations for both variables, `T` and `T_NEW`, it is obvious that the transitions have the same level of complexity as the equations and therefore the same depth of nesting.

B. Correctness of the FBD-to-SCChart Translation

The correctness of the translation can be evaluated by showing that the I/O behavior of both models, the original IEC 61131-3 FBD and the resulting SCChart, is the same.

Conjecture 1. *The resulting SCChart always shows the same I/O behavior as the underlying IEC 61131-3 FBD, if the logic was translated via a backward translation to an equivalent ST model and then to SCChart, assuming that all operators are supported by the* KIELER *framework.*

Both models, the IEC 61131-3 FBD and SCChart, contain the same interface. The I/O behavior is tested for the IEC 61131-3 standard function blocks [1] `SR`, `RS`, `R_TRIG`, `F_TRIG`, `CTU`, `CTD`, `CTUD`, `TON`, `TOF`, and `TP`. In this context, the `TIME()` function used in the standard function blocks is replaced as shown in the presented generic on-delay timer in Fig. 1 by a generic cycle-dependent counter. In addition, based on the listed standard function blocks, 10 different applications are tested to check the applicability to more complicated algorithms. The I/O behavior is compared

via simulation inside the *KIELER* framework and IEC 61131-3 development environment. This does not represent a general proof of the correctness, but it confirms the applicability for algorithms limited to the supported *KIELER* operators.

IV. From IEC 61131-3 FBDs to Quartz Models

Picking up on the first step in the translation from FBDs to ST models in Fig. 2, the following two sections describe a second approach to translate existing FBDs to SCCharts, where this section focuses on translating FBDs to *Quartz*. Fig. 4 illustrates how a possible subsequent translation to *Quartz* models can be realized: In a first step, a code refactoring of the ST model is performed to get a ST model that contains core operators typical for imperative languages. This step is trivial and does not require further details at this point. In a second step, the resulted ST model is translated to *Quartz*, considering the synchronous paradigm as illustrated with the delayed assignment to `T` using the `next` operator and the additional `pause` statement.

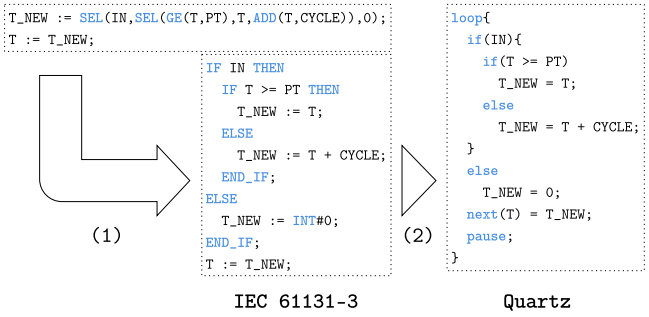
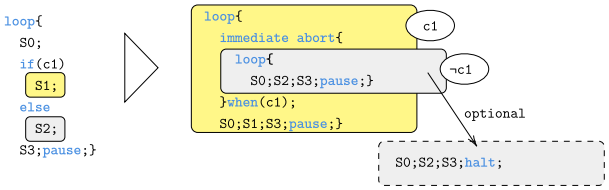
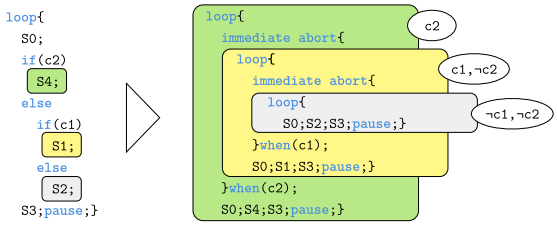


Fig. 4. Workflow to translate existing FBDs to Quartz models

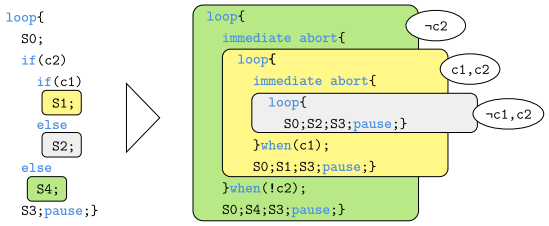
To improve the readability of the final SCChart, additional *Quartz* code refactoring is performed within the synchronous paradigm. The basic strategy is illustrated in Fig. 5(a). The `if-else` condition is replaced by a `immediate abort` statement taking into account possible immediate statements before and after, i.e., in aborted and not aborted branches. In both scenarios, `loop S` iterations are executed of which the inner one can be replaced by a `halt` statement depending on the time dependency of the included statements. Nesting of multiple `if-else` conditions is common in many applications which is why Fig. 5(b) illustrates the case where the `else` branch contains another condition. This results in a hierarchy of `immediate abort` statements. The hierarchy follows the sequential execution order of the original `if-else` condition. This means that the last `else` branch represents the innermost iteration and the first `if` condition the outermost iteration. In this case, it is also possible to replace one of the inner `loop S` iterations with a `halt` statement, where in addition to the time-independence of the statements, it must be considered that this means that it is no longer possible to switch to an inner iteration and can only be terminated by a surrounding `abort` environment. In contrast, nesting inside the `if` branch cannot be replaced in the same way, because the inner `if-else` condition would have a higher priority in the evaluation. This would change the semantics. In such cases, the logic of the Boolean condition must be inverted accordingly to move the statements to the `else` branch which is shown in Fig. 5(c).



(a) Code refactoring of condition statement



(b) Code refactoring of nested condition statement within *else* branch



(c) Code refactoring of nested condition statement within *if* branch

Fig. 5. Quartz code refactoring approach

A. Correctness of the Quartz Code Refactoring

The correctness of the code refactoring can be evaluated by showing that the I/O behavior of both models, the original *Quartz* model and the refactored *Quartz* model, is the same.

Conjecture 2. *The* `if-else` *condition can be refactored by a* `immediate abort` *statement, considering possible immediate statements before and after it.*

Both *Quartz* models, before and after code refactoring, contain at least one statement in the respective `if` and `else` branch which is verified by forcing the Boolean condition accordingly at simulation time within the *Averest* framework and checking the state transitions. For all three demonstrated scenarios, the correctness of the respective code refactoring approach is validated. It is obvious that the approach may only be applied as long as the conditions are independent of `S0`, because otherwise potential causality errors may result.

V. From Quartz Models to SCCharts

Following an approach to synthesizing safe state machines from *Esterel* [13], we define a set of transformation rules for existing *Quartz* statements to allow synthesizing SCCharts. We limit the definition of the rules to a subset of the *Quartz* core statements [6] and then apply them to translate the generic on-delay timer application.

1) **Transformation Rule 1 (nothing):** The dashed arrow in Fig. 6 represents an immediate transition from state `S1` to `S2` and is checked as soon as `S1` is entered. An immediate transition can lead to situations where the parent state is entered and left in the same tick as in this case, since the `nothing` statement neither changes variables nor stops the control flow and terminates immediately [6].

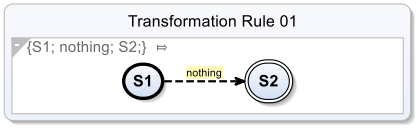


Fig. 6. SCChart of Quartz sequence {S1; nothing; S2;}

2) **Transformation Rule 2 (await(a)):** The solid arrow in Fig. 7 represents a transition from state `S1` to `S2` that becomes active in the next tick after its parent state `S1` has been entered. The `await(a)` instruction depends on a Boolean condition to terminate [6]. That means the statement waits at `S1` until the Boolean condition `a` is true, and then terminates.

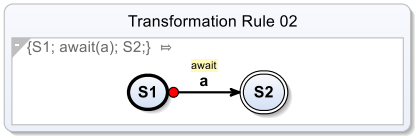


Fig. 7. SCChart of Quartz sequence {S1; await(a); S2;}

3) **Transformation Rule 3 (pause):** The execution of a `pause` statement consumes one logical unit of time and leads to a stop of the control flow and current macro step. In the next macro step, the control is resumed from this point. Therefore, the `pause` is never instantaneous and it follows, among other things, that it behaves equivalently to `await(true)` [6]. Fig. 8 illustrates both equivalent representations.
4) **Transformation Rule 4 (sequence):** A sequence, as illustrated in Fig. 9, is a sequential execution of different statements after termination. Switching between statements does not consume time. If the individual statements terminate instantaneously, the sequence also terminates instantaneously [6]. The change of sequences containing other sequences, such as `S2`, is indicated by the green triangle.
5) **Transformation Rule 5 (synchronous concurrency):** The application of synchronous concurrency of different

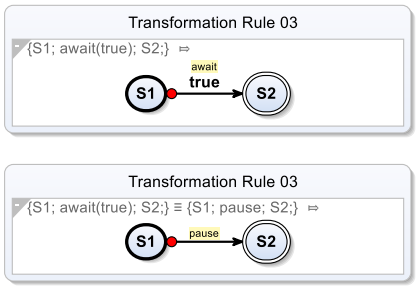


Fig. 8. SCChart of Quartz sequences {S1; await(true); S2;} and {S1; pause; S2;}

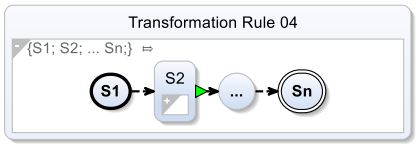


Fig. 9. SCChart of Quartz sequence {S1; S2; ... Sn;}

statements illustrated in Fig. 10 is a common use case for synchronous languages. At the macro steps, the parallel instructions are synchronized and can interact. Any `abort` statements affect all parallel statements and the statements terminate as soon as the last of the parallel statements terminates [6].

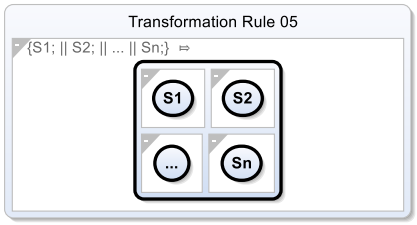


Fig. 10. SCChart of Quartz sequence {S1; || S2; || ... || Sn;}

6) **Transformation Rule 6 (do $\mathcal{S}$ while(a)):** The `do S while(a)` statement represents an iteration and is executed as illustrated in Fig. 11. A statement `S1` is executed and the Boolean condition `a` is ignored. After `S1` has been terminated, the Boolean condition `a` is checked. If `a` is true, then `do S1 while(a)` is executed again. Otherwise, the iteration terminates [6].

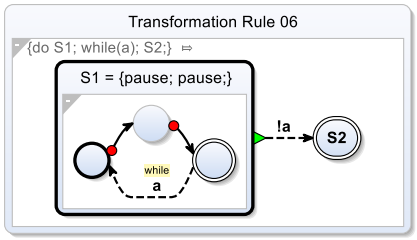


Fig. 11. SCChart of Quartz sequence {do S1; while(a); S2;}

7) **Transformation Rule 7 (loop $\mathcal{S}$):** The `loop S` statement is a special case of the `do S while(a)` iteration

statement, illustrated in Fig. 12. More precisely, it is an infinite loop for which we can assume that the termination condition is never true [6].

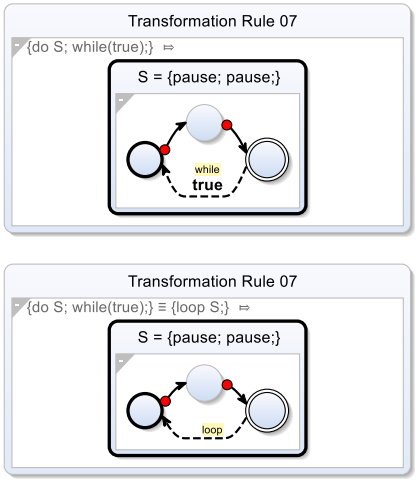


Fig. 12. SCChart of Quartz sequences {do S; while(true);} and {loop S;}

8) **Transformation Rule 8 (halt):** As illustrated in Fig. 13, the `halt` statement represents an infinite loop executing a `pause` statement. This means that the `halt` statement never terminates [6].

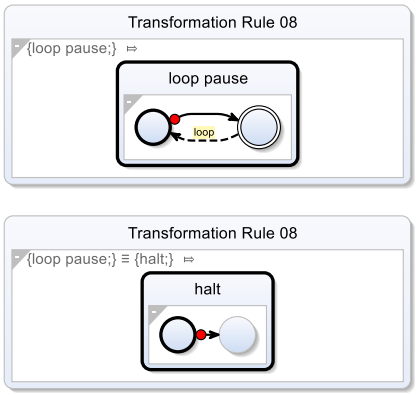


Fig. 13. SCChart of Quartz sequences {loop pause;} and {halt;}

9) **Transformation Rule 9 (immediate assignment):** The immediate assignment `a = b`, as illustrated in Fig. 14, instantaneously modifies the value of `a` so that `a` has the same value as `b` [6].

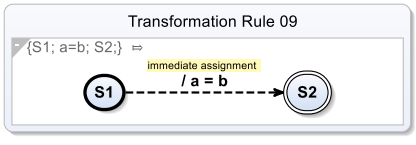


Fig. 14. SCChart of Quartz sequence {S1; a=b; S2;}

10) **Transformation Rule 10 (delayed assignment):** The delayed assignment is executed instantaneously, after a delay of one logical instant. This is illustrated in Fig. 15.

In the upper use case, `a` is instantaneously set to `1` and in the next macro step, `b` is set to `1`, although `a` is set to `2` in the same macro step. In contrast, the statement `halt` does not terminate in the lower use case, so the immediate assignment `a = 2` is unreachable, but `b` is set to `1` [6].

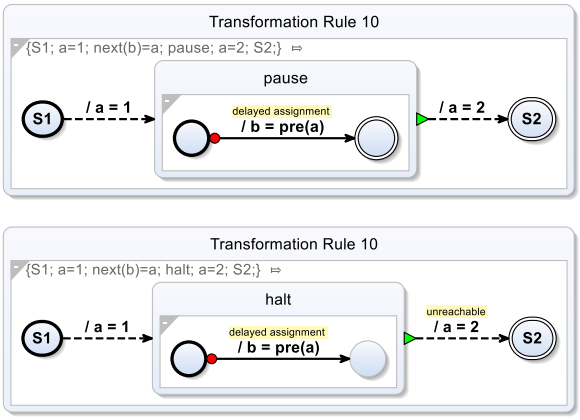


Fig. 15. SCChart of Quartz sequences {S1; a=1; next(b)=a; pause; a=2; S2;} and {S1; a=1; next(b)=a; halt; a=2; S2;}

11) **Transformation Rule 11 (abort $\mathcal{S}$ when(a)):** The `abort S when(a)` statement in Fig. 16 evaluates `a` in each macro step while `S` is running. In case `a` is true, then the execution of `S` will be aborted. Otherwise, the execution of `S` will not be disturbed [6].

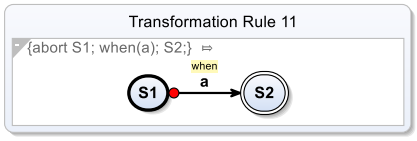


Fig. 16. SCChart of Quartz sequence {abort S1; when(a); S2;}

12) **Transformation Rule 12 (immediate abort $\mathcal{S}$ when(a)):** The `immediate abort S when(a)` statement, in contrast to the `abort S when(a)` statement, already checks the condition before `S` is running, i.e., any immediate statements in this macro step are not executed [6]. As illustrated in Fig. 17, the representation is very similar considering the semantics of dashed and solid transitions explained in the previous transformation rules.

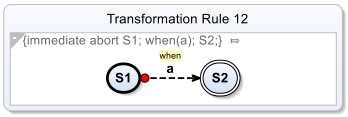


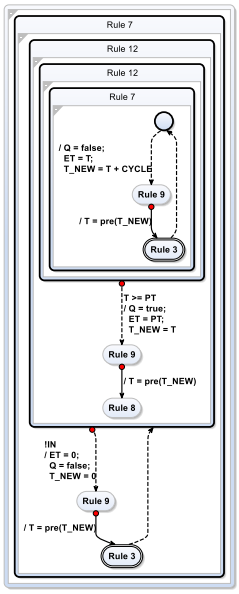
Fig. 17. SCChart of Quartz sequence {immediate abort S1; when(a); S2;}

Thus, all transformation rules are defined to represent the generic on-delay *Quartz* model in Fig. 18(a) as SCChart in Fig. 18(b). This may lead to unnecessary hierarchical states which can be eliminated by applying general optimization

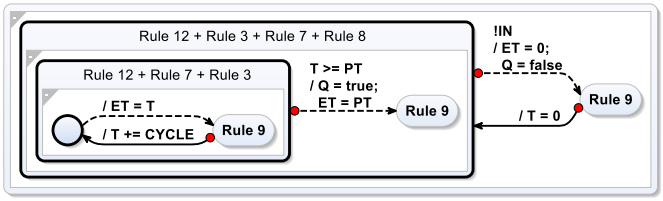
strategies [13]. In addition, instantaneous transitions can be combined, redundant actions such as `Q = false` can be eliminated, and, taking into account any side effects, auxiliary variables such as `T_NEW` may be resolved. In this example, this leads to the optimized SCChart in Fig. 18(c).

```
loop{ //Rule 7
  immediate abort{ //Rule 12
    immediate abort{ //Rule 12
      loop{ //Rule 7
        Q = false; //Rule 9
        ET = T; //Rule 9
        T_NEW = T + CYCLE; //Rule 9
        next(T) = T_NEW; //Rule 10
        pause; //Rule 3
      }
    }when(T >= PT); //Rule 12
    Q = true; //Rule 9
    ET = PT; //Rule 9
    T_NEW = T; //Rule 9
    next(T) = T_NEW; //Rule 10
    halt; //Rule 8
  }when(!IN); //Rule 12
  ET = 0; //Rule 9
  Q = false; //Rule 9
  T_NEW = 0; //Rule 9
  next(T) = T_NEW; //Rule 10
  pause; //Rule 3
}
```

(a) Quartz Model



(b) Applied transformation rules



(c) Optimized SCChart

Fig. 18. Applying the transformation rules to the generic on-delay timer with subsequent optimization options

A. Readability of Quartz-based SCChart

Compared to the underlying IEC 61131-3 FBD, we see an opportunity to improve the readability through the translation into *Quartz* and then to an optimized SCChart. This is due to the fact that, unlike in the first approach, there is no nesting of the sequentially executed function blocks within the state-oriented SCChart transitions. Furthermore, since a notation very similar to UML can be achieved, we assume that this representation may be an accepted alternative view for users who are not familiar with data-flow notation.

B. Correctness of the FBD-to-Quartz-to-SCChart Translation

The correctness of the second translation approach can be evaluated in the same way as the first approach by checking whether both models, the FBD-based *Quartz* model and the resulting SCChart, have the same I/O behavior.

Conjecture 3. *The FBD-based* Quartz *model and the resulting SCChart have the same I/O behavior when the presented transformation rules for translating* Quartz *to SCChart are applied.*

The translation from IEC 61131-3 FBD to *Quartz* is assumed to be correct, so the test is reduced to the translation from *Quartz* to SCChart. The same I/O behavior is expected for the listed test cases in the first approach. In this context, the same behavior is tested via simulation using the *KIELER* framework and the *Quartz* framework. Furthermore, the transformation rules of the *Quartz* statements are based on the general definitions of the *Quartz* language [6] and the equivalence of different *Quartz* statements, respectively. The correctness is given by the fact that the rules are based on each other and equivalent *Quartz* statements, such as `await(true)` and `pause`, or `while(true)` and `loop S` are tested using the *KIELER* framework. This does not represent a general proof of correctness, but shows the correctness of the transformation rules presented in this paper and confirms the use of available general optimization strategies.

VI. Related Work

Due to the intuitive data-flow notation, the implementation of IEC 61131-3 based systems with FBDs is widely used in the field and can be found in various engineering tools (like *Simulink*, *Labview*[7] and others). At the same time, it can be observed that tool vendors offer more and more features for even more abstract modeling that is easy to understand by users from different domains (like *SysML*[8], UML, *Stateflow*[9]). The challenge with FBDs in real-world applications is, among other things, the increasing complexity that arises with growing program size. In addition, verification proofs are required for safety-critical applications which is why the translation of existing IEC 61131-3 programs to formal models for verification purposes dominates the research [14]–[20]. Furthermore, some approaches follow the strategy of a functional reuse of FBDs in IEC 61499 based systems. It has been shown [21] that this standard has no valuable benefit for improving the IEC 61131-3 development process. In another approach [4], the authors pursue the goal of reusing existing IEC 61131-3 FBDs as a synchronous *Quartz* model in the context of a model-based design by splitting the function blocks into concurrent threads to counteract the sequential execution order. Furthermore, after an extension of the *KIELER* framework over the last years [22], a semantically equivalent control flow oriented counterpart to the data-flow representation as SCChart can be generated. Since the complexity of the transitions in the control-flow representation relates to the sequential execution of the blocks in the FBD, we see the risk that the representation can quickly become difficult to understand. In another approach [23] related to the *KIELER* framework, the translation from the synchronous

[7] https://www.ni.com/de-de/shop/labview.html
[8] https://sysml.org/
[9] https://de.mathworks.com/products/stateflow.html

language *Blech*[10] to SCCharts is introduced. The authors try not to capture all the semantics of the original code, but only the underlying state structure. The goal is an automatically generated documentation with a higher level of abstraction. As far as we know, our approach is one of the few or even the only one, which is motivated in contrast by not only obtaining a higher level of abstraction for documentation purposes, but still enabling a formal verification and a functional reuse of existing FBDs as a better readable model-based design in software engineering.

VII. Conclusions and Future Work

This paper presents two translations for existing IEC 61131-3 FBDs into synchronous graphical SCCharts described by their textual description and evaluates the impact on readability. The first approach is based on the idea of translating existing FBDs following the strategy of a backward translation to equivalent textual ST models from which the textual input format for data-flow-oriented SCCharts can be derived. Using the example of the generic on-delay timer application, it is explained that a graphical data-flow-oriented SCChart at the same level of readability as the underlying FBD can be created. The equivalent representation as a state-oriented SCChart has the risk that the complexity of the nested function block calls may lead to a worse readability of the original FBD. The second approach is based on the idea of translating the translated ST models into imperative synchronous *Quartz* models and performing code refactoring within the synchronous paradigm to nest the function blocks and logic hierarchically. Depending on the FBD complexity, this allows deriving more readable state-oriented SCCharts compared to the first approach, which represents an alternative representation of the original FBD. Thus, a beneficial functional reuse in software engineering is possible.

This research activity will further investigate additional *Quartz* code refactoring approaches to further improve the *Quartz* model as well as the readability of the resulting SCCharts. As an example, we will analyze whether the instantaneous statements before and after the nested `if-else` conditions can be eliminated by including synchronous concurrency statements.

References

[1] DIN Deutsches Institut für Normung e.V., "Programmable controllers – part 3: Programming languages (IEC 61131-3:2013); german version en 61131-3:2013," Berlin, 2014.

[2] D. Witsch and B. Vogel-Heuser, "Automatische Codegenerierung aus der UML für die IEC 61131-3," in *Eingebettete Systeme*, P. Holleczek and B. Vogel-Heuser, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 9–18.

[3] G. Bayrak, F. Abrishamchian, and B. Vogel-Heuser, "Effiziente Steuerungsprogrammierung durch automatische Modelltransformation von Matlab/Simulink/Stateflow nach IEC 61131-3," *Automatisierungstechnische Praxis (atp)*, vol. 50, no. 12, pp. 49–55, 2008.

[4] M. C. Werner and K. Schneider, *Translation of Continuous Function Charts to Imperative Synchronous Quartz Programs*. New York, NY, USA: Association for Computing Machinery, 2021, p. 104–110.

[5] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O'Brien, "SCCharts: Sequentially Constructive Statecharts for Safety-Critical Applications," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 372–383.

[6] K. Schneider, *The synchronous programming language Quartz*, 2nd ed. Kaiserslautern: Department of Computer Science, University of Kaiserslautern, 2010.

[7] R. W. Lewis, *Programming Industrial Control Systems Using IEC 1131-3 (IEE Control Engineering Series)*. Institution of Engineering and Technology, 1998.

[8] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Guernic, and R. Simone, "The Synchronous Languages 12 Years Later," in *Proceedings of the IEEE*, vol. 91, 02 2003, pp. 64 – 83.

[9] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," in *Proceedings of the IEEE*, vol. 79, no. 9, 1991, pp. 1270–1282.

[10] G. Berry and G. Gonthier, "The Esterel synchronous programming language: design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[11] D. Darvas, I. Majzik, and E. Blanco Viñuela, "Generic representation of PLC programming languages for formal verification," in *Proceedings of the 23rd PhD Mini-Symposium*. Zenodo, Feb. 2016, pp. 6–9.

[12] J. Yoo, E. S. Kim, and J. S. Lee, "A behavior-preserving translation from FBD design to C implementation for reactor protection system software," *Nuclear Engineering and Technology*, vol. 45, no. 4, pp. 489–504, 2013.

[13] S. Prochnow, C. Traulsen, and R. von Hanxleden, "Synthesizing Safe State Machines from Esterel," in *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, ser. LCTES '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 113–124.

[14] S. Rösch, S. Ulewicz, J. Provost, and B. Vogel-Heuser, "Review of model-based testing approaches in production automation and adjacent domains—current challenges and research gaps," *Journal of Software Engineering and Applications*, vol. 08, pp. 499–519, 01 2015.

[15] T. Ovatman, A. Aral, D. Polat, and A. O. Ünver, "An overview of model checking practices on verification of PLC software," *Software and Systems Modeling*, vol. 15, no. 4, pp. 937–960, 2016.

[16] D. Darvas, I. Majzik, and E. Blanco Viñuela, "Formal Verification of Safety PLC Based Control Software," in *Proceedings of the 12th International Conference on Integrated Formal Methods - Volume 9681*, ser. IFM 2016. Springer Berlin Heidelberg, 2016, p. 508–522.

[17] H. Barbosa and D. Déharbe, "Formal Verification of PLC Programs Using the B Method," in *Abstract State Machines, Alloy, B, VDM, and Z*, J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 353–356.

[18] N. Völker and B. J. Krämer, "Automated Verification of Function Block Based Industrial Control Systems," *Sci. Comput. Program.*, vol. 42, pp. 101–113, 01 2002.

[19] V. Gourcuff, O. De Smet, and J. M. Faure, "Efficient Representation for Formal Verification of PLC Programs," in *2006 8th International Workshop on Discrete Event Systems*. IEEE, 2006, pp. 182–187.

[20] B. Fernandez Adiego, D. Darvas, E. B. Vinuela, J.-C. Tournier, S. Bliudze, J. O. Blech, and V. M. Gonzalez Suarez, "Applying Model Checking to Industrial-Sized PLC Programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.

[21] K. Thramboulidis and G. Frey, "Towards a Model-Driven IEC 61131-Based Development Process in Industrial Automation," *Journal of Software Engineering and Applications*, vol. 04, pp. 217–226, 2011.

[22] L. Grimm, S. Smyth, A. Schulz-Rosengarten, R. von Hanxleden, and M. Pouzet, "From Lustre to Graphical Models and SCCharts," in *2020 Forum for Specification and Design Languages (FDL)*, 2020, pp. 1–8.

[23] D. Lucas, A. Schulz-Rosengarten, R. von Hanxleden, F. Gretz, and F. Grosch, "Extracting Mode Diagrams from Blech Code," in *2021 Forum on specification and Design Languages (FDL)*, 2021, pp. 01–08.

[10] https://www.blech-lang.org/