Journal of Software Engineering and Applications, 2017, 10, 457-481
http://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Evaluation of Selected Control Programming Languages for Process Engineers by Means of Cognitive Effectiveness and Dimensions

Gülden Bayrak, Felix Ocker, Birgit Vogel-Heuser

Institute of Automation and Information Systems, Technical University of Munich, Munich, Germany
Email: gueldenbayrak@hotmail.de, felix.ocker@tum.de, vogel-heuser@tum.de

How to cite this paper: Bayrak, G., Ocker, F. and Vogel-Heuser, B. (2017) Evaluation of Selected Control Programming Languages for Process Engineers by Means of Cognitive Effectiveness and Dimensions. *Journal of Software Engineering and Applications*, **10**, 457-481.
https://doi.org/10.4236/jsea.2017.105026

Received: April 7, 2017
Accepted: May 23, 2017
Published: May 26, 2017

Abstract

Different programming languages can be used for discrete, abstract and process-oriented programming. Depending on the application, there exist additional requirements, which are not fulfilled by every programming language. Flexible programming and maintainability are especially important requirements for process engineers. In this paper, the programming languages Activity Diagram, State Chart Diagram and Sequential Function Chart are compared and evaluated with regard to these requirements. This evaluation is based on the principles of cognitive effectiveness and cognitive dimensions. The aim of this paper is to identify the programming language suited best for controlling sequential processes, e.g. thermomechanical or batch processes.

Keywords

Cognition, Cognitive Science, Graphical User Interfaces, Human Computer Interaction, Human Factors, Programmable Logic Controllers, Process Control

1. Introduction

Certain products can only be produced by a combination of discrete manufacturing and process technologies. The manufacturing process of these products integrates both characteristics of the process and of the manufacturing technology. Companies offering such products are therefore subject to the challenges of both technologies. These include "increasing efficiency, effectiveness and quality in design of software engineering […] to shorten engineering and start-up and ease maintenance" [1]. Thermomechanical and batch processes usually consist of a number of sub-processes in a unique sequential order [2]. These sub-processes generally affect the resulting product and change the product state to "non-commutative". Companies usually rely on Programmable Logic Controllers (PLCs) to control (experimental) machines that implement these processes [3].

A desired product in this domain is defined by the individual process steps, their parameters and the steps' sequence. In industry, the correlations between these influencing factors and the resulting product properties are often analyzed experimentally. As process engineers have the appropriate expertise, they develop and modify the sequence as well as the parameters of the individual steps in each experiment. Such modifications are realized on code level (PLC). However, adaptation of sequences, parameters and timing constraints in PLC code is error-prone and time consuming for process engineers, who are often PLC programming novices [4]. This holds true for a wide variety of lab size machines to develop new process technologies, e.g. batch processes or even a timber press. Batch processes in chemical production are characterized by the existence of a recipe for each product, which specifies the sequence of process steps and the corresponding control of valves.

In order to support especially process engineers, who have the necessary expert knowledge but lack in programming skills, code should be changeable. This greatly facilitates the engineering of processes based on experiments.

Another important factor is the reduction of development costs. Maintainability costs increase continuously and already account for up to 90% of costs throughout the software product lifecycle, according to estimates [5]. Therefore, maintainability has to be especially considered.

Flexible programming very well indicates changeability, which refers to the possibility of changing or adjusting a program according to different needs [6]. Flexible programming can be broken down into modifiability [6] and modularity [7] [8]. Harrison *et al.* [9] subdivided maintainability into modifiability and understandability. All in all, an adequate way of engineering processes should fulfill the requirements of flexible programming (R1) and those of maintainability (R2). These two overlap in the need for modifiability (R1.1). In addition to this, R1 also includes modularity (R1.2), while R2 demands understandability (R2.1).

A hierarchical supervision (hierarchical scheme) programming is also important because it is of relatively low complexity. Hence, its understandability is higher than that of a conventional nonhierarchical one [10]. It was shown that understandability and maintainability in graphical programming languages are better than in textual programming languages [11]-[17]. Therefore, we do not consider textual programming languages in this paper. However, there also exist limitations to understandability in graphical programming languages. Miller's Law [18] states that an average human can hold up to 7 ± 2 objects in working memory.

The main contribution of this publication is the analysis of the programming languages Activity Diagram (AD), State Chart Diagram (SC) and Sequential Function Chart (SFC) regarding the concepts of flexible programming and maintainability. These consist of the requirements modifiability, understandability and modularity. Since the requirements refer to the visual syntax of the programming languages, the methods for analyzing the structural complexity as well as the cognitive efficiency of the notations are researched first and applied afterwards.

The remainder of this contribution is structured as follows. Within Section 2, we present an application example and introduce the discrete programming languages AD, SC and SFC. This section concludes with a comparison of the programming languages' visual syntax and metaclasses. Section 3 discusses the state of the research on methods for comparison of programming languages in terms of the requirements modifiability (R1.1), modularity (R1.2) and understandability (R2.1). It also includes an overview of existing work and their results. The correlation between the requirements and descriptive methods for comparison of dissimilar languages is discussed in Section 4. Section 4 also includes the comparison and evaluation of AD, SC and SFC in terms of modifiability, modularity and understandability. It ends with a summary of the evaluation's results. Finally, Section 5 concludes the paper and gives an outlook on future research.

2. Application Example in AD, SC and SFC

PLCs provide various graphical control-flow-oriented programming languages for the implementation of discrete processes. IEC 61131-3 [19] includes three graphical programming languages: Sequential Function Chart (SFC), Ladder Diagram (LD) and Function Block Diagram (FBD). Previous work of our group shows that the Activity Diagram (AD) and State Chart Diagram (SC) are the most appropriate languages for process engineers [20] [21]. The plcML-Editor [20], which includes AD and SC, provides model-driven graphical modeling for PLCs. Both the visual syntax and the meta-model of the programming language SC are based on Witsch [22] and the SFC programming language is based on CoDeSys V3 [23].

In order to prevent gaps in the information flow of the design process, it is desirable to select notations which have high suitability values in many, or even in all, phases of the design process of production systems [24]. In an expert survey, the different suitability potentials of notations were assessed according to the phases of the design process [24]. It was shown that SC are of great importance in all phases, while the SFC seems to be the most important description language in late phases. The flowchart, which is similar to an AD, and the SC have similar suitability values in the implementation and commissioning phase. We conclude that the programming languages AD, SC and SFC should be further investigated regarding their appropriateness for thermomechanical and batch processes.

2.1. Process Steps in the Application Example

As an application example, we use the production of a shaft with flanges with graded properties in a metal forming plant. The flange forming consists of the four process steps heating, transforming, forming and cooling. This is exemplary for most of the processes in the CRC TTR30, independent from the resources used [25]. In this case, resources refer to technical compositions/machines. “Heating”, e.g., may be realized in one prototype (metal forming) by inductive heating and in another (friction pressure) by friction heating. Each possible combination of the process steps in metal forming (cp. **Figure 1**) leads to specific product properties. On the one hand, the discrete process sequence “preheat, transport, heat, form and cool” delivers a product A (shaft with flanges) with the property functionally graded A1 (distribution of hardness at the edge of the workpiece). The process sequence “parallel heat and cool form and finally cool” on the other hand delivers a product A with the property functionally graded A2 (distribution of hardness in the interior of the workpiece).

Another possible sequential process that consists of the same four process steps is a plastic pressing (cp. **Figure 1**, Product B). We consider the workpiece to already be put into the press, so it starts with a preheating of the hardboards. Thereafter, the workpiece is heated and formed in parallel. After this, depending on which mode has been selected, either the forming process is completed and the workpiece is cooled (for example by water- or airflow), or the workpiece is cooled during the forming process.

In the following, we compare the visual syntax and the metamodels of AD, SC and SFC at the example of the third process sequence. We realized AD and SC with the UML-Editor [20] and SFC with CoDeSys 3.0, which is a PLC Programming environment [23] (cp. **Figures 2(a)-(c)**).

2.2. Activity Diagram (AD)

In **Figure 2(a)**, the AD for the process of Product B (cp. **Figure 1**) is presented.

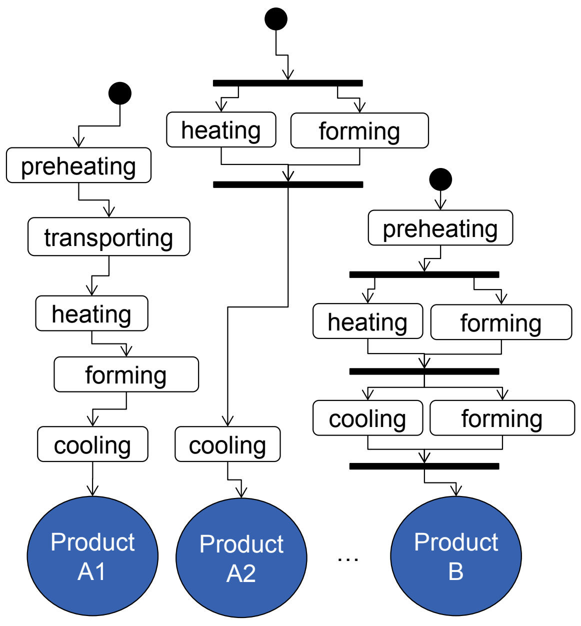
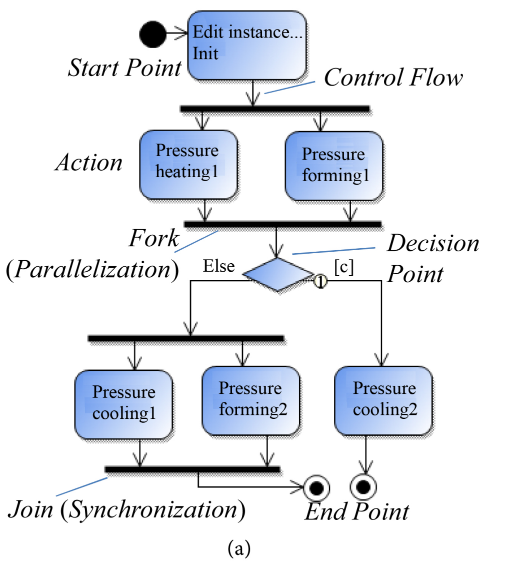
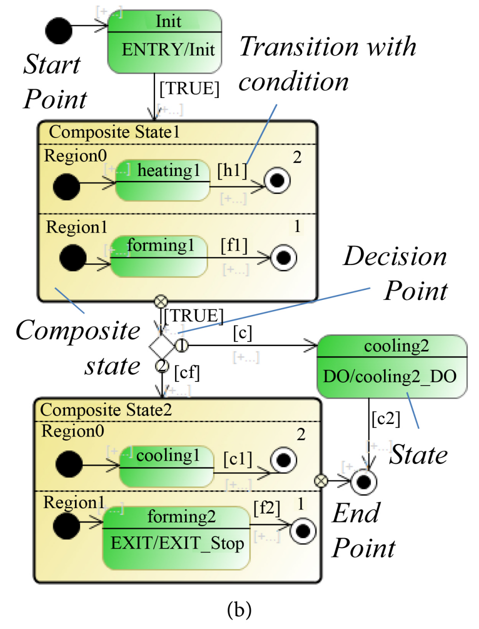


Figure 1. Different sequences of the basic process steps heating, transporting, forming and cooling resulting in products A (shaft with flanges)or B with different product characteristics (A1, A2,⋯).





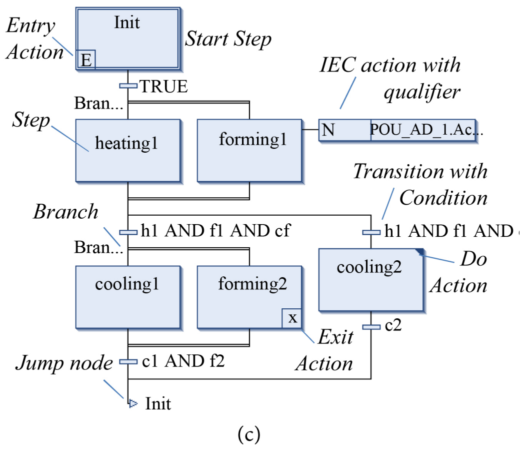


Figure 2. (a) Simple steps of plastic pressing in AD. (b) Simple steps of plastic pressing in SC. (c) Simple steps of plastic pressing in SFC.

The program starts (Start Point) with an initialization Action such as preheating of the hardboards. The parallelization of processes, e.g. heating and forming, is realized by Fork and Join. The decision between the nodes cooling and forming and only cooling is represented by a Decision Point. Finally, the program ends with two End Points.

This action is performed until a fixed end-condition, defined by the process engineers, holds true. The end condition is a parameter of the action and cannot be learned from the control flow. In AD, only the outgoing transitions of decision nodes have guard conditions. An action corresponds to the real duration of a process in terms of a technical system, e.g. the time until a work piece has reached a certain temperature. This corresponds to the comprehension of a process engineer.

2.3. State Chart Diagram (SC)

States are the main elements of the SC. From the perspective of a process engineer, states in a programmed technical system are either static (e.g., "heated", "formed") or dynamic (e.g. "heating", "forming"). Figure 2(b) shows the dynamic states of a plastic pressing process. States can have Entry, Do, and Exit Actions. The elements Start-, Choice- and End Point have the same function/semantic construct as their equivalents in AD (Start-, Decision-, and End Point). Additionally, there is a static conditional branch Junction, which also has a merge and decision function. Parallelization can be realized by Fork/Join (analogously to AD) or with an additional composite State (depicted in Figure 2(b)). Within SC, processes run pseudo parallel, *i.e.* that the modeled parallel behavior is realized sequentially with priority numbers. In contrast to an AD, the (end-) conditions are integrated in the Transitions.

2.4. Sequential Function Chart (SFC)

Figure 2(c) shows the sequence of an exemplary plastic pressing process in SFC. SFC is suitable for depicting sequential behaviors of a control system. The control sequences are time- and event-driven. Unlike AD and SC, which begin with a Start Point, all SFC programs must begin with an Init Step. This is the first step to be activated whenever an SFC is started. The Init Step (depicted by a rectangle with a double border line) has a similar visual syntax like a "normal" Step (rectangle with thin border line). A Step can be associated with an IEC action with qualifier. These are shown to the right of the corresponding step in a box that consists of two parts. SFC has no End Point, but the Jump Node may be used for designing the end of the process step (jump to the Init Step) or for jumping to another step in the program. Branches and parallelization can be realized with a visual syntax (horizontal double lines). As in the case of SC, SFC can also have Steps with Entry, Do and Exit actions.

2.5. Comparison of the Metaclasses of AD, SC and SFC

Within this section, we show the similarities and differences between AD, SC, and SFC on the metaclass level. To compare the different programming languages, it is necessary to analyze the metamodels of AD, SC and SFC. Table 1 shows an overview of the concrete metaclasses of the different metamodels. Generic metaclasses can be created by comparing the individual metamodels' various metaclasses including their internal relationships. This means that different generic semantic constructs are realized by certain metaclasses of different programming languages. Exemplary, the generic metaclass Flow is mapped to the metaclass Control Flow in AD but to the metaclass Transition in SC and SFC. A Branching is realized in AD with Decision and Merge Node, in SC with Pseudo State (Choice, Junction) and in SFC with a Transition Flow (cp. Table 1). Some metaclasses cannot be mapped to generic metaclasses, though. These are Object Flow, Pin and Central Buffer of AD, Deep History and Exit Point Pseudo State of SC and Action Association of SFC.

3. State of the Art

In the area of PLC programming different benchmark studies exist, that address the comparison of LD and Petri Net (PN) design methods with the criteria understandability, complexity and flexibility [26] [27]. The comparison of 11 high-level system design methods, among others the state transition diagram and state machine, is shown in [28]. Cao *et al.* [29] presented case studies of visual and formal modeling and design. All these studies have different focuses and a comparison of the visual syntax between AD, SC and SFC does not exist. In [30], an approach is presented that combines the advantages of SC and SFC in sequential statecharts (SSC) by using a function block encapsulation. The focus is put on performance, though, and explicitly not on legibility and usability of graphical notations. Lukman *et al.* [31] introduced a model-driven engineering approach for process control based on the newly developed domain-specific modeling language ProcGraph. They don't focus on maintainability by process engineers. We selected AD, SC and SFC, though, due to their prevalence in the domain [17] [32] and their ease of use, especially for process engineers, who are often programming novices.

Table 1. Comparison of the metaclasses of AD, SC and SFC.

Metaclasses generic	Metaclasses AD	Metaclasses SC		Metaclasses SFC
Process Step	Action	State		Step, Action
Flow	Control Flow	Transition		
Branching	Decision Node	Pseudo State	Choice	
	Merge Node		Junction	Transition
Parallelization	Join Node		Join	
	Fork Node		Fork	
Start	Initial Node		Initial	Step, Action
End	Activity Final Node	Final State		Transition (Jump)
Structure	Activity Partition	Composite State		-

The following research of methods for comparison of programming languages focused on those that consider the requirements modifiability (R1.1), modularity (R1.2) and understandability (R2.1).

There exist different descriptive methods for the comparison of dissimilar languages of these sub-requirements. The two prescriptive decision theories most applied are cognitive dimensions and cognitive effectiveness (cf. **Table 2**). Hereby, cognitive dimensions (CD) is a task-specific broad-brush framework for assessing almost any kind of cognitive artifact addressing primarily non-specialists [33]. In contrast, cognitive effectiveness (CE) aims at evaluating the "speed, ease, and accuracy with which a representation can be processed by the human mind" [34]. Thus, the focus of cognitive effectiveness is rather put on understandability.

Green [37] and Britton and Jones [36] present a dimension analysis of programming languages. The dimension analysis facilitates comparing dissimilar languages and also helps to identify the relationship between them. Roast *et al.* [38] focus on the dimension analysis for program modification. Moody [35] defines the principles for the design of effective visual notations with a cognitive effectiveness analysis. These principles are intended to support the comparison of notations in terms of understandability. These methods (cognitive dimensions and effectiveness) are used for evaluation purposes of programming and modeling languages [39] [40] [41] and [42] [43]. **Table 3** shows an overview of existing work with these methods and their results especially for AD and SC separately.

Barji *et al.* [41] assess graphic expressiveness and intuitive comprehension of IEC 61499 function blocks, UML SC and Petri net based CNet. The notations are evaluated by specific criteria. The criterion visual modularity is partially fulfilled by SC and the criterion visual hierarchy is completely fulfilled by SC. The level of abstraction of SC is classified as low. The cognitive analysis of Figl *et al.* [40] is limited to the routing elements, wherein the routing elements of AD were evaluated as easiest to understand. Moody and van Hillegersberg [39] evaluated and compared the various UML diagrams (i.a. AD) with each other. From a visual representation view AD was rated better than the other diagrams of UML.

Table 2. Methods of comparing programming languages.

Reference	Evaluation of	in terms of	with
Moody (2009) [35]	Visual notations in software engineering	Understandability	Cognitive Effectiveness
Green and Petre (1996) [33]	Computer programs and visual notations	Understandability	Cognitive Dimensions
Britton and Jones (1999) [36]	Software specification languages	Ease of understandability	Cognitive Dimensions
Green (1989) [37]	Computer/programming languages	Understandability	Cognitive Dimensions
Roast *et al.* (2000) [38]	Visual and a textual programming language: LabVIEW and pseudo-Code	Program modification	Cognitive Dimension: repetitively viscous

Table 3. Comparison of programming languages in terms of ease of understanding/maintainable results for UML.

Reference	Evaluation of	in terms of	with	Result
Moody *et al.* (2009) [39]	the Visual Syntax of UML 2.0	understandability	Cognitive Effectiveness	"AD is the best from a visual representation viewpoint."
Figl *et al.* (2010) [40]	Process modeling languages (only routing Elements of EPC, UML (*i.e.* AD), YAWL, and BMPN)	creating and understanding models	Cognitive Effectiveness	"AD has the most scores."
Barji *et al.* (2006) [41]	IEC 61499 function blocks, UML SC and Petri net based CNet	graphic expressiveness and intuitive comprehension	Design criteria for real-time control systems	"The availability of modularization and reuse in SC is partially. SC fulfills the composition/hierarchy. The abstraction level of SC is low."

In the above mentioned studies, the UML diagrams (UML 2.0) of OMG were analyzed, which have a different syntax than the plcML programming languages [22]. In this work, the plcML programming languages AD and SC [22] are investigated which have adapted and selected syntax and semantics for the PLC environment.

Bauer [44] investigated the semantic differences between SFC and SC and showed that it is not possible to translate one language into another. But this does not exclude that a discrete process can be realized in two programming languages (cf. section 2.5). The visual syntax differences between SFC and SC for the above mentioned requirements are not addressed. The ontological analysis as defined by the Bunge-Wand-Weber Model [45], which is also used for the evaluation of modeling languages, cannot distinguish between two notations which have the same semantics but different syntax [46]. Our focus is to compare the different programming languages, which have similar semantics (generic semantic) but different syntax.

In summary, for the comparison of AD, SC and SFC academia lacks approaches which investigate visual syntax for programming mechanical processes in terms of the requirements R1 and R2.

4. Concept of Analysis

In order to evaluate the programming languages AD, SC and SFC, the relationships between requirements, cognitive dimensions and effectiveness, and their criteria are investigated. Table 4 shows the correlation between the requirements R1 and R2 as well as the cognitive methods.

The requirements can be evaluated by the corresponding principles/criteria of cognitive dimensions or/and effectiveness. For assessing the programming languages' modularity (R1.2), both their degree of abstraction (CD) and their complexity management (CE) have to be considered. It was shown, that modularity is very much related to abstraction [47], but the criterion complexity management additionally analyzes the visual modularity. The requirement modifiability (R1.1) can be evaluated by the cognitive dimension viscosity. However, concerning understandability (R2.1), both cognitive dimensions and cognitive effectiveness offer different criteria to be taken into account. When comparing the relevant properties of cognitive dimensions [48] with the principles of cognitive effectiveness [34] concerning understandability, it becomes evident, that the prior are a subset of the latter (cf. **Table 5**). Britton and Jones [36] thereby define the properties of cognitive dimensions as properties of languages that contribute to ease of understanding of representations. Thus, for the requirement understandability, it is sufficient to examine the principles of cognitive effectiveness.

4.1. Modifiability Analysis (R1.1)

The modifiability of the three programming languages is analyzed with the cognitive dimension viscosity. Viscosity is defined as resistance to change, *i.e.* the effort for making changes in a program [38]. Thereby, repetitive viscosity and

Table 4. Relationship between requirements and cognitive dimensions and effectiveness.

Requirement	Requirement (sub)	Evaluation with Cognitive…	Relevant principles/criteria
R1: flexible programming	R1.2: modularity	Dimensions	abstraction, (can increase hidden dependencies and visibility)
		Effectiveness	complexity management
	R1.1: modifiability	Dimensions	viscosity
R2: maintainability of the program	R2.1: understandability	Dimensions	redundant recording, consistency, visibility, closeness of mapping, hidden dependencies, abstraction
		Effectiveness	semiotic clarity, perceptual discriminability, semantic transparency, complexity management, cognitive integration, visual expressiveness, dual coding, graphic economy, cognitive fit

Table 5. Comparison of CE and CD concerning understandability.

Criteria of Cognitive Effectiveness [34]	Criteria of Cognitive Dimensions [48]
Semiotic clarity, graphic economy	Consistency (hard mental operations)
Semiotic clarity, perceptual discriminability	Consistency (visibility)
Cognitive fit	Closeness of mapping (role expressiveness)
Perceptual discriminability, visual expressiveness, dual coding	Hidden dependencies (visibility)
Complexity management	Abstraction gradient, hidden dependencies

knock-on viscosity can be distinguished. Repetitive viscosity considers "the nature of the change which takes place in terms of pre- and post-conditions and the user actions which can achieve that change" [38]. This can be easily quantified through the number of necessary further actions. Knock-on viscosity on the other hand "concerns the manner in which an artifact can limit or restrict how a goal is reached" [38]. This is harder to examine, though, and is partially reflected by the number of necessary actions. Therefore, we limit the viscosity evaluation within this paper to the metric of repetitive viscosity: the length of the minimal action sequence for a program modification. The shorter this action sequence, the easier the realization of the associated program modification [38]. We investigate the change in a program with the following program modification actions, which are typical requirements of process engineers for thermomechanical processes. Thereby we assume that in all programs a Process-Library (Activity-, State-, and Step-Library for thermomechanical processes (like heating, forming, etc.)) exists:

1) Creation of the different process sequences (greenfield, total of 18 different process sequences)

2) All possible maximal process sequences in a program with decision nodes (includes 9 process steps, 2 parallel process steps and 6 decision nodes)

3) Exchange the process step with another process step (minimal change in a process sequence)

4) Add new process step(s)/parallelization/branching (minimal change in a process sequence)

5) Changes for a new experimental test (includes 14 different changes)

The creation of the different process sequences 1) contains for each generated program the activities: insert nodes (start-, stop, process step-, decision-node, etc.), connect the elements, insert, if necessary, a transition condition. The "maximal process sequence" 2) includes all possible program sequences in one program by using decision nodes. The "exchange [of] the process step with another process step" 3) includes the activities delete and insert a process step and connect the new process step with others. The activities of "add new element" 4) are delete edges, move elements, insert nodes, connect and, if necessary, insert a transition condition. Finally, the "changes for a new experimental test" 5) consist of the activities delete edges, delete nodes, move nodes, insert nodes, connect nodes, and, if necessary, adjust/insert transition conditions. The result of this program modification is presented in **Table 6**. It is striking that by the modifications (1 - 5) in SC more actions are needed as in AD and SFC. This can be explained by the fact that by every change, the transitions need to be adjusted in the appropriate places. Although this is also the case in SFC, SFC allows the simple adding of elements. That means if one step is added in some position then the in- and outgoing edges are connected automatically with the preceding and following steps. Thereby the number of activities is reduced. In case of AD, this is not the case, so that the number of actions is minimally larger than in the SFC. Moreover, in SFC not every step has an outgoing edge with a transition

Table 6. Results of the modifiability analysis.

Programming languages	(1) Creation of the different process sequences	(2) All possible maximal process sequences in a program with decision nodes	(3) Exchange the process step with another process step	(4) Add new process step(s)/parallelization/branching	(5) Changes for a new experimental test	Average of the number of the activities
AD	15.89	48	3	7.5	7.36	16.35
SC	23.22	63	4.33	10.67	10.5	22.34
SFC	15.44	44	3	5.33	4.07	14.37

condition. For example, all conditions forn-parallel process steps are combined within one transition condition in SFC (see section 4.3). Instead of amending various transition conditions (more actions), just one complex transition condition (one action) must be adjusted. This has the disadvantage that the transition condition is more complex than simple transitions and thereby prevents the modularization. This point is discussed in the next section.

In summary, the SFC has the minimal actions by program modification and fulfills best the requirement modifiability with AD.

4.2. Modularity Analysis (R1.2)

In this section, the criteria of complexity management and abstraction are analyzed to show the visual degree of modularization of programming languages. (cf. Table 4). These criteria are complexity management by modularity and hierarchy and abstraction. Abstraction describes the amount of structure inherence in the languages.

4.2.1. Complexity Management

Modularization and hierarchy are two important mechanisms that reduce program complexity by means of reuse and, thus, can increase the understandability [46] [49]. To make a statement about the degree of modularization, the programming languages must be studied for the visual modularizing capability. The visual modularization depends on the visual syntax of programming languages. That means that the process steps, transitions and all further information that belong together must be encapsulated for modular programming.

For example, the transition "temperature is reached" is associated with the process step heat, the transition "destination reached" is associated with the process step transport etc. In order to support the way of modular thinking, transitions, which are dependent from the previous process step, are encapsulated in these.

Which programming languages support visual modularization is discussed on basis of Figure 3. This figure shows the representation of the individual process

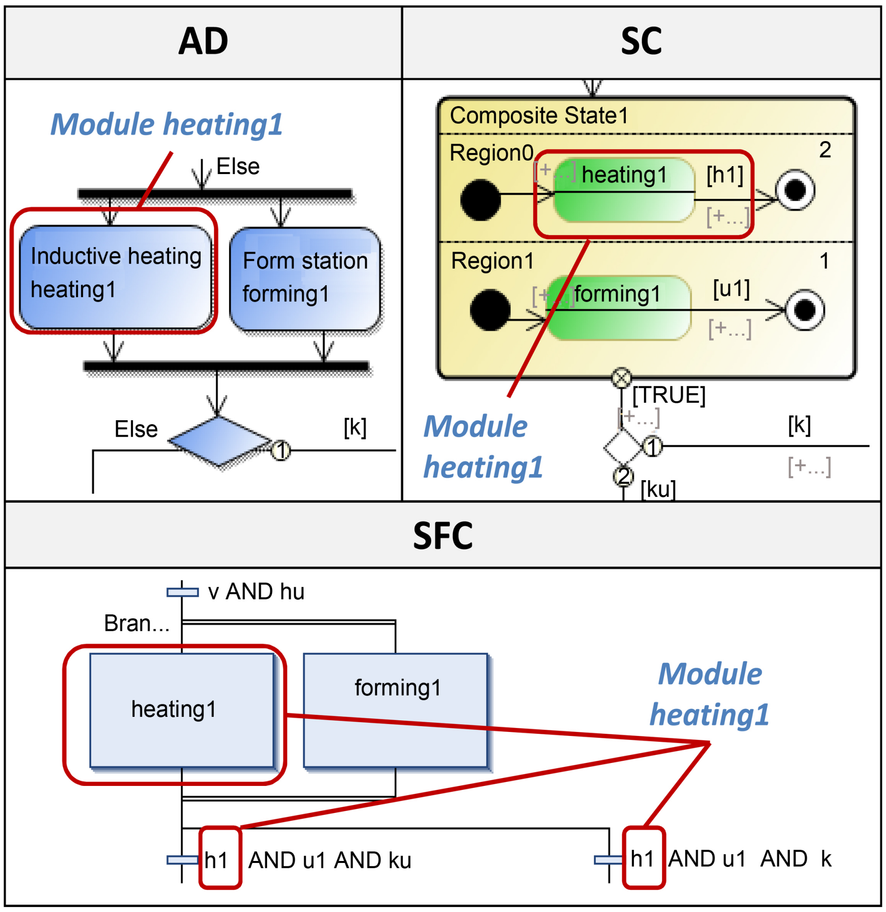


Figure 3. Realization of modularity in different programming languages.

steps with the corresponding switching conditions. Module heating1 in SC is composed of a state heating1 and a transition h1. However, the module heating1 in SFC consists a step heating1 and two transitions ("h1 AND u1 AND ku" and "h1 AND u1 AND k"). Since in SFC no (visual) decision nodes exist, the transition ku has to be inserted together with the process step dependent transition h1 into one transition condition (see SFC part in Figure 3: "h1 AND u1 AND ku"). Therefore, transition conditions will be more complex.

This means that if a change is necessary (for example exchange of one process step) the dependent transition conditions must also be changed. This requires high effort, is time consuming and the combination of transition conditions of several steps inhibits the modularization (cf. Figure 4).

The process step-end conditions in AD are integrated in the activity step/process step respectively. After completion of the parallel steps, decision nodes decide which step should come next. AD is the only programming language which encapsulates the end condition in their activity, and thus, fulfills the visual modularization. SC and SFC do not fulfill this property because of their distinction between states/steps and transitions.

The second mechanism is the hierarchy which reduces the complexity of the program. In a hierarchy the programs can be represented on different levels of

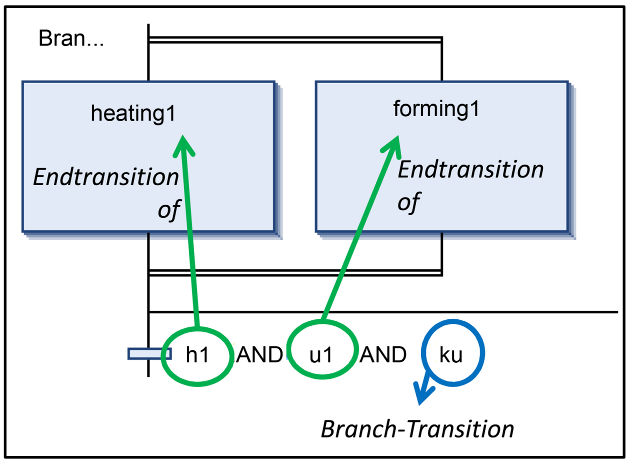


Figure 4. Different step dependent-transition conditions inside of one transition in SFC. h1: heating completed, u1: forming completed.

detail, so that the complexity remains manageable at each level. All three languages support the top-down understanding and hence the hierarchy.

4.2.2. Abstraction

Decomposition and abstraction of a language are important properties [36]. The decomposition can be divided into horizontal and vertical decomposition [43]. While the horizontal decomposition takes place at the same level of abstraction (for example composite State in SC), in the vertical cutting, the model is decomposed into sub-hierarchical models. The horizontal decomposition is often referred to as a hierarchical decomposition. AD provides the horizontal decomposition through the mechanism swimlanes and SC through the mechanisms of composite state and orthogonal state. SFC does not support the horizontal decomposition. All programming languages, however, fulfill the vertical decomposition. Abstraction is on the one side a useful dimension for program modification [37] and can reduce viscosity [50]. Although the degree of abstraction is dependent from the application domain [51], the degree of visual abstraction of a programming language may be determined by the hidden dependencies [50]. With hidden dependencies, important relationships are not visible in a model, such as the process step dependent transition (cf. **Figure 3**). Compared to SC and SFC, AD has a high level of abstraction.

4.3. Understandability Analysis (R2.1)

This section includes the cognitive effectiveness analysis in terms of understandability. The nine principles for designing cognitively effective visual notations are defined as follows [34]:

- Semiotic Clarity: There should be a 1:1 correspondence between semantic constructs and graphical symbols.
- Perceptual Discriminability: Different symbols should be clearly distinguishable from each other.

- Semantic Transparency: Use visual representations whose appearance suggests their meaning.
- Complexity Management: Include explicit mechanisms for dealing with complexity.
- Cognitive Integration: Include explicit mechanisms to support integration of information from different diagrams.
- Visual Expressiveness: Use the full range and capacities of visual variables.
- Dual Coding: Use text to complement graphics.
- Graphic Economy: The number of different graphical symbols should be cognitively manageable.
- Cognitive Fit: Use different visual dialects for different tasks and audiences.

Complexity Management has already been discussed in the previous section. The remaining eight principles are investigated for AD, SC, and SFC in the following.

4.3.1. Semiotic Clarity

The principle of semiotic clarity is defined by the 1:1 mapping of semantic constructs to the visual syntax (graphical symbols). The list of semantic constructs is defined by the list of concrete metaclasses in the AD, SC, and SFC metamodels [22]. Thereby, concrete metaclasses are not enumeration and not abstract metaclasses [34]. The metaclasses of AD, SC and SFC were already introduced in Table 1. If semantic constructs cannot be assigned to a symbol, four types of anomalies can occur [34] [43]:

- Symbol deficit: a construct is not represented by any symbol
- Symbol redundancy: a single construct is represented by multiple symbols
- Symbol overload: a single symbol is used to represent multiple constructs
- Symbol excess: a symbol does not represent any construct

We calculate the net symbol balance (nbSymbols) which gives a statement about the actual number of symbols in consideration of the four types of anomalies. This is calculated as follows [52] (1):

$$\text{nbSymbols} = n_{\text{Constructs}} + n_{\text{Excess}} - n_{\text{Deficit}} + n_{\text{Redundancy}} - n_{\text{Overload}} \quad (1)$$

The language, which has the least percentage of overall anomaly types (symbol excess, deficit, redundancy, and overload), fulfills the principle of semiotic clarity best. Table 7 shows the result of the evaluation of AD, SC and SFC. In summary AD has 12 constructs and 9 symbols. With 8.3% symbol redundancy (metaclass “Activity Partition”) and 25% symbol overload (metaclasses “Control and Object Flow”, “Decision and Merge Node”, and “Join and Fork Node”), AD has a net symbol balance (nbSymbols) of 10. Concerning SC, we distinguish two variants. SC includes a “Pseudo State” that has a kind “Pseudo State Kind” which is an enumeration class with a different visual syntax. If we consider the enumeration (cp. column SC-A in Table 7), SC has 10 constructs and 11 symbols. With 20.0% symbol redundancy and 10.0% symbol overload, SC has an nbSymbols of 12. In contrast, according to the definition of Moody [34], we do not calculate the enumeration as a semantic construct in variant B. In that case, SC has 5

Table 7. Results of the comparison of semiotic clarity.

	AD	SC		SFC	
		A	B	A	B
Constructs	12	10	5	6	6
Symbols	9	11	11	18	10
Excess in %	0	0	20	0	0
Deficit in %	0	0	0	0	0
Redundancy in %	8.3	20	20	50	33.3
Overload in %	25	10	20	0	0
Total anomaly in %	33.3	30	60	50	33.3
Net Symbol Balance	10	11	6	9	8

constructs and 11 symbols. Variant B of SC has 20% symbol excess, 20% symbol redundancy and 20% symbol overload, resulting in an nbSymbols of 6. The SFC metamodel [22] has the concrete metaclasses "Step", "Action Association", "Transition", and "Action". "Action" has attributes (entry, do, exit) causing variations in the corresponding symbol, which corresponds to different semantic constructs. The evaluation shows that SFC has 6 constructs. Considering the different labels of "Action Qualifier" as symbols of their own leads to a total of 18 symbols (cp. column A in SFC). With 50% symbol redundancy SFC has an nbSymbols of 9. Variant B of SFC, which disregards the different labels, has 33.3% symbol redundancy and a net symbol balance of 8.

Variant A of SC exhibits the lowest anomaly (30.0%) and fulfills the principle of semiotic clarity best. AD and SFC (variant B) are almost as good with an anomaly of 33.3%.

4.3.2. Perceptual Discriminability

Perceptual Discriminability is defined by the ease and accuracy to distinguish between different symbols. This is a precondition for correct interpretation of diagrams [53]. For this we calculate in this section the visual distance and the visual-semantic congruence of the visual syntax of AD, SC and SFC and compare these.

1) Visual Distance

There exist eight distinct visual variables (vVar) by Bertin [54], which are divided into categories planar (horizontal and vertical position of the symbol-(x, y) coordinate) and retinal variables (shape, color, size, brightness, orientation/direction, texture/grain). A symbol can consist of combination of these visual variables (e.g. shape = circle and color = blue), which can generate an infinite number of symbols by use of different visual variables. The differentiation is measured by the number of visual variables (where they differ) and the size of this difference. The visual distance between symbols is calculated in according to [34] (2):

$$\begin{aligned}\text{visual distance} = \text{Symbol1}\{\text{vVar1}, \text{vVar2}, \cdots, \text{vVarn}\} \\ \cap \text{Symbol2}\{\text{vVar1}, \text{vVar2}, \cdots, \text{vVarn}\}\end{aligned} \quad (2)$$

The greater the visual distance between symbols, the faster and more accurately they can be recognized. The differentiation also depends on the user's expertise. For example, novices have a higher requirement to distinctness than experts.

For example, the visual distance between an activity and a decision node is equal to one, because it only differs by the form. The colors of the symbols are the same depending on the tool.

The comparison of all symbols within an AD yields the result that AD has two symbols with visual distance of zero (merge and decision node, fork and join node). Overall, the symbols can be clearly distinguished from each other since 38% of the symbols in AD have a visual distance of four.

Compared to other programming languages (AD and SC) SFC has with a visual distance of one, the highest percentage of 35. This means that in SFC 35% of all symbols have only a visual distance of one and therefore the differentiation is rather poor. However, 53% of symbols in SFC have a visual distance of three. SC has a marginally better differentiation as AD, because 43% of all symbols in SC have a visual distance of 4. AD has only 38%.

2) Visual-Semantic Congruence

In general, the visual distance should be equal between two symbols and the semantic distance between the corresponding two constructs (metaclasses). Constructs with different semantics should have clearly distinguishable symbols and similar constructs should have symbols as similar as possible [55]. The semantic distance is defined by the shortest path between the metaclasses of the inheritance hierarchy [34]. In AD, for example each semantic distance between a Control Flow and Object Flow and Decision Node and Merge is two. The difference between visual and semantic distance is between one and two. In SC and SFC, however, the metric cannot be used directly for the calculation of the semantic distance because SC has the enumeration-class "Pseudostate" which has a plurality of different "attributes", each having a separate symbol. According to [34], this results in a semantic distance of zero. This is not possible, though, because the semantic distance between constructs must be one at minimum. A semantic distance equal to zero would mean that this is one and the same construct.

In order to determine the semantic distance in the constructs, such metaclasses must be converted into an equivalent model. The State metaclass with its IsAttribute, for example, can be transferred into an abstract class State with four child classes. Thus, a semantic distance of two between the respective child classes would result therefrom. The enumeration metaclass pseudo-State can also be converted for the calculation of the semantic distance into an abstract class with corresponding child classes except for the metaclass Pseudostate ForkJoin. Here, fork and join have the same implementation [22] and have therefore a semantic distance of zero.

In summary, AD fulfills the discriminability requirement best and SC and SFC fulfill it partially.

4.3.3. Semantic Transparency

The principle of semantic transparency improves speed and accuracy of understanding by naive users [34] and means that the semantics of a symbol is transparent from its appearance alone. Here Moody [34] distinguishes between three levels of semantic transparency with a continuous changeover. Semantic immediacy means that the meaning of a symbol can be derived solely from its appearance without explanation (strong positive association). In contrast, semantic perversity means that a different or opposite meaning is associated by the appearance (negative association, false mnemonic). Between semantic immediacy and semantic perversity is the semantic opacity, in which the symbol has any desired relationship between appearance and meaning (neutral, conventional). The AD and SC symbols are international standard [32]. The common graphical design of SFC is also defined in a norm [17]. These defined symbols of AD, SC, and SFC are widely used and internationally understood. Moody investigated the symbols of the notation BPMN 2.0 [56]. The symbols of BPMN already contain some symbols that are used also in AD, SC and SFC. According to Moody, the symbols of the following constructs are semantically opaque:

- AD: Activity, Decision Node,
- SC: State, Decision Node, Junction, Start- und End-Node,
- SFC: Step, Init-Step.

In general, almost all symbols of AD, SC and SFC are semantically opaque. Generally, we can assume that process engineers are not experts and hence, they cannot interpret or properly associate the abstract symbols. For example, the entry, do and exit actions are implemented in CoDeSys with three different markers (cf. **Figures 2(a)-(c)**). The marker for entry, which is represented with the letter "E" and the marker for do, which is represented by a filled rectangle, can be interpreted differently by a naive user. However, the marker X, which stands for "exit-action", is more semantically immediate, because open windows in windows or other operating systems will leave with an "X" and therefore the association with "exit" is given. Generally, all of the symbols in AD, SC and SFC are semantically opaque. AD and SC additionally have the symbol swimlane and composite state respectively, which are more semantically transparent because of using spatial enclosure [43]. The Fork symbol of AD and SC is a split icon that effectively conveys the notion of "path transformation" but that does not clarify whether it represents a fork or a join [56]. The Join Symbol of AD and SC and the Branch symbol of SFC share its analysis with the Fork. Additionally, the alternative Branch symbol of SFC does not make clear if it stands for a normal transition or for an alternative branch. In summary, all symbols of the programming languages are semantically opaque und partially semantically perverse.

4.3.4. Cognitive Integration

The cognitive integration will only be used if a system is represented by several diagrams, whereupon conceptual and perceptual integration are distinguished. Conceptual integration is a mechanism that aims to help the reader to bring together information from individual diagrams into a coherent mental representation of the system.

The process flow for thermomechanical processes is hierarchical (cp. section 1). All three programming languages fulfill the conceptual integration because of this hierarchical representation. The AD also supports the structural information with use of swimlanes. The information comes from a different diagram (class diagram) and is integrated by the swimlane in AD. The perceptual integration, however, is a mechanism that aims at simplifying the navigation between multiple diagrams. Because of the project structure overview (which is now supported by all programming environments such as CoDeSys) all three languages fulfill this requirement.

4.3.5. Visual Expressiveness

The visual expressiveness gives a conclusion about the use of the full range and capacities of visual variables. This principle is measured by the number of used visual variables (planar and retinal variables) and the capacity (the range of values for each variable) [54]. The scale level and the capacity of the visual variables are defined in [43] [56]. The spectrum of the visual variable "texture" is fully used by all programming languages (see **Table 8**). The programming languages AD and SC use the spectrum of visual variable "color" with a saturation of at least 50%. The saturation of SFC however is only a maximum 28%, because only two colors are used. The planar variables are used in AD by swimlanes (vertical and horizontal) and SC by regions within a composite state.

In summary, the capacity of visual variables is used more in AD and SC than in SFC.

4.3.6. Dual Coding

This principle describes the use of a combination of text and graphics. As mentioned before, the discriminability of transitions and branches by SFC is very difficult, but the additional label branch on the visual representation makes it easier to convey information. All of the programming languages support the additional label information on action/state/step and transition that helps the process engineer on the one hand to modify the newly created or advanced program and on the other hand to understand the completed program. AD and SC additionally support labels on branches (such as decision node, junction, etc.). In

Table 8. Visual expressiveness of AD, SC and SFC.

[40] [54]			AD		SC		SFC	
Visual variable	Power	Capacity	Count	Saturation	Count	Saturation	Count	Saturation
Position (x,y)	Interval	10 - 15	2	13.3% - 20%	1	6.6% - 10%	0	0%
Shape	Nominal	Unlimited	5	-	4	-	3	-
Colour	Nominal	7 - 10	6	60% - 85.7%	5	50% - 71.4%	2	20% - 28.6%
Grain	Nominal	2 - 5	3	100%	3	100%	2	100%

summary AD, SC, and SFC fulfill the principle dual coding.

4.3.7. Graphic Economy

The graphical complexity is defined by the number of graphical symbols in a notation [35]. The number of symbols of AD, SC and SFC varies between 9 and 11. However, Miller's Law [18] states that the maximum number of objects an average human can hold in working memory is 9. The number of symbols of AD is 9, the one of SC is 11 and the one of SFC is 10 (cf. Table 7), so only AD is still in the range of Miller's Law.

4.3.8. Cognitive Fit

Different adapted representations of information should be used for different tasks and different target groups according to cognitive fit. For this purpose, the three points of cognitive fit (3-way fit) must be taken into account [43]:

- target group (customer, user, domain expert)
- medium (paper, whiteboard, computer)
- task characteristics

The notation must be adapted or improved, according to which target group is addressed, which medium is used, and what task is to be solved. Therefore, different representations of information within a notation can arise for different tasks and different target groups (expert, novice). The following 3-way fit points are relevant for the analysis of AD, SC and SFC:

- target group: process engineer of thermomechanical processes (no experts for AD, SC and SFC)
- medium: Computer, PLC Control Software
- task characteristics: flexible programming etc.

4.4. Summary of the Evaluation

Within the previous sections, the three programming languages AD, SC and SFC were evaluated concerning changeability, indicated by flexible programming, and maintainability. We assessed these two factors indirectly by use of the requirements modifiability, modularity and understandability. The result of the modifiability analysis has shown that SFC fulfilled the modifiability requirement best, followed by AD. In contrast, SC fulfilled this requirement marginally. According to the modularity analysis, AD is the only programming language, which meets this requirement completely. SC meets this requirement only partially and SFC hardly. Concerning the requirement understandability, AD fulfills many criteria completely in contrast to SC and SFC. In summary (see Table 9), AD meets the requirements of process engineers, namely changeability and maintainability, best. SFC and SC fulfill these requirements only partially. Therefore, the AD is the most suitable programming language according to the cognitive effectiveness and dimensions analysis.

5. Conclusion and Outlook

In industry, process engineers/technologists often have to adapt and design new

Table 9. Results of the analysis.

	Requirements	AD	SC	SFC
	Modifiabilty	+	o	+
	Modularity	+	o	-
Understandability	Semiotic Clarity	+	+	o
	Perceptual Discriminability	+	+	o
	Semantic Transparency	o	o	o
	Cognitive Integration	+	+	+
	Visual Expressiveness	+	+	o
	Dual Coding	+	+	+
	Graphic Economy	+	o	o
	Cognitive Fit		*no rating*	

control sequences from a technological point of view. More specifically, they change and test alternative control parameters to develop a certain product by combining prepared library elements from a logical point of view. However, we cannot expect any PLC or other programming skills from these process engineers or technologists. Their main requirements towards programming languages are changeability, indicated by flexible programming, and maintainability. These are represented by the requirements modifiability (R1.1), modularity (R1.2) and understandability (R2.1). An evaluation of these requirements was realized by use of the principles of cognitive effectiveness and dimensions. Since companies in this field of business usually rely on PLCs to control their (experimental) machines, we analyzed AD, SC and SFC concerning modifiability, modularity and understandability. It became apparent that AD is suited best to fulfill the specific requirements of process engineers, namely changeability and maintainability. Therefore, Activity Diagram should be provided as an additional language in PLC environments to allow technologists to adapt existing or design new recipes and control sequences. In future work these theoretical results should be proven by empirical validation with students and engineers. These results also allow further optimization of the programming languages' visual syntax for new generations of engineering environments.

References

[1] Vogel-Heuser, B. (2014) Usability Experiments to Evaluate UML/SysML-Based Model Driven Software Engineering Notations for Logic Control in Manufacturing Automation. *Journal of Software Engineering and Applications*, **7**, 943-973. https://doi.org/10.4236/jsea.2014.711084

[2] Mersch, H., Behnen, D., Schmitz, D., Epple, U., Brecher, C. and Jarke, M. (2011) Gemeinsamkeiten und Unterschiede der Prozess-und Fertigungstechnik (Commonalities and Differences of Process and Production Technology). *Automatisierungstechnik*, **59**, 7-17.

[3] Basile, F., Chiacchio, P. and Gerbasio, D. (2013) On the Implementation of Industrial Automation Systems Based on PLC. *IEEE Transactions on Automation Science and Engineering*, **10**, 990-1003. https://doi.org/10.1109/TASE.2012.2226578

[4] Vogel-Heuser, B., Friedrich, D. and Bristol, E.H. (2003) Evaluation of Modeling Notations for Basic Software Engineering in Process Control. *Annual Conference of the IEEE*, **3**, 2209-2214.

[5] Bayrak, G. (2015) Vergleich und Evaluation von Beschreibungsmitteln für die Automatisierung hybrider Prozesse. PhD Thesis, Technical University of Munich, Munich.

[6] Sturm, A., Dori, D. and Shehory, O. (2010) An Object-Process-Based Modeling Language for Multiagent Systems. *IEEE Transactions on Systems*, *Man*, *and Cybernetics Part C*: *Applications and Reviews*, **40**, 227-241. https://doi.org/10.1109/TSMCC.2009.2037133

[7] Walz, G.A. (1980) Design Tactics for Optimal Modularity. *Proceedings of AUTOTESTCON*: *International Automatic Testing Conference*, Washington DC, 281-284.

[8] Bhat, J.M. and Deshmukh, N. (2005) Methods for Modeling Flexibility in Business Processes. BPMDS Workshop in Conjunction with CAISE, Montpellier.

[9] Harrison, R., Counsell, S. and Nithi, R. (2000) Experimental Assessment of the Effect of Inheritance on the Maintainability of Object-Oriented Systems. *Journal of Systems and Software*, **52**, 173-179.

[10] Lee, J. and Hsu, P. (2007) Implementation of a Remote Hierarchical Supervision System Using Petri Nets and Agent Technology. *IEEE Transactions on Systems*, *Man*, *and Cybernetics*, *Part C*: *Applications and Reviews*, **37**, 77-85. https://doi.org/10.1109/TSMCC.2006.876056

[11] Curtis, B., Sheppard, S.B., Kruesi-Bailey, E., Bailey, J.W. and Boehm-Davis, D.A. (1989) Experimental Evaluation of Software Documentation Formats. *Journal of Systems and Software*, **9**, 167-207.

[12] Cunniff, N. and Taylor, R.P. (1987) Graphical vs. Textual Representation: An Empirical Study of Novices' Program Comprehension. *Empirical Studies of Programmers*: *Second Workshop*, Washington DC, 7-8 December 1987, 114-131.

[13] Meyer, B. (1992) Pictures Depicting pictures: On the Specification of Visual Languages by Visual Grammars. *Proceedings of the* 1992 *IEEE Workshop on Visual Languages*, Seattle, 15-18 September 1992, 41-47. https://doi.org/10.1109/WVL.1992.275785

[14] Kahn, K.M. and Saraswat, V.A. (1990) Complete Visualizations of Concurrent Programs and Their Executions. *Proceedings of the* 1990 *IEEE Workshop on Visual Languages*, Skokie, 4-6 October 1990, 7-15. https://doi.org/10.1109/WVL.1990.128375

[15] Glinert, E.P. (1990) Nontextual Programming Environments. In: Chang, S.-K., Ed., *Principles of Visual Programming System*, Prentice-Hall, Inc., Upper Saddle River, 144-230.

[16] Raeder, G. (1985) A Survey of Current Graphical Programming Techniques. *Computer*, **18**, 11-25. https://doi.org/10.1109/MC.1985.1662971

[17] Nordbotten, J.C. and Crosby, M.E. (1999) The Effect of Graphic Style on Data Model Interpretation. *Information Systems Journal*, **9**, 139-156. https://doi.org/10.1046/j.1365-2575.1999.00052.x

[18] Miller, G.A. (1956) The Magical Number Seven, plus or minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review*, **63**, 81-97. https://doi.org/10.1037/h0043158

[19] Lewis, R.W. (1998) Programming Industrial Control Systems Using IEC 1131-3. IET, London. https://doi.org/10.1049/PBCE050E

[20] Witsch. D. and Vogel-Heuser, B. (2011) PLC-Statecharts: An Approach to Integrate UML-Statecharts in Open-Loop Control Engineering-Aspects on Behavioral Semantics and Model-Checking. *World Congress of International Federation of Automation Control*, Milan, 29 August-3 September 2011, 7866-7872. https://doi.org/10.3182/20110828-6-it-1002.02207

[21] Witsch, D., Ricken, M., Kormann, B. and Vogel-Heuser, B. (2010) PLC-Statecharts: An Approach to Integrate UML-Statecharts in Open-Loop Control Engineering. *InternationalConference on Industrial Informatics*, Osaka, 13-16 July 2010, 915-920. https://doi.org/10.1109/indin.2010.5549619

[22] Witsch, D. (2013) Model-Driven Design of Control Software Basing on UML under Consideration of the Domain-Specific Requirements of Machine and Plant Engineering (Modellgetriebene Entwicklung von Steuerungssoftware auf Basis der UML unter Berücksichtigung der domänenspezifischen Anforderungen des Maschinen- und Anlagenbaus). PhD Thesis, Technical University of Munich, Munich.

[23] 3S-Smart Software Solutions, CoDeSys. http://www.codesys.com/

[24] Foehr, M., Lüder, A. and Steblau, A. (2012) Analyse der praktischen Relevanz verschiedener Beschreibungsmittel im Entwurfsprozess von Produktionssystemen. *Entwurf komplexer Automatisierungssysteme*, *Kongress*: *Fachtagung EKA*, Magdeburg, 61-72.

[25] Bayrak, G., Flach, A. and Vogel-Heuser, B. (2009) New Methods of Process Management in the Development of Technological Treatments. In: Steinhoff, K., Maier, H.J. and Biermann, D., Eds., *Functional Graded Materials in Industrial Mass Production*, Verlag Wissenschaftliche Scripten, Auerbach, Rep. of Collaborative Research Centre TRR30, 145-153.

[26] Zhou, M.C. and Twiss, E. (1998) Design of Industrial Automated Systems via Relay Ladder Logic Programming and Petri Nets. *IEEE Transactions on Systems*, *Man*, *and Cybernetics*, *Part C*: *Applications and Reviews*, **28**, 137-150. https://doi.org/10.1109/5326.661096

[27] Peng, S.S. and Zhou, M.C. (2004) Ladder Diagram and Petri-Net-Based Discrete-Event Control Design Methods. *IEEE Transactions on Systems*, *Man*, *and Cybernetics*, *Part C*: *Applications and Reviews*, **34**, 523-531. https://doi.org/10.1109/TSMCC.2004.829286

[28] Bahill, T., Alford, M., Bharathan, K., Clymer, J.R., Dean, D.L., Duke, J., Hill, G., LaBudde, E.V., Taipale, E.J. and Wayne Wymore, A. (1998) The Design-Methods Comparison Project. *IEEE Transactions on Systems*, *Man*, *and Cybernetics*, *Part C*: *Applications and Reviews*, **28**, 80-103. https://doi.org/10.1109/5326.661092

[29] Cao, L., Zhang, C. and Zhou, M. (2008) Engineering Open Complex Agent Systems: A Case Study. *IEEE Transactions on Systems*, *Man*, *and Cybernetics*, *Part C*: *Applications and Reviews*, **38**, 483-496. https://doi.org/10.1109/TSMCC.2008.923863

[30] Yu, L., Grüner, S. and Epple, U. (2013) An Engineerable Procedure Description Method for Industrial Automation. *Conference on Emerging Technologies and Factory Automation*, Cagliari, 10-13 September 2013, 1-8.

[31] Lukman, T., Godena, G., Gray, J. and Strmčnik, S. (2010) Model-Driven Engineering of Industrial Process Control Applications. *Conference on Emerging Technologies and Factory Automation*, Bilbao, 13-16 September 2010, 1-8. https://doi.org/10.1109/etfa.2010.5641224

[32] Object Management Group (2007) OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. OMG Document Number, formal/2007-11-02. http://www.omg.org/spec/UML/2.1.2/

[33] Green, T.R.G. and Petre, M. (1996) Usability Analysis of Visual Programming Environments: A "Cognitive Dimensions" Framework. *Journal of Visual Languages and Computing*, **7**, 131-174. https://doi.org/10.1006/jvlc.1996.0009

[34] Moody, D.L. (2011) Why a Diagram Is Only Sometimes Worth a Thousand Words: An Analysis of the BPMN 2.0 Visual Notation. https://pdfs.semanticscholar.org/dc09/25bd6d879f6b867806f8badfc70d2e30b4a4.pdf

[35] Moody, D.L. (2009) The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, **35**, 756-779. https://doi.org/10.1109/TSE.2009.67

[36] Britton, C. and Jones, S. (1999) The Untrained Eye: How Languages for Software Specification Support Understanding in Untrained Users. *Human-Computer Interaction*, **14**,191-244. https://doi.org/10.1080/07370024.1999.9667269

[37] Green, T.R.G. (1989) Cognitive Dimensions of Notations. In: Sutcliffe, V.A. and Macaulay, L., Eds., *People and Computers*, Cambridge University Press, Cambridge, 443-460.

[38] Roast, C., Khazaei, B. and Siddiqi, J.L.A. (2000) Formal Comparisons of Program Modification. 2000 *IEEE International Symposium on Visual Languages*, Seattle, 10-13 September 2000, 165-171. https://doi.org/10.1109/VL.2000.874380

[39] Moody, D.L. and van Hillegersberg, J. (2008) Evaluating the Visual Syntax of UML: An Analysis of the Cognitive Effectiveness of the UML Family of Diagrams. *International Conference on Software Language Engineering*, Toulouse, 29-30 September 2008, 16-34.

[40] Figl, K., Mendling, J., Strembeck, J.M. and Recker, J. (2010) On the Cognitive Effectiveness of Routing Symbols in Process Modeling Languages. *Lecture Notes in Business Information Processing*, **47**, 230-241.

[41] Barji, A., Hagge, N. and Wagner, B. (2006) Comparative Study of Using CNet, IEC 61499, and Statecharts for Behavioral Models of Real-Time Control Applications. *International Conference on Emerging Technologies in Factory Automation*, Prague, 20-22 September 2006, 750-757.

[42] Reijers, H.A. and Mendling, J. (2011) A Study into the Factors that Influence the Understandability of Business Process Models. *SMCA*, **41**, 449-462. https://doi.org/10.1109/tsmca.2010.2087017

[43] Genon, N., Heymans, P. and Amyot, D. (2011) Analysing the Cognitive Effectiveness of the BPMN 2.0 Visual Notation. *International Conference on Software Language Engineering*, Braga, 3-4 July 2011, 377-396. https://doi.org/10.1007/978-3-642-19440-5_25

[44] Bauer, N. (2002) Statecharts versus Sequential Function Charts. *Automatisierungstechnik*, **50**, 533-540.

[45] Wand, Y. and Weber, R.A. (1990) An Ontological Model of an Information System. *IEEE Transactions on Software Engineering*, **16**, 1282-1292. https://doi.org/10.1109/32.60316

[46] Jazdi, N., Maga, C. and Göhner, P. (2011) Reusable Models in Industrial Automation: Experiences in Defining Appropriate Levels of Granularity. *IFAC World Congress*, **18**, 9145-9150.

[47] Prieto-Diaz, R. and Neighbors, J.M. (1986) Module Interconnection Languages. *Journal of Systems and Software*, **6**, 307-334.

[48] Blackwell, A.F., Whitley, K.N., Good, J. and Petre, M. (2001) Cognitive Factors in Programming with Diagrams. *Artificial Intelligence Review*, **15**, 95-114.

https://doi.org/10.1023/A:1006689708296

[49] Vogel-Heuser, B., Fischer, J., Rösch, S., Feldmann, S. and Ulewicz, S. (2015) Challenges for Maintenance of PLC-Software and Its Related Hardware for Automated Production Systems: Selected Industrial Case Studies. *Industrial Conference on Software Maintenance and Evolution*, Bremen, 29 September-1 October 2015, 362-371.

[50] Blackwell, A.F. and Green, T.R.G. (2003) Notational Systems—The Cognitive Dimensions of Notations framework. In: Carroll, J.M., Ed., *HCI Models, Theories and Frameworks: Toward a Multidisciplinary Science*, Morgan Kaufmann, San Francisco, 103-134.

[51] Börger, E. and Stärk, R. (2003) Abstract State Machines: A Method for High-Level System Design and Analysis. In: *Science and Business Media*, Springer, Berlin. https://doi.org/10.1007/978-3-642-18216-7

[52] Moody, D.L., Heymans, P. and Matulevicius, R. (2010) Visual Syntax Does Matter: Improving the Cognitive Effectiveness of the I Visual Notation. *Requirements Engineering*, **15**, 141-175. https://doi.org/10.1007/s00766-010-0100-1

[53] Winn, W. (1993) An Account of How Readers Search for Information in Diagrams. *Contemporary Educational Psychology*, **18**, 162-185. https://doi.org/10.1006/ceps.1993.1016

[54] Bertin, J. (1983) Semiology of Graphics: Diagrams, Networks, Maps. University of Wisconsin Press, Madison.

[55] Gurr, C.A. (1999) Effective Diagrammatic Communication: Syntactic, Semantic and Pragmatic Issues. *Journal of Visual Languages and Computing*, **10**, 317-342. https://doi.org/10.1006/jvlc.1999.0130

[56] Genon, N., Amyot, D. and Heymans, P. (2010) Analysing the Cognitive Effectiveness of the UCM Visual Notation. In: *International Workshop on System Analysis and Modeling*, Springer, Berlin Heidelberg, 221-240.