Computers in Industry 61 (2010) 419–431

Contents lists available at ScienceDirect

Computers in Industry

journal homepage: www.elsevier.com/locate/compind

# Domain specific model-based development of software for programmable logic controllers

Gregor Kandare [*], Stanislav Strmčnik, Giovanni Godena

*Jozef Stefan Institute, Jamova 39, SI-1000 Ljubljana, Slovenia*

ARTICLE INFO

*Article history:*
Received 3 January 2008
Received in revised form 26 August 2009
Accepted 26 October 2009
Available online 2 December 2009

*Keywords:*
Procedural process control
Software engineering
Domain specific languages
Programable logic controllers

ABSTRACT

Procedural process control is responsible for coordination of control units that perform basic control in a typical industrial control system. Basic control, in turn, performs actions necessary for maintaining a desired state of process variables and equipment. Software in the domain of procedural process control consists of modules responsible for management of startup and shutdown sequences, exception handling and module communication. In this work we present the domain specific modeling language (DSL) ProcGraph together with its corresponding code generation tool that was designed for the development of software in the domain of procedural process control systems. The advantage of using a domain specific language is that not only the programmers, but also domain experts are able to understand and modify the code. The DSL code is self-documenting, as it is expressed in the idiom of the problem domain. In the article we present a formal description of the ProcGraph language. Furthermore, we describe how the formal model is used in the implementation of the automatic IEC 1131-3 code generator.

## 1. Introduction

Software for industrial control systems is without doubt very complex and challenging to develop. The reason lies in the very nature of control systems, which are designed to control machines, devices, and processes. If the software is not error-free, the machines, devices, processes can go out of control. Reliability, safety and real-time reactions are therefore among the most important attributes of this kind of software [1,2].

On the other hand, modern control systems cover an ample set of functions and encompass a wide assortment of hardware devices ranging from simple sensors, actuators, and controllers to complex computer systems. Consequently, control software is becoming ever more complex and therefore even more difficult to develop and maintain. It is therefore not surprising that the quality of software and the productivity of software development process are key issues also in the field of control systems [3].

A widespread approach of coping with some of the complexity issues mentioned above that is used by software providers and engineering companies in this field, is to employ the concept of reusability. In simplified terms, employing reusability means relying on a system of well-tested and field-proven basic software blocks and modules which can be configured into a dedicated system by using suitable tools. This approach is quite appropriate for realization of *basic control* functions (according to ISA S88 standard, [4]), which are primarily dedicated to establishing and maintaining a specific state of process equipment and process variables, and are the same or similar in entirely different processes.

In contrast to basic control functions, the *procedural control* activities, which consist of various sequences of events, transitions between states such as starting and stopping, are much more problematic from the software development point of view. The procedural part of control is entirely process-specific and requires development of custom software. Management of procedural control software development process is therefore quite an essential task. Implementation of procedural control functions (as well as basic control functions) is today largely based on *programable logic controllers* (PLCs). The main issue, therefore, is how to improve the PLC software development process.

The aim of this article is to present a new concept of procedural control software development based on the domain specific modeling language ProcGraph, and a special design tool for automatic generation of program code and documentation. In the

[*] Corresponding author at: Jozef Stefan Institute, Department of Systems and Control, Jamova 39, 1000 Ljubljana, Slovenia. Tel.: +386 14 773 798; fax: +386 14 773 994.
*E-mail address:* gregor.kandare@ijs.si (G. Kandare).

0166-3615/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.compind.2009.10.001

article, the transition between the final phases of software development, namely the translation of software models to source code (automatic code generation-code synthesis), is emphasized. The main contribution of the approach presented here is the application of domain specific modeling languages and tools in the field of procedural control software development.

In the following section, a modular structure of a continuous process control system is defined. Furthermore, some basic issues related to the PLC software development process are discussed. The subsequent section presents the modeling language ProcGraph together with its formal description. The technique of mapping ProcGraph models into IEC 61131-3 source code is presented. Further on, the code synthesis process and the corresponding software tool are described. In the last section, an example of application of the modeling tool is given.

## 2. Formalization of continuous process control structure

Formalization of process control structure is on the one hand necessary to deal with complexity of the control system by division of control functions. On the other hand the formalization is indispensable if we want to design the control software on a higher level of abstraction or even to automate a part of the software development process.

In the past a lot of work has been done with the aim of simplification and standardization of the design process of control systems and software. The design of batch process control systems and their organization is for example well covered by the ISA S88 standard. Various software tools for batch process control are available on the market. This is, however, not the case in the domain of continuous processes.

Let us for the purpose of software design formalize the continuous process control with the aid of the batch control standard ISA S88. In accordance with the standard, the control system of a continuous process can be divided into two parts, a *basic control* module and a *procedural control* module, as shown in Fig. 1. By partitioning the control system, its modularity is achieved. The modularity yields easier analysis, design, implementation, use, and maintenance of the system. Moreover, the software developer observes the software through the eyes of a domain expert (process engineer). If the software developer and domain expert use the same abstractions, the communication between them is easier and less prone to ambiguity.

Basic control is dedicated to establishing and maintaining a specific state of the process equipment. Devices used in basic control are controllers, programable logic controllers (PLCs), sensors, and actuators as well as other similar equipment that can be equipped with dedicated software as well as corresponding modules for basic control.

Procedural control, on the other hand, directs equipment-oriented actions to take place in an ordered sequence in order to carry out a process-oriented task. Equipment-oriented actions are performed by sending commands to basic control, in the same way as a human operator would drive the process manually. The main dynamics of procedural control consists of sequences of starting and stopping actions as well as handling of extraordinary situations such as alarms and safety shutdowns. The manner in which these situations are to be handled depends primarily on the requirements of the controlled process. Procedural control entities are therefore more unique than basic control entities. Fig. 1 shows the communication between procedural control and basic control. The commands depend on the input signals and the current state of the procedural control. This situation is typical of reactive systems. For that reason we can consider a procedural control system to be a reactive system.

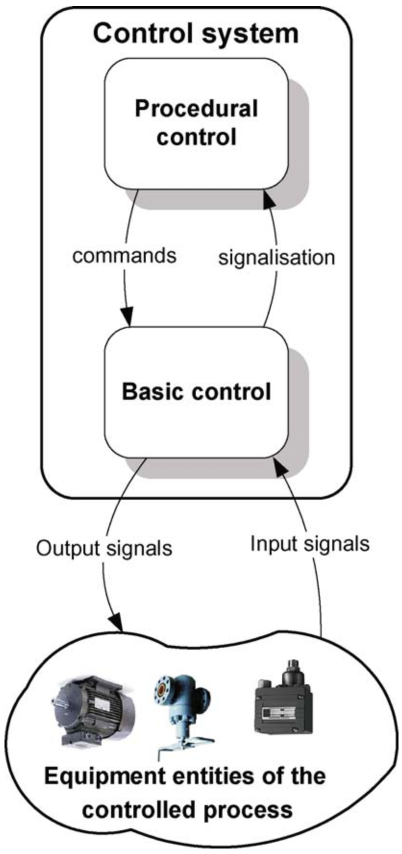


**Fig. 1.** Control system of a continuous process.

## 3. Basic issues related to improvement of PLC software development process

Programmable logic controllers (PLCs) are the devices most frequently used to perform basic and procedural process control. The process of PLC software development is not a straightforward one, due to the high demands for control software quality and reliability. Furthermore, the PLC programming languages, recommended by the IEC 61131-3 standard described in [5], provide a rather low level of abstraction. This aspect makes the PLC programming languages unsuitable for complex systems.

Several solutions for tackling the problems of PLC software development have been suggested in the literature such as [1–3,6–9]. The solutions are mainly based on a lifecycle view of software. According to this view, software is considered as any other product that undergoes various stages of evolution throughout its creation and application. Consequently, the process of software development and use (evolution of a software product) can be divided into several phases of the lifecycle. The software lifecycle usually starts with a requirements definition phase and ends with operation and maintenance phases. The intermediate phases are development phases of the product, which can be further divided into modeling and realization phases. In the modeling phases, modeling and software tools for analysis and design are used.

Currently, the most widespread modeling language for object-oriented systems is UML (unified modeling language, [10]). In the field of procedural control software, however, the drawback of using UML or similar standard modeling languages is that they are too generic and not sufficiently domain specific. The use of generic modeling languages increases *cognitive distance* defined in [11], which is a measure of the effort needed to progress from one phase of software development to the next. To reduce the cognitive distance, it is preferable that similar abstractions be used throughout all phases of software development. This can be achieved by using domain specific modeling languages [2,12–17]. The main difference between domain specific modeling languages and generic modeling languages such as UML is that the elements of the domain languages represent items from the domain world, whereas UML elements represent things from the code world. This enables the developer to concentrate on building solutions in the domain without worrying about the code. While UML does not raise the abstraction level with respect to the programming languages, the domain specific modeling languages do. The abstraction gap between domain specific modeling languages and programming languages can be compared to the gap between high-level programming languages such as Java and low-level machine languages. Since the domain specific language design requires a detailed knowledge about the domain, it has to be carried out by domain experts. One further advantage of domain specific modeling languages is that they automate a significant fraction development phases which helps to boost productivity and reliability.

Another issue in the procedural control software design process is the choice of appropriate computer automated software engineering tools (CASE). CASE tools play the same role in software development as CAD/CAM (computer-aided design/computer-aided manufacturing) tools play in the development of other products. The main objective of CASE tools is to automate development phases and transitions between the phases in a software lifecycle. Thus, CASE tools support design, editing, consistency and correctness checking as well as saving and retrieval of graphical and textual models of software. Last but not least, an essential capability of CASE tools is that they can automatically generate program code from the models.

One of the main problems related to CASE tools is that the ones available on the market only support standard modeling languages such as UML, YSD, Statecharts and similar generic modeling languages. The majority of the CASE tools can automatically generate code in one of the high-level programming languages such as C++ or Java. Moreover, code generation is most frequently limited to the mapping of architectural diagrams such as UML class diagrams to static code constructs and modules.

According to [18], if we want to use computer automated design tools in specific domains such as the development of procedural control software, we have two options. The first option is to use an existing commercial CASE tool that supports a modeling language that serves our purpose best and ignore or neglect the differences. In the majority of cases, however, automatic code generation provided by the commercial CASE tools cannot be used because domain specific modeling languages are not supported. Alternatively, we can develop a domain specific CASE tool [19] that supports domain specific modeling languages as well as conversion to desired target programming languages. The second alternative requires much higher initial effort but proves to be more efficient in the long run. A well-known fact is that synthesis of domain specific code is more efficient than synthesis of generic code [20]. In this article, the second alternative is presented using an earlier-developed modeling language, called ProcGraph, as the key element. To implement code synthesis in the CASE tool, a formal description of the modeling language syntax as well as semantics has to be made. Formal description of the modeling language also allows us to perform syntactic analysis of the models, such as consistency and correctness checking. The following section provides a brief insight into the ProcGraph syntax.

## 4. Formal description of the ProcGraph modeling language

ProcGraph is a modeling language specialized for the domain of procedural process control software [21]. The advantage of ProcGraph over generic modeling languages lies in its use of the same abstractions for the description of the control system as the process engineer uses for the description of the process control system (*subject-domain-oriented decomposition*, [22]).

ProcGraph models consist of three types of diagrams, each describing a particular aspect of the control software:

- *Procedural control entities diagram* (PCED) depicts the conceptual decomposition of the system as well as relationships between conceptual components.
- *State transition diagram* (STD) describes the dynamic view (behavior) of conceptual components.
- *Entity dependency diagram* (EDD) portrays causal and conditional dependencies between conceptual components (procedural control entities).

### 4.1. Procedural control entities diagram

Procedural control entities diagram depicts the architectural view of the control software system. Nodes in the PCED represent conceptual components—procedural control modules. Connections in the PCED stand for relationships (dependencies) between the procedural control entities. An example of a procedural control entities diagram is shown in Fig. 2. Full arrows illustrate that a certain state transition of the arrow sink entity depends on a certain state of the arrow source entity (*conditional dependency*). Dashed arrows indicate that entering of an arrow source entity into a certain state fires a certain state transition of the arrow sink entity (*causal dependency*).

From a topological point of view the procedural control entities diagram is a directed graph. The nodes of the graph are represented by rounded rectangles. Fig. 3 illustrates a graphical description of procedural control entities diagram syntax using UML class diagrams. The syntax determines that each node has a name and can be connected to one or more nodes. Each connection has a type (full or dashed).

Another way of describing the syntax of procedural control entities diagrams is by using the 6-tuple:

$$\langle \mathrm{PCED} \rangle = \{V_\mathrm{E}, P_\mathrm{E}, \rho, iz, po, tp\}, \quad (1)$$

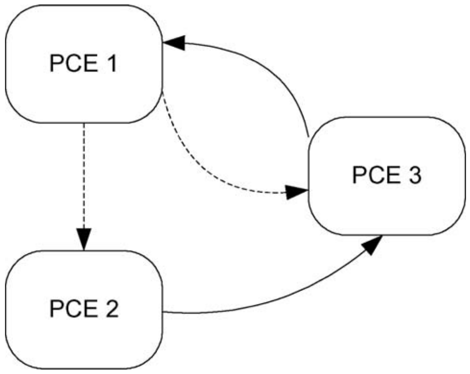


**Fig. 2.** Procedural control entities diagram of a continuous process control system.

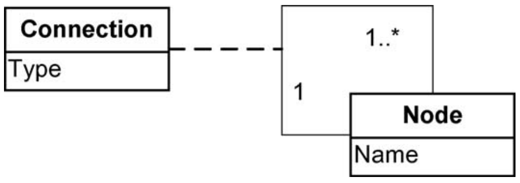


**Fig. 3.** Syntax of procedural control entities (PCE) diagrams.

where $V_E$ is a set of nodes, $P_E$ a set of connections, $\rho$ a function that maps the nodes to corresponding state transition diagrams, *iz* a function that returns the source node of a connection, *po* a function that returns the sink node of a connection and *tp* a function defining the type of connection.

## 4.2. State transition diagram

Procedural control systems are reactive systems by nature. The most appropriate model for describing the behavior of such systems is the *state transition diagram* (STD). The description of this model can be found in [22–26] and several additional sources. State transition diagrams are graphical models consisting of vertices that represent states and arcs that portray transitions.

ProcGraph uses an extended version of STDs. Besides states and transitions, extended state transition diagrams also contain *composite states* or *superstates*. Superstates can contain substates and can also be a part of other superstates. With the introduction of superstates, a reduction in the amount of transitions is achieved. In this manner the model can be simplified without a loss of information. By using superstates a hierarchy of states can be built in a model. Furthermore, hierarchical structure also enables us to simplify the top-down design of the system. The design can be performed starting from the highest level states and refining then stepwise all the way down to the basic states.

Each state has a unique name. Transitions are designated according to the syntax:

$$\frac{\text{event[condition]}}{\text{action}}$$

This designation can be interpreted as follows: if the *event* occurs and the *condition* is fulfilled, the transition fires and the *action* is executed. Fig. 4 shows an example of an extended state transition diagram.

The state hierarchy of the diagram in Fig. 4 is represented by the tree in Fig. 5. The root node represents the entire state transition diagram. Basic states are represented by leaf nodes and composite states by subtrees. In Fig. 4, concurrency is interpreted in the sense that if a node is active, all nodes on the path leading to the root node are also active.

Fig. 6 illustrates a graphical description of the state transition diagram syntax. From this figure we can see that each node can contain zero or more nodes and that each node is connected via one or more connections with further nodes.

Syntax can also be described by the 9-tuple:

$$\langle \text{STD} \rangle = \{V_S, P_S, iz, po, nad, I, P, O, \lambda\}. \tag{2}$$

$V_S$ is a set of nodes, $P_S$ a set of connections, *iz* a function that returns the source node of a connection, *po* a function that returns the sink node of a connection, *nad* the function that returns a supernode of a node, *I* a set of input signal descriptions, *P* a set of parameter descriptions, *O* a set of descriptions of output signals, and $\lambda$ a function that attributes a logical expression to each transition from the set $P_S$. The logical expression represents the event that fires the transition.

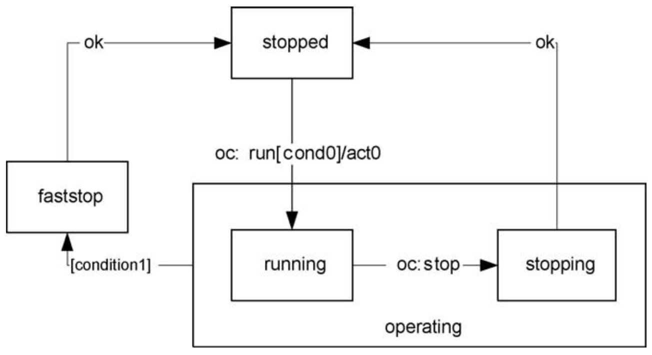


Fig. 4. A sample extended state transition diagram.

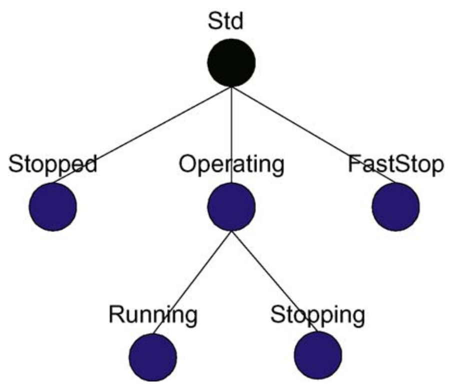


Fig. 5. State hierarchy of the state transition diagram in Fig. 4.

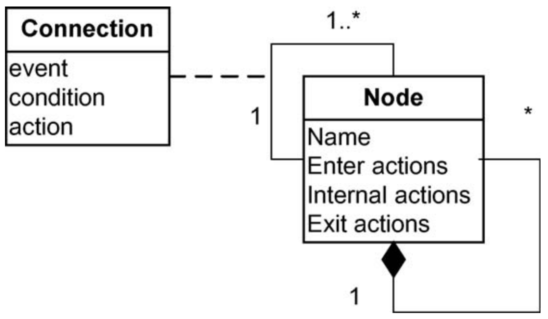


Fig. 6. Syntax of the state transition diagrams.

## 4.3. Entity dependency diagram

Entity dependency diagram (EDD) is a blend of the procedural control entities diagram and the state transition diagram. The EDD depicts state transition diagrams of two or more procedural control entities and the dependencies (relations) between the state transition diagrams. While the procedural control entities diagram shows only the existence of dependencies between entities, the entity dependency diagram also specifies the sources and sinks of relationships in more detail. A conditional relationship is represented by an arrow, while a causal relationship is indicated by a dashed arrow. The arrows in an EDD always originate in nodes (states) and sink in transitions. Fig. 7 shows a sample EDD.

Syntax of the entity dependency diagrams in terms of UML is shown in Fig. 8. The syntax is similar to the syntax of state transition diagrams. The difference is that in addition to connections between the nodes there also exist connections (representing dependencies) that originate in nodes and sink in connections.

The 8-tuple:

$$\langle \text{EDD} \rangle = \{V_S, P_S, O, iz, po, izp, pop, nad\} \tag{3}$$

also describes the syntax of entity dependency diagrams. Compared to the expression (2), the expression (3) contains a set of conditional relationship connections (*O*) and functions *izp* and *pop* that return the connection source node and sink transition connection, respectively. On the other hand, the transitions in entity dependency diagrams are not designated. Therefore (3) does not contain the sets containing the elements of transition designation.

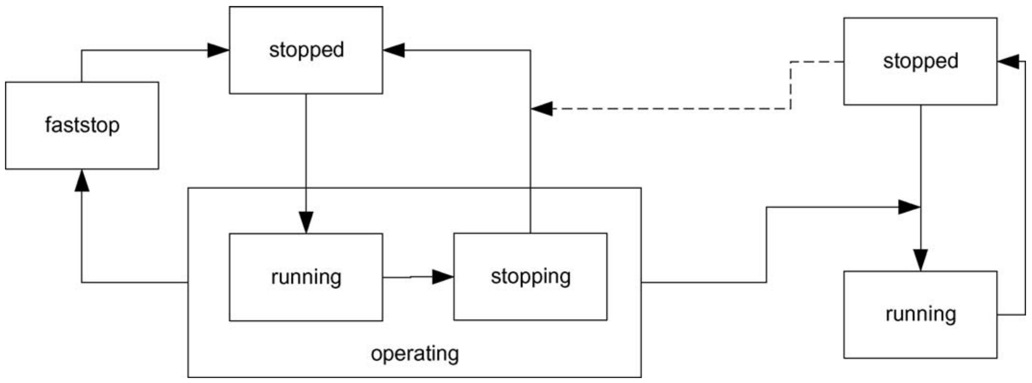


**Fig. 7.** A sample entity dependency diagram.

## 5. Mapping of models to source code

To achieve seamless transitions between the development phases of a software product, abstractions appearing in the models of each phase have to be as similar as possible. If this is the case, the effort needed to advance from one phase to another is minimized.

As we have seen earlier, the ProcGraph modeling language is designed in such a manner that its abstractions match closely with the ones that appear in the problem domain (procedural process control). The modeling phase is followed by the programming (coding) phase. Whereas in the modeling phase we can design and adapt the modeling language so that it suits our needs, language adaptation is not feasible in the programming phase. The model that results from the programming phase is in fact program source code in one of the programming languages. Programming languages have fixed syntax and semantics that cannot be changed by the programmer. Furthermore, in the choice of the target programming language, we are limited by the hardware platform of the control system.

In the case of programmable logic controllers, available programming languages are defined by the IEC 61131-3 standard [5]. This standard defines five programming languages: ladder diagram (LD), sequential function charts (SFC), function block diagram (FBD), structured text (ST), and instruction list (IL). While the latter two are textual, the first three are both graphical and textual. Considering the seamlessness issues, the most appropriate language for our purpose appears to be the function block diagram. Its abstractions and their hierarchy match well with the abstractions and hierarchy of ProcGraph models. FBD is a dataflow diagram where function blocks represent functions. In some implementations FBD can also contain ladder diagram elements such as switches and coils.

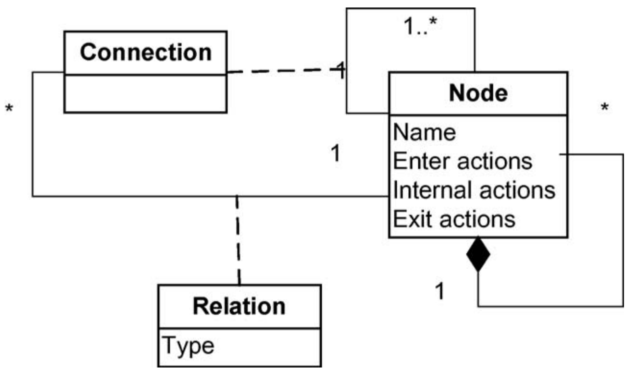


**Fig. 8.** Syntax model of the entity dependency diagrams.

For the description of function block diagram syntax, the triplet:

$$\langle \mathrm{FBD} \rangle = \{B, L, V\} \tag{4}$$

can be used, where $B$ represents a set of function blocks, $L$ a set of connections and $V$ a set of variables. Furthermore, the syntax of function block diagram language is depicted in Fig. 9.

The syntax definition in Fig. 9 shows that a function block can be connected to zero or more function blocks or to zero or more variables. Each function block has a name, input and output ports, internal variables, and an algorithm that is executed while the function block is active.

### 5.1. Definition of the mapping function

A ProcGraph model consists of three different types of submodels (diagrams), which can be described with the expression:

$$\langle \mathrm{ProcGraph} \rangle = \langle \mathrm{PCED} \rangle + \langle \mathrm{STD} \rangle + \langle \mathrm{EDD} \rangle. \tag{5}$$

Using the expressions (4) and (5), we can define the mapping function of ProcGraph models into function block diagram language as:

$$C_G : \langle \mathrm{ProcGraph} \rangle \rightarrow \langle \mathrm{FBD} \rangle. \tag{6}$$

The expression (6) is general and describes the transformation of the entire ProcGraph model to function block diagram source code.

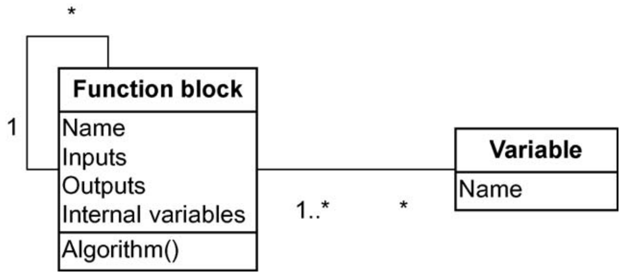


**Fig. 9.** Syntax model of function block diagram programming language.

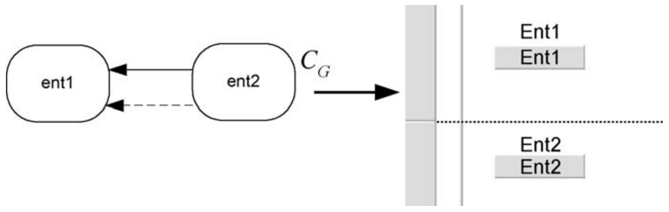


**Fig. 10.** Mapping of procedural control entities diagrams.

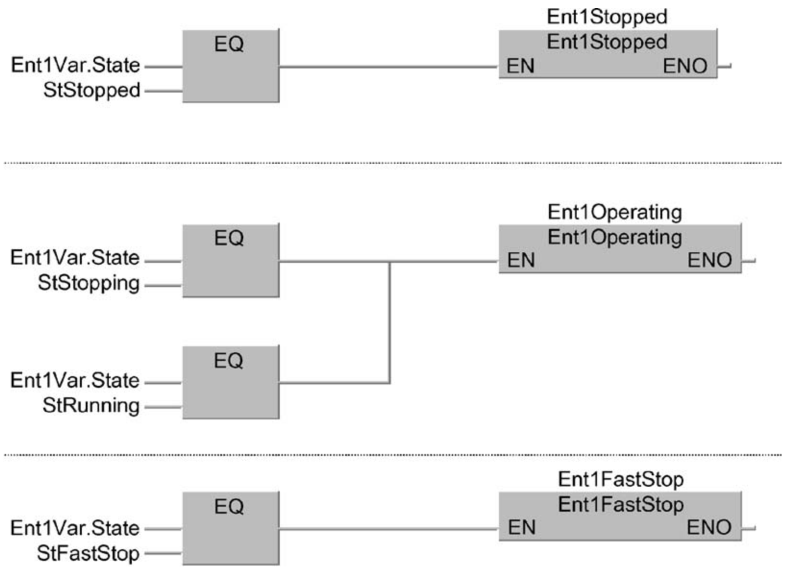


**Fig. 11.** The code of the body of the function block *ent1* from Fig. 10.

Respecting the seamlessness principle, the nodes of ProcGraph models are mapped to function blocks. As we have seen earlier, the nodes of ProcGraph models represent procedural control entities in procedural control entities diagrams and states in state transition diagrams. Fig. 10 shows an example of how two entities of a procedural control entities diagram map to corresponding function blocks in function block diagram code.

Each entity in the model corresponds with a state transition diagram, which describes its dynamics. For example, the dynamics of the entity e*nt1* in Fig. 10 is described by the state transition diagram in Fig. 4. Therefore, the algorithm of each function block depicting an entity is an implementation of the entity's state transition diagram. Fig. 11 shows the implementation code of the state transition diagram shown in Fig. 4.

To ensure seamlessness, the hierarchy of program code elements (function blocks) is equivalent to the hierarchy of model elements (procedural control entities and states). In Fig. 11, consequently, appear only function blocks depicting the states of the highest hierarchy level of STD from Fig. 4. The substates are implemented by function blocks that are called (activated) at a

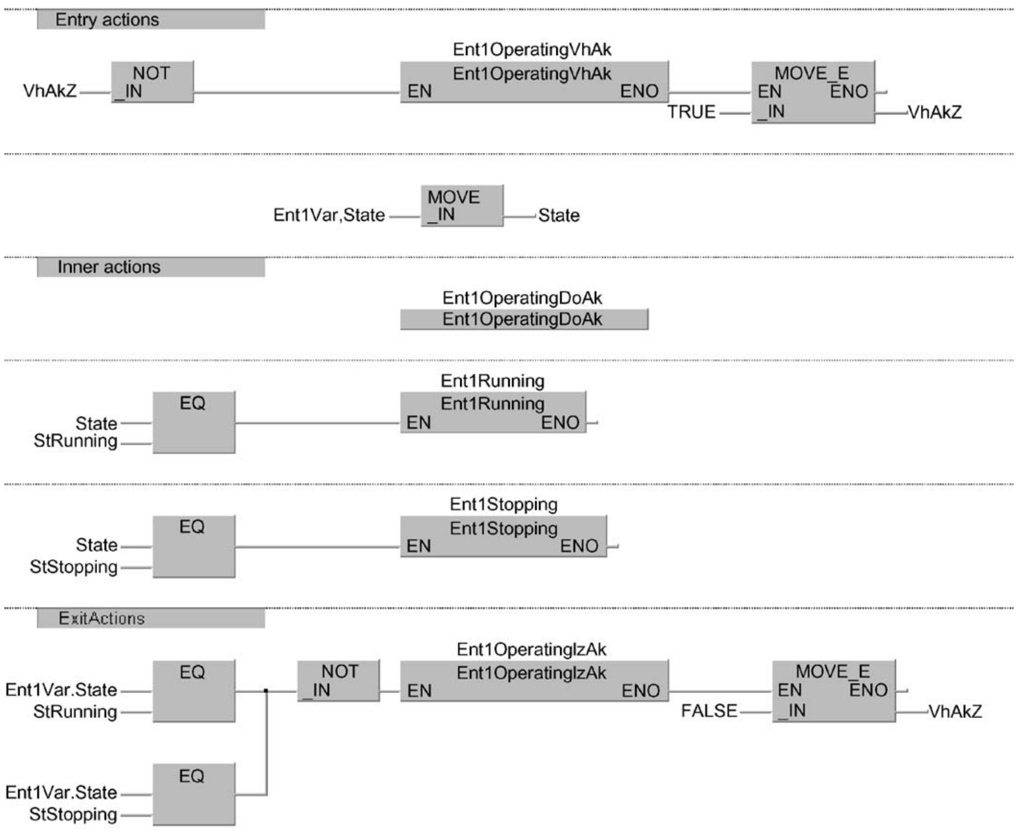


**Fig. 12.** The body of the *ent1operating* function block.

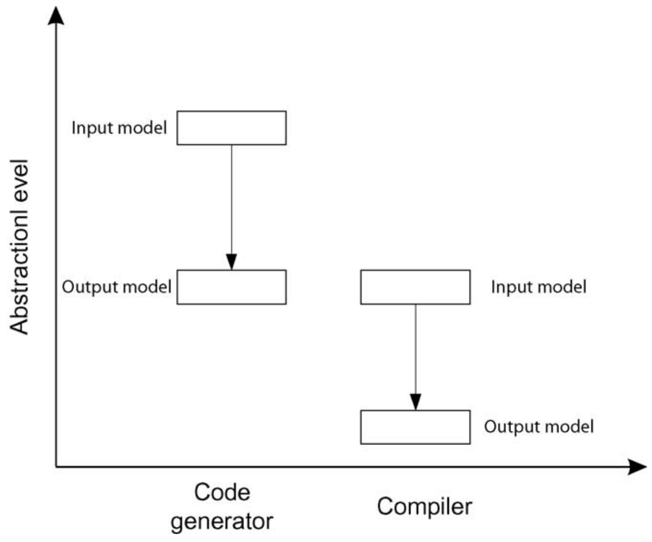


**Fig. 13.** Code generator and compiler abstraction levels.

corresponding level of function block hierarchy. The *operating* state of the STD in Fig. 4 contains the states *running* and *stopping*, therefore the algorithm of the function block that corresponds to the *operating* state contains calls of function blocks that embody the *running* and *stopping* states (Fig. 12). If, for example, the *running* state had a further substate, the substate function block would be called inside the *running* state function block algorithm. In addition to the function blocks representing the substates, the code in Fig. 12 also encloses function blocks that contain entry, inner and exit actions of the state. The entry and exit actions are executed when the system enters and exits the state, respectively. The inner actions are executed repeatedly while the system is in the corresponding state. The state transition mechanism together with the actions of particular states is embedded inside the code of the algorithm of function blocks representing the states.

## 6. Automatic code generation

Automatic code generation has gained in importance over the past few years. It has also been known under names such as model-based programming and model-driven application development. The reason for the popularity of automatic code generation is the fact that software development companies realize that manual programming is a time-consuming and error-prone task. Moreover, if the software model built by the analyst/designer is sufficiently complete and at an appropriate abstraction level, the programming itself is a relatively mechanical task that can easily be taken over by a computer. In automating the programming process, costs of the human resources can be reduced and development times can be significantly shortened. Furthermore, the error incidence rate is reduced.

Automatic code generator resembles a compiler in the sense that it transforms a model from a higher level of abstraction to a model on a lower level of abstraction. In the case of a compiler, the input model is the program source code and the output model the executable code. On the other hand, the objective of a code generator is to transform the model made by the designer to the

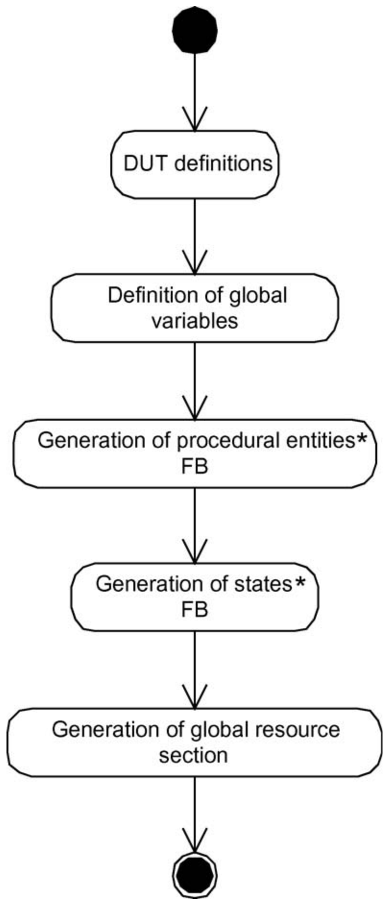

DUT definitions
Definition of global variables
Generation of procedural entities* FB
Generation of states* FB
Generation of global resource section

**Fig. 15.** Automatic code generation algorithm.

source code. The output model of the code generator is therefore the input model of the compiler. Abstraction levels of the aforementioned models are shown in Fig. 13.

### 6.1. Automatic code generation from ProcGraph models

The process of conversion from software model to the end product of software development (executable code) is shown in Fig. 14. The input of the code generator is a software model (in our case a ProcGraph model) and the output is a corresponding source code (FBD). The code is generated on the basis of the information about the ProcGraph model and parameters as well as code templates implemented in the code generator. A rough description of the code generation procedure is depicted with the activity diagram shown in Fig. 15. Definitions of data structures are created first, followed by the generation of function blocks representing procedural control entities. Furthermore, a hierarchy of function blocks representing states is constructed and finally global variables are defined. Each step of the algorithm from Fig. 15 is further decomposed into more atomic procedures that finally result in atomic actions.

The code generator creates a text file describing both graphical as well as textual segments of the source code in the function block diagram. Creating the graphical part of the code is a more

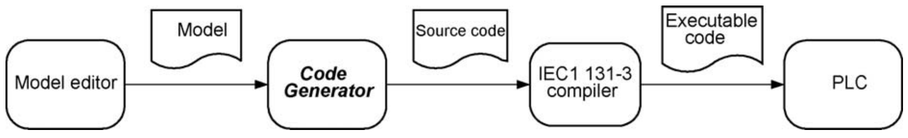


**Fig. 14.** Conversion of software models to executable code.

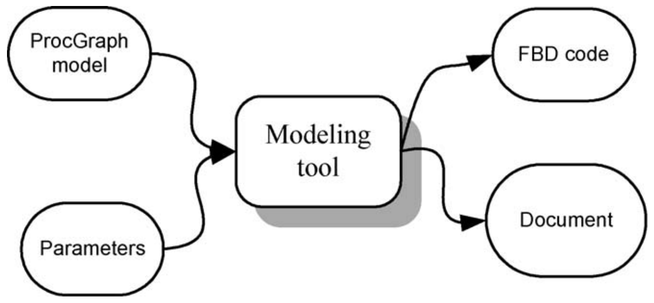


**Fig. 16.** Inputs and outputs of the modeling tool.

demanding task because topological issues such as placement of elements in the scheme and routing of connections have to be taken into account. The generated text file is imported into the IEC 61131-3 programming tool, which interprets it as a function block diagram source code. To obtain executable code, the source code has to be compiled.

### 6.2. Software tool for modeling and code generation

The function of the modeling tool is to provide a graphical editor for ProcGraph models. Furthermore, the tool should automatically generate function block diagram code and a report document code from the model. Fig. 16 shows the modeling tool from the point of view of its inputs and outputs.

A use case diagram representing the functionality structure of the tool is shown in Fig. 17. The functionalities are represented by ellipses. Activities of some of the functionalities such as model editing require direct involvement of the user (control software developer). Additional functionalities, such as code generation, only have to be activated by the user.

In generating program code in a graphical programming language such as function block diagram, we have to create not only the correct content but also the form (graphical image) of the code. For example, if a simple program scheme consists of some blocks and connections between them, it is not sufficient to describe the blocks and connections. An exact placement of blocks and routing of connections in the program scheme have to be portrayed as well.

The user interacts with the modeling tool by using two types of editors: with the help of textual interfaces she or he edits the information about parameters, commands, device types, and device instances. Furthermore, complementary graphical interfaces are used for editing of the graphical part of ProcGraph models—procedural control entities diagrams, state transition diagrams, and entity dependency diagrams.

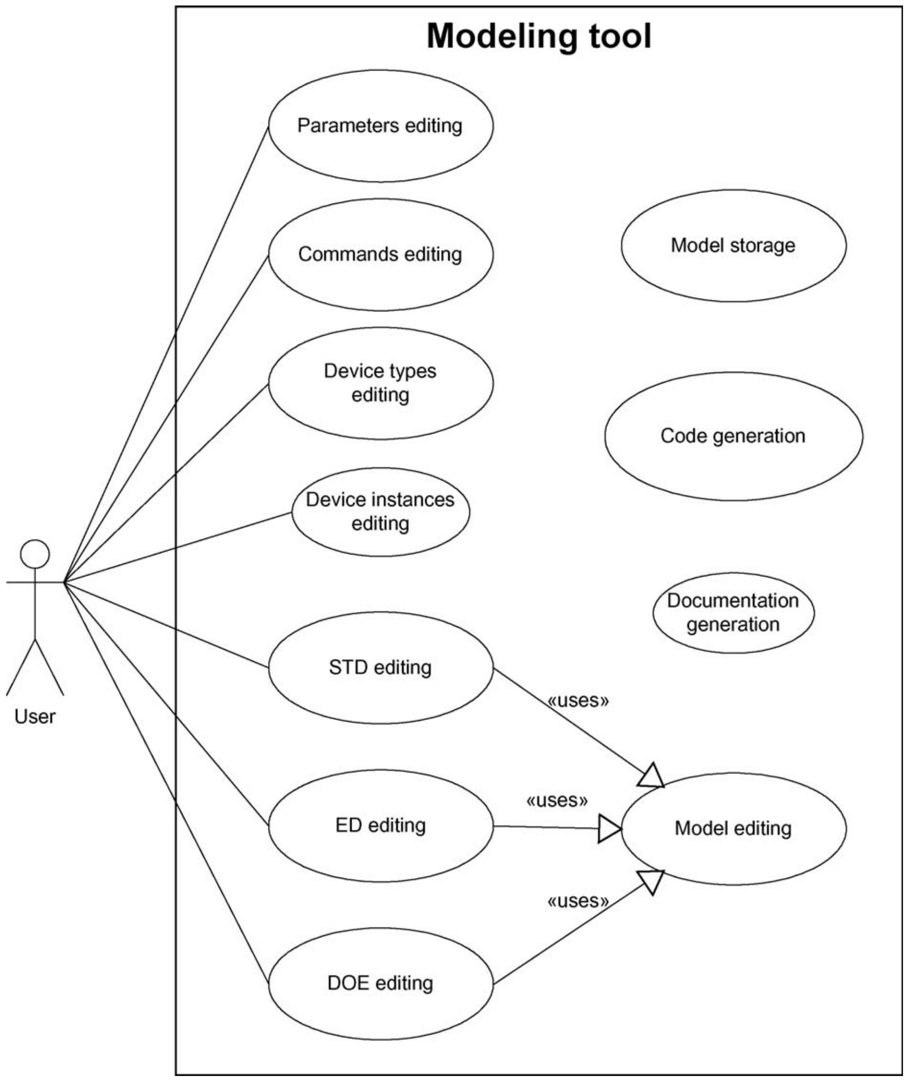


**Fig. 17.** Functionality of the modeling tool.

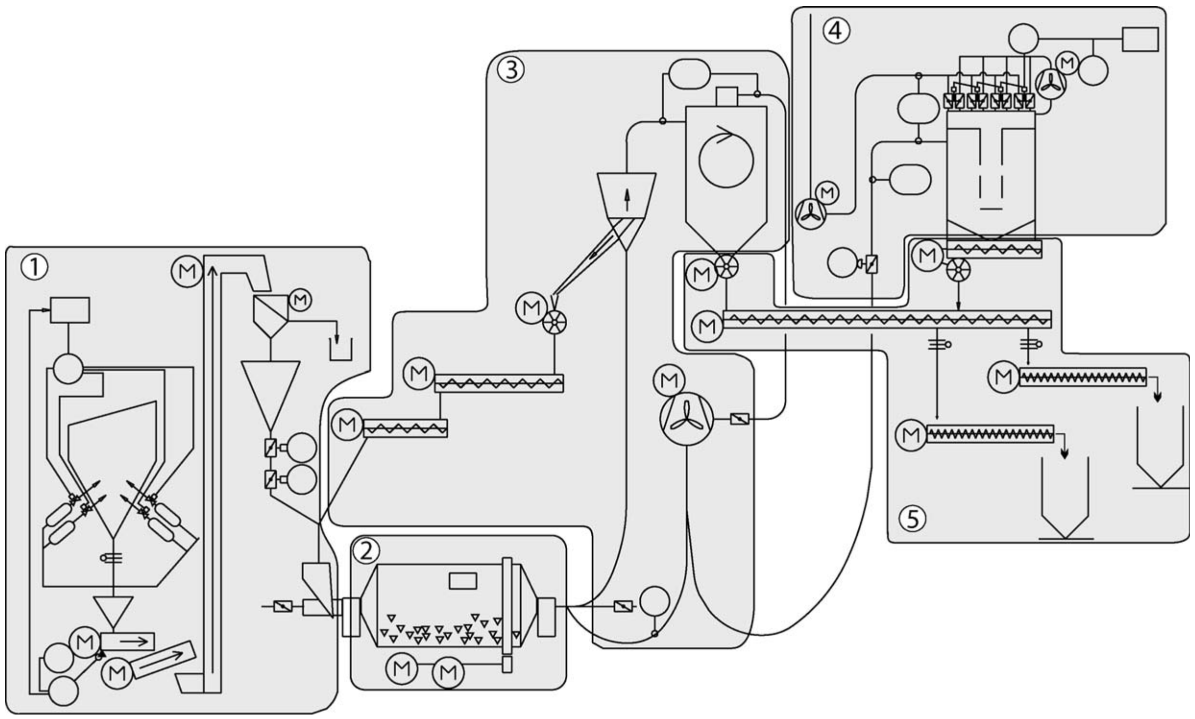


**Fig. 18.** Ore grinding process.

## 7. Example of application of the modeling tool

To demonstrate the use of the modeling tool a control system of an ore grinding process described in [27] is employed. The process illustrated in Fig. 18 was divided by the plant engineers into five subprocesses: dosing, grinding, pneumatic transport, dust separation, and silo transport. From the storage silo, the ore is poured through a funnel onto a belt scale, where a frequency converter controls the mass flow of the ore. From the belt scale, the ore is transported by a conveyor belt to the elevator and from there to a

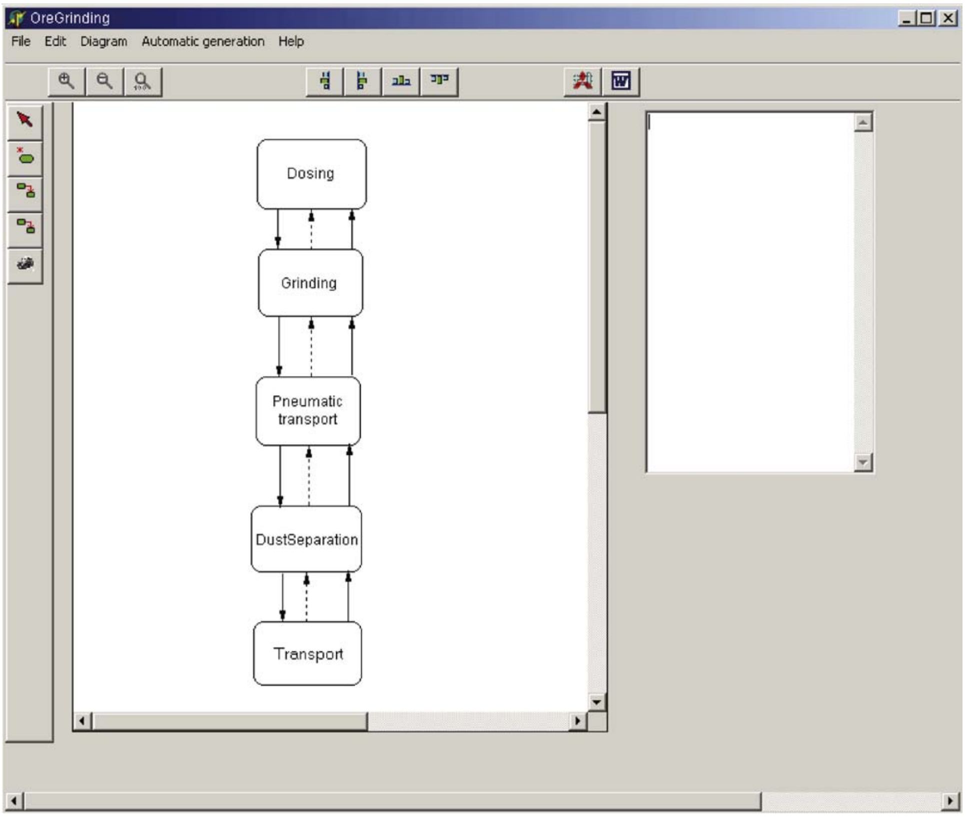


**Fig. 19.** Screenshot of the main window of the modeling tool.

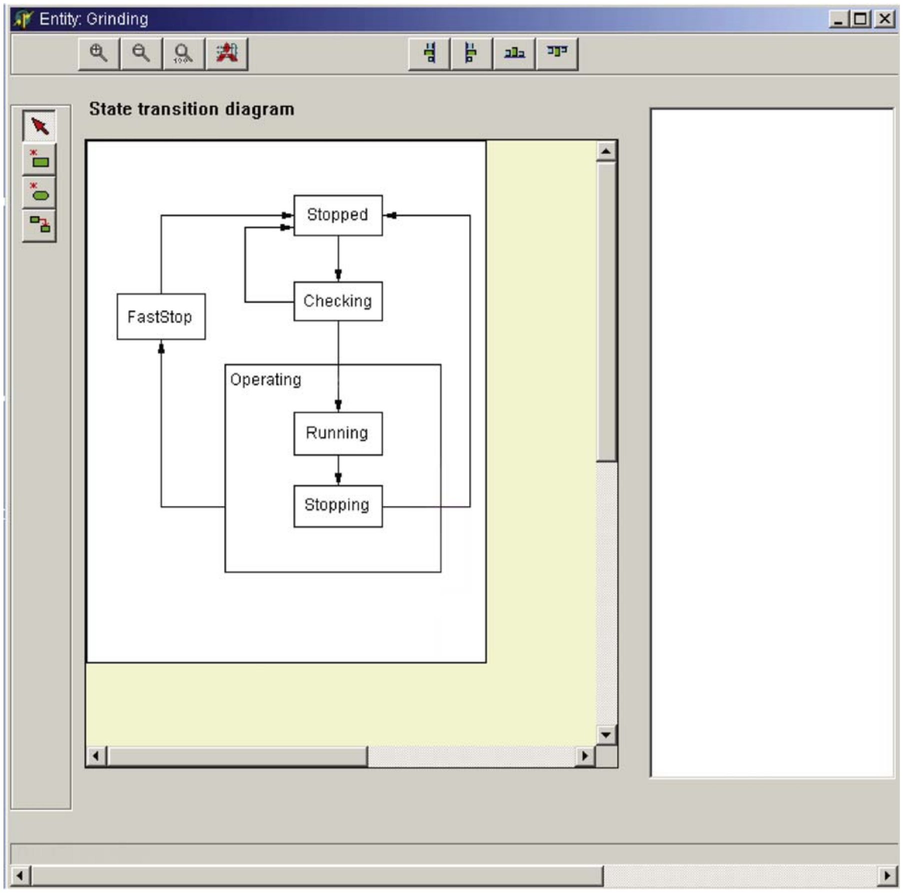


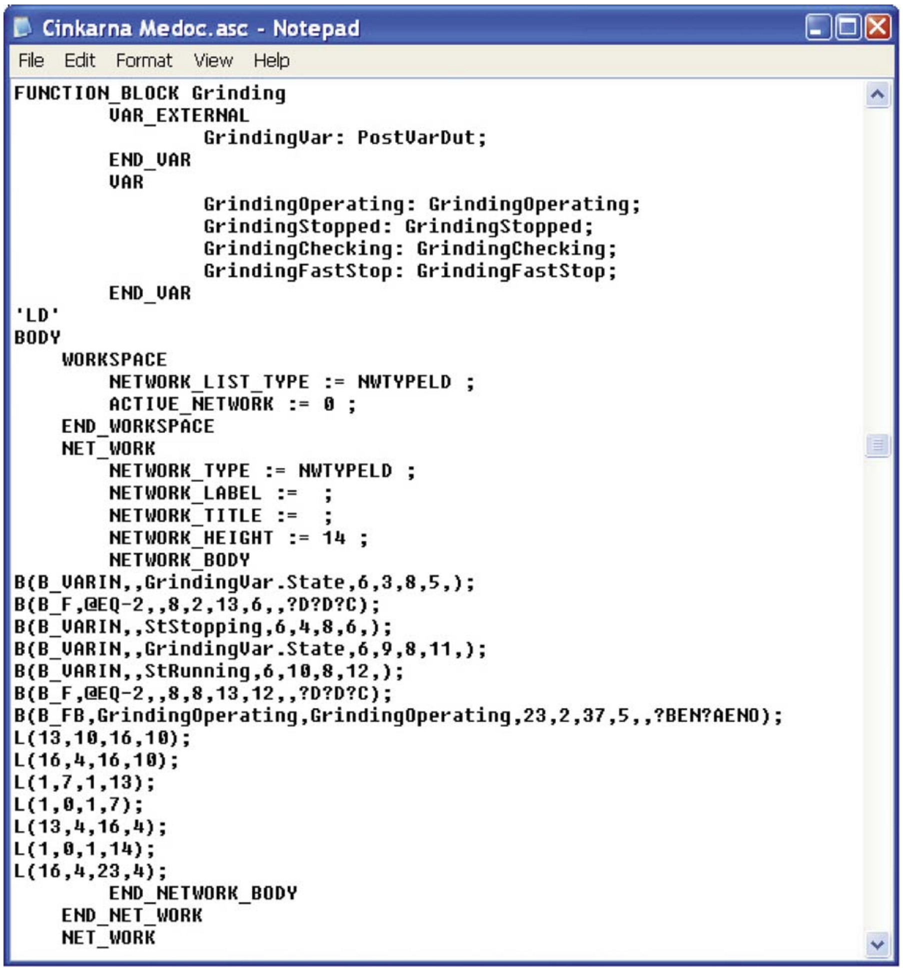
**Fig. 20.** State transition diagram editor.

vibration sieve, where coarser particles and impurities are removed. The sieved ore falls through a funnel and a damper system into the grinding mill. The damper system prevents excessive airflow from entering the mill. In the rotating mill, the ore is ground by the grinding bodies. The ore then travels to the separator, where any unground ore is separated and fed back into the mill by a conveyor belt system. The ground ore is transported to a cyclone air separator, where fine particles are extracted and transported by a conveyor belt system to the corresponding storage silo. The excessive air in the pneumatic transporting system is then led to a bag filter, where it is cleaned and released through the chimney.

According to the principles of subject-domain-oriented decomposition, the main architectural modules (procedural control entities) of the control system are chosen in such a manner that they reflect the decomposition of the controlled process. Thus, the control system consists of the five procedural control entities: dosing, grinding, pneumatic transport, dust separation, and silo transport.

The screenshot in Fig. 19 shows the main window of the modeling tool. The main window contains the graphical editor of procedural control entities diagrams. The user creates a procedural control entities diagram model by dragging and dropping the entity elements (rounded rectangles) onto the drawing surface and connecting them with respective arrows. In the case of ore grinding process control, the procedural control entities diagram contains the five procedural control entities. Connections between the nodes in Fig. 19 indicate that there exist some causal and conditional dependencies between the entities. From the procedural control entities diagram, the user can proceed to editing of state transition diagrams of the entities. This step is performed by clicking on the respective entity nodes. Consequently, the state transition diagram editor is activated (Fig. 20). The state transition diagram editor is similar to the procedural control entities diagram editor. The difference is that instead of entities and dependencies the user edits states and transitions. In each state, several actions are executed. The actions have to be described in structured text in a corresponding window that is invoked when the user clicks on the respective state. An additional window is used for the description of events, conditions and actions associated with state transitions. The state transition diagram in Fig. 20 describes the dynamics of the grinding procedural control entity. Furthermore, to complete the model, the data about system parameters, commands and devices have to be provided by the user. The data are entered by way of using a series of dialog windows.

When editing of the model is completed, code generation procedure can be activated. The result of the code generation procedure is a text file (Fig. 21) describing graphical as well as textual parts of the ladder/function block diagram source code. The actual code that implements the model (Figs. 4 and 7) is obtained by importing the text file into the IEC 61131-3 programming environment. The result is shown in Fig. 22. From this figure we can see that on the highest level of code hierarchy there exist five function blocks, which correspond to procedural control entities of the model. Furthermore, the procedural control entities correspond to the decomposition of the controlled process. Hierarchy and granulation of the controlled process remain preserved in the code—the same abstractions the process engineers use for description of the controlled process are implemented in the PLC code. Seamlessness of the software design process is thus achieved.



```
FUNCTION_BLOCK Grinding
        VAR_EXTERNAL
                GrindingVar: PostVarDut;
        END_VAR
        VAR
                GrindingOperating: GrindingOperating;
                GrindingStopped: GrindingStopped;
                GrindingChecking: GrindingChecking;
                GrindingFastStop: GrindingFastStop;
        END_VAR
'LD'
BODY
    WORKSPACE
        NETWORK_LIST_TYPE := NWTYPELD ;
        ACTIVE_NETWORK := 0 ;
    END_WORKSPACE
    NET_WORK
        NETWORK_TYPE := NWTYPELD ;
        NETWORK_LABEL :=  ;
        NETWORK_TITLE :=  ;
        NETWORK_HEIGHT := 14 ;
        NETWORK_BODY
B(B_VARIN,,GrindingVar.State,6,3,8,5,);
B(B_F,@EQ-2,,8,2,13,6,,?D?D?C);
B(B_VARIN,,StStopping,6,4,8,6,);
B(B_VARIN,,GrindingVar.State,6,9,8,11,);
B(B_VARIN,,StRunning,6,10,8,12,);
B(B_F,@EQ-2,,8,8,13,12,,?D?D?C);
B(B_FB,GrindingOperating,GrindingOperating,23,2,37,5,,?BEN?AENO);
L(13,10,16,10);
L(16,4,16,10);
L(1,7,1,13);
L(1,0,1,7);
L(13,4,16,4);
L(1,0,1,14);
L(16,4,23,4);
        END_NETWORK_BODY
    END_NET_WORK
    NET_WORK
```

**Fig. 21.** A fragment of the generated source code text file.

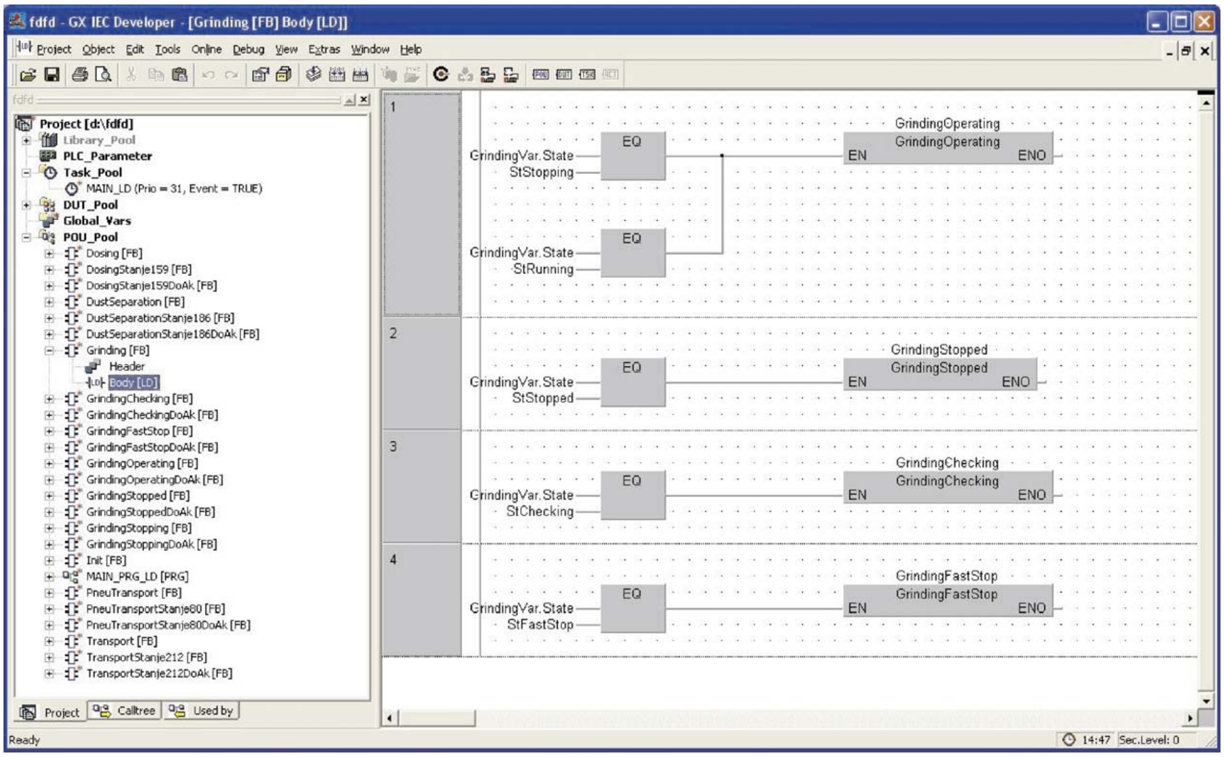


**Fig. 22.** Screenshot of generated function block diagram (FBD) program code.

## 8. Discussion

According to our experience from industrial and laboratory applications, manual conversion of software models to code is a rather tedious and time-consuming process. Automatic code generation, on the other hand, has several advantages over manual programming but also some disadvantages. The most important benefit is that the mapping from the model to the final product (program code) requires no human effort and practically no time.

In order to compare manual and automatic code generation, the CASE tool-assisted development process of the control system described in the previous section was also performed manually. The comparison of the two approaches has shown that even in the case of a simple control system, the effort needed in CASE tool-assisted development is significantly lower and development time shorter.

Another benefit is that due to automation of the mapping process the probability of occurrence of mapping errors decreases radically. As a consequence, software development costs are significantly reduced and at the same time software quality and reliability are improved.

As for disadvantages, one of the difficulties regarding PLC code generation is that in most of the cases the programming environments (languages) provided by various PLC vendors do not strictly comply with the IEC 61131 standard. Consequently, the code generator of the CASE tool must be tailored to a certain PLC vendor.

Another issue is related to the reversibility principle discussed in [28]. A software model does not only have a prescriptive purpose [29], but also serves as part of the control system documentation, which means that it has to be in tune with the resulting program code. A change that is made manually in the program code of the control system should therefore automatically update the underlying model. Automatic model updating is the inverse process of automatic code generation. The former process is much more complex and difficult to implement than the latter. Automatic model updating is not supported in the present version of the CASE tool described in this article. To implement automatic model updating, a stronger coupling/integration of the CASE tool and the PLC programming environment would have to be made. Another way of maintaining the model and the program code in tune is to manage the software development process in a manner that only allows changes on the model level and not on the code level.

## 9. Conclusions

Control software development for programmable logic controllers has become a demanding task due to the ever-increasing complexity of controlled processes and also due to the low abstraction level of PLC programming languages. The programming process is time-consuming as well as extremely error-prone and consequently consumes a great deal of human resources.

It is therefore beneficial to concentrate on solving problems in the domain (process control). This can be effectively done by using domain specific development. Using domain specific languages such as ProcGraph together with an appropriate code generation tool enables the developers to concentrate on solving control problems and not worry about implementation. Coding usually requires a significant share of development time and if it is automated this time can be saved.

## Acknowledgement

The support of the Ministry of Higher Education, Science and Technology of the Republic of Slovenia is gratefully acknowledged.

## References

[1] G. Frey, L. Litz, Formal methods in PLC programming, in: Proceedings of the IEEE Conference on Systems Man and Cybernetics SMC 2000, Nashville, USA, (2000), pp. 2431–2436.
[2] J. Yoo, S. Cha, H.S. Son, C.H. Kim, J.-S. Lee, PLC-based safety critical software development for nuclear power plants, in: Proceedings of the SAFECOMP—International Conference on Computer Safety, Reliability, and Security, Potsdam, Germany, (2004), pp. 155–165.
[3] C.M. Davidson, J. McWhinnie, M. Mannion, Introducing object oriented methods to PLC software design, in: Proceedings of the International Conference and Workshop: Engineering of Computer-Based Systems (ECBS '98), Jerusalem, Israel, (1998), pp. 150–157.
[4] ISA (ANSI/ISA-S88.01.1995), Standard Batch Control. Part 1. Models and Terminology, Instrument Society of America, 1995.
[5] R.W. Lewis, Programming Industrial Control Systems Using IEC 1131-3, The Institution of Electrical Engineers, London, 1998.
[6] Y. Edan, N. Pliskin, Transfer of software engineering tools from information systems to production systems, Computers & Industrial Engineering 39 (1) (2001) 19–34.
[7] F. Bonfatti, G. Gadda, P.D. Monari, Re-usable software design for programmable logic controllers, in: Proceedings of the Workshop on Languages, Compilers & Tools for Real-Time Systems (LCT-RTS 1995), La Jolla, CA, (1995), pp. 31–40.
[8] K. Fischer, B. Vogel-Heuser, UML in der automatisierungstechnischen Anwendung–Stärken und Schwächen, Automatisierungstechnische Praxis 44 (10) (2002) 63–69.
[9] M.F. Zaeh, C. Poernbacher, Model-driven development of PLC software for machine tools, Production Engineering 2 (1) (2008) 39–46.
[10] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison Wesley, Boston, 1999.
[11] C.W. Krueger, Software reuse, ACM Computing Surveys 24 (2) (1992) 131–184.
[12] D.C. Schmidt, Model-driven engineering, Computer 39 (2) (2006) 25–31.
[13] M. Mernik, J. Heering, A.M. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys 37 (4) (2005) 316–344.
[14] M. Marcos Estévez, D. Orive, Automatic generation of PLC automation projects from components-based models, The International Journal of Advanced Manufacturing Technology 35 (5–6) (2007) 527–540.
[15] K. Czarnecki, U.W. Eisenecker, Generative Programming, Methods, Tools, and Applications, Addison Wesley, 2000.
[16] L. Liu, B. Roussev, Management of the Object-Oriented Development Process, IGI Global, 2005.
[17] J. Estublier, G. Vega, A.D. Ionita, Composing domain-specific languages for wide-scope software engineering applications, in: Proceedings of the MoDELS—Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, (2005), pp. 69–83.
[18] J.P. Gray, B. Ryan, Technologies and techniques for rapid CASE tool development, in: Proceedings of the International Database and Applications Symposium, Cardiff, UK, (1998), pp. 188–201.
[19] D. Troy, R. McQueen, An approach for developing domain specific CASE tools and its application to manufacturing process control, Journal of Systems and Software 38 (2) (1997) 165–192.
[20] D. Barstow, Domain specific automatic programming, IEEE Transactions on Software Engineering 11 (11) (1985) 1321–1336.
[21] G. Godena, ProcGraph: a procedure-oriented graphical notation for process-control software specification, Control Engineering Practice 12 (1) (2004) 99–111.
[22] R. Wieringa, A survey of structured and object-oriented software specification methods and techniques, ACM Computing Surveys 30 (4) (1998) 459–527.
[23] F. Wagner, R. Schmuki, T. Wagner, P. Wolstenholme, Modeling Software with Finite State Machines, Auerbach Publications, 2006.
[24] D. Harel, Statecharts:, A visual formalism for complex charts, Science of Computer Programming 8 (3) (1987) 231–274.
[25] D. Harel, H. Lachover, A. Namaad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. Trakhtenbrot, STATEMATE: A working environment for the development of complex reactive systems, IEEE Transactions on Software Engineering 16 (4) (1990) 403–413.
[26] M. Glinz, Statecharts for requirements specification—as simple as possible—as rich as needed, in: Proceedings of the ICSE Workshop Scenarios and state machines: models, algorithms, and tools, Orlando, FL, 2002.
[27] G. Kandare, G. Godena, S. Strmčnik, A new approach to PLC software design, ISA Transactions 42 (2) (2003) 279–288.
[28] R.F. Paige, J.S. Ostroff, P.J. Brooke, Principles for modeling language design, Information and Software Technology 42 (10) (2000) 665–675.
[29] J. Ludewig, Models in software engineering—an introduction, Software and Systems Modeling 2 (1) (2003) 5–14.

**Gregor Kandare** received his B.Sc. and Ph.D. degree in electrical engineering from the Faculty of Electrical Engineering, University of Ljubljana in 1997 and 2004, respectively. He is currently a postdoctoral researcher at the Department of Systems and Control at the Jozef Stefan Institute. His main research interests are software specification techniques and software engineering methods for process control.

**Stanislav Strmčnik** received his B.Sc., M.Sc., and Ph.D. from the Faculty of Electrical Engineering, University of Ljubljana in 1972, 1975 and 1979, respectively. Since 1986 he has been head of Department of Systems and Control at the same institute. His research interests concern mathematical modeling, identification, process control and non-technical aspects of automation. He is author and co-author of numerous papers and has taken part in many domestic and international research and development projects. He is also an associated professor at the Faculty of Electrical Engineering, University of Ljubljana and University of Nova Gorica.

**Giovanni Godena** is a senior software engineer at the Jozef Stefan Institute, Department of Computer Automation and Control. He has 20 years experience in analysis and design of real-time software for process control. He has authored, or co-authored, some 20 journal and conference papers and book chapters on software in computer automation. His current research interests include software engineering methodologies and automated software reusability in computer automation. He is a member of the IEEE Computer Society. He received a B.Sc. degree in control engineering in 1984 from the University of Ljubljana, Slovenia.