Development of process control software using software engienering techniques

Gregor Kandare

Department of Systems and Control
Jozef Stefan Institute
Jamova 39, 1000 Ljubljana
Slovenia

E-mail: `gregor.kandare@ijs.si`

Abstract: In the paper a model-based automated approach to procedural process control software is presented. A domain-specific modelling language specialised for analysis and design of procedural process control software is described. A formal description of the language syntax is necessary in order to define a mapping function from models to programme code. Furthermore, a software modelling tool is described that supports editing of software models and automatic source code generation for programmable logic controllers as well as automatic documentation generation.

Keywords: Process control, Software specification, Software engineering, Programmable logic controllers.

1. INTRODUCTION

Software for control systems is one of the most problematic types of software. The reason lies in the nature of control systems, which are designed to control machines, devices and processes. Reliability, safety and real-time reactions are therefore among the most important attributes of this kind of software ([1]).

On the other hand, modern control systems cover an ample set of functions and encompass a wide array of hardware devices ranging from simple sensors, actuators and controllers to complex computer systems. Consequently, control software is becoming ever more complex and consequently 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 very important issues also in the field of control systems ([2]).

A very common way of coping with some of the issues mentioned above, which is used by software providers and engineering companies in this field, is to employ the concept of reusability. In simplified terms, this means relying on a system of well-tested and field-proven preprogrammed 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 so-called *basic control* functions (according to ISA S88 standard, [3]), which are primarily dedicated to establishing and maintaining a specific state of process equipment and process variables, and are the same or very similar in entirely different processes.

In contrast to basic control functions, the so-called *procedural control* activities, which consist of various sequences of events, transitions of states, starting, stopping, etc., are much more problematic from the software development point of view. The procedural part of control is entirely process-specific and requires the development of custom software. Improvement of procedural control software development process is therefore quite an important task.

Implementation of procedural control functions (as well as basic control functions) is nowadays largely based on *programmable logic controllers* (PLCs). The main issue, therefore, is how to improve the PLC software development process.

The aim of this article is thus 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 article, the transition between the final phases of software development, namely the translation of software models into 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 Section 2, a modular structure of a continuous process control system is defined. Furthermore, in Section 3, some basic issues related to the PLC software development process are discussed. Section 4 presents the modeling language ProcGraph together with its formal description. In Section 5, the technique of mapping ProcGraph models into IEC 61131-3 source code is presented. Automatic code generation process is described in Section 6. In Section 7, an example of application of the modeling tool is given.

2. CONTINUOUS PROCESS CONTROL STRUCTURE

Formalization of process control structure is on the one hand necessary in order to be able to cope with complexity of the control system by division of control functions. On the other hand it is absolutely 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 in order to simplify and standardize the process of design of this kind of systems and software. The design of batch process control systems and their organization is for example well covered by the ISA S88 standard. There also exist various software tools for batch process control. 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.

In accordance with the ISA S88 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 Figure 1.

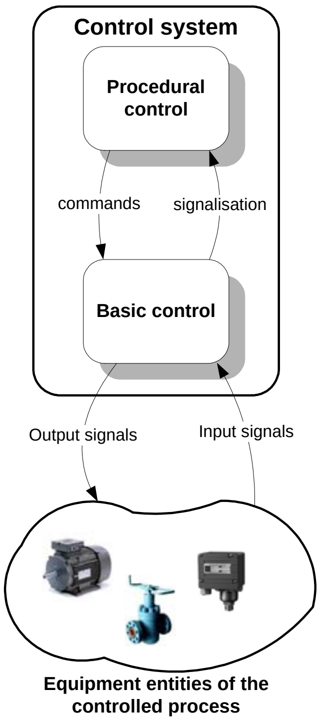


Figure 1: Control system of a continuous process

By partitioning the control system, its modularity is achieved. The modularity yields easier analysis, design, implementation, use and maintenance of the system. At the same time, the way of looking upon the process becomes similar to the view of the domain expert (process engineer). If the control system developer and domain expert use the same abstractions, the communication between them is easier and less prone to misunderstanding.

Basic control is dedicated to establishing and maintaining a specific state of the process equipment. Devices used in basic control are controllers, programmable 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, which in some ways reflects the domain expert's view of the system. 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 to handle these situations depends predominantly on the requirements of the controlled process. Procedural control entities are therefore more unique than basic control entities.

As we can see from Figure 1, the procedural control receives signalisation from the basic control and sends commands to the basic control. The commands depend on the input signals and the current state of the procedural control. This situation is typical of so-called reactive systems. For that reason we can consider the procedural control system to be a reactive system.

3. PLC SOFTWARE DEVELOPMENT PROCESS

To perform basic and procedural process control, programmable logic controllers are most frequently used. The process of PLC software development is not a straightforward task, due to the high demands for control software quality and reliability. Furthermore, programming languages, proposed by the IEC 61131-3 standard ([4]), provide a relatively low level of abstraction. This aspect makes the aforementioned programming languages unsuitable for complex systems.

Several solutions for tackling the problems of PLC software development have been proposed in the literature ([1], [2], [5], [6], [7]). 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, [8]). In the field of procedural control software, however, the drawback of using UML or similar standard modeling languages is that they are too general and not sufficiently domain-specific. The use of very general modeling languages increases *cognitive distance* ([9]), which is a measure of the effort needed to progress from one phase of software development to the next. In order to reduce the cognitive distance, it is preferable that similar abstractions be used throughout all phases of software development, which can be realized by using a domain-specific modeling language.

Another important 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 the 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, a very important 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 general modeling languages. The majority of these 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 mere mapping of architectural diagrams such as UML class diagrams to static code constructs and modules.
If we want to use computer automated design tools in specific domains such as the development of procedural control software, we have two options ([10]). 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 programming languages are not supported.
Alternatively, we can develop a domain-specific CASE tool ([11]) that supports domain-specific modeling languages as well as target programming languages. The second alternative requires much more initial effort but proves to be more efficient in the long run. Synthesis of domain-specific code is namely more efficient than synthesis of generic code ([12]). In this article, the second alternative is presented using an earlier-developed modeling language, referred to as ProcGraph, as the key element.
In order to be able 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 overview of ProcGraph syntax.

4. FORMAL DESCRIPTION OF THE PROCGRAPH MODELLING LANGUAGE

ProcGraph is a specialized modeling language for the design of procedural process control software. A more detailed description of the language can be found in [13]. The advantage of ProcGraph 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 (*subject domain-oriented decomposition*, [14]). In doing so, and with the use of an appropriate target programming language, seamlessness of the software development process can be achieved. Seamless transition between the development phases means that less effort is needed for the transitions – the cognitive distance is smaller ([9]). Seamlessness has many benefits, one of which is that communication between the specialists in the software development process (process engineers, system modelers and programmers) is simplified, because they are communicating in the same terms. Furthermore, reverse transitions between software development phases are easier. It is well known that the software development process is not a linear and reversible one. It often occurs that a change made in a certain development phase necessitates modifications in preceding phases as well. If, for example, something is changed in the program code, the model has to be readapted too. This is simplified by using the same abstractions throughout all development phases.
ProcGraph models consist of three types of diagrams, each describing a particular view of the system to be built:

- *Procedural control entities diagram* (PCED) depicts the concurrent and at the same time 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).

Procedural control entities diagram

Procedural control entities diagram depicts the architectural view of the control software system. Nodes of the PCED represent conceptual components – procedural control modules. Vertices in the PCED portray relationships (dependencies) between the procedural control entities. An example of a procedural control entities diagram is shown in Figure 2.

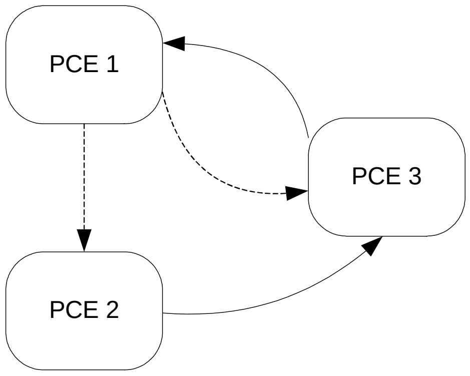


Figure 2. Procedural control entities diagram of a control system of a continuous process

Two types of connections are allowed:

- 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. Figure 3 illustrates a graphical depiction of procedural control entities diagram syntax using UML Class diagrams. The figure shows that each node has a name and can be connected to one or more other nodes. Each connection has a type (full or dashed).

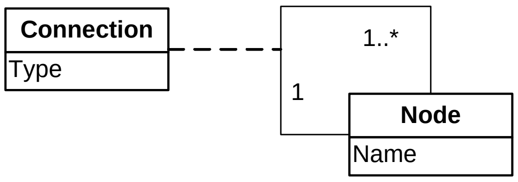


Figure 3. Syntax of procedural control entities diagrams

Another way of describing the syntax of procedural control entities diagrams is by using the 6-tuplet (1), where V_E is a set of nodes, P_E a set of connections, ρ 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.

$$\langle PCED \rangle = \{V_E, P_E, \rho, iz, po, tp\} \tag{1}$$

State transition diagram

As we have asserted earlier in this article, procedural control systems can be classified as reactive systems. The most appropriate model for describing the behavior of such systems is the *state transition diagram* (STD) ([15], [16], [17]). 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 so-called *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. This hierarchical structure also enables us to simplify the stepwise design of the system.

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

```
event[condition]/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. Figure 4 shows an example of an extended state transition diagram.

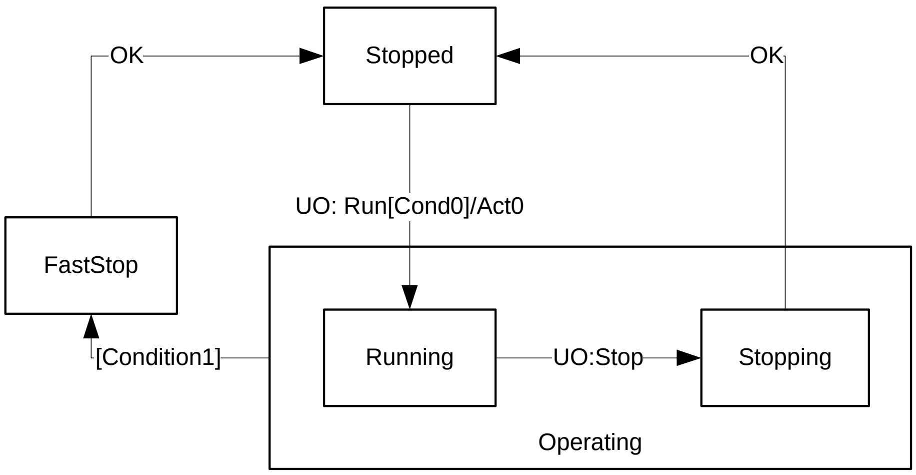


Figure 4. A sample extended state transition diagram

The state hierarchy of the diagram in Figure 4 is represented by the tree in Figure 5. The root node represents the entire state transition diagram. Basic states are represented by leaf nodes and composite states by subtrees. In Figure 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.

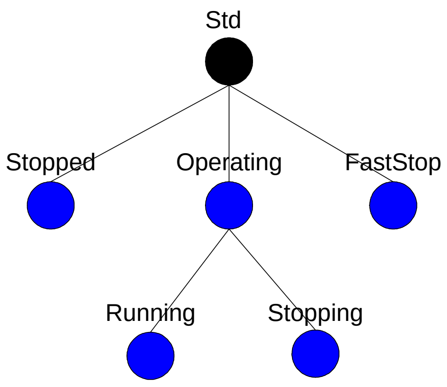


Figure 5. State hierarchy of the diagram in Figure 4

Figure 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 other nodes.

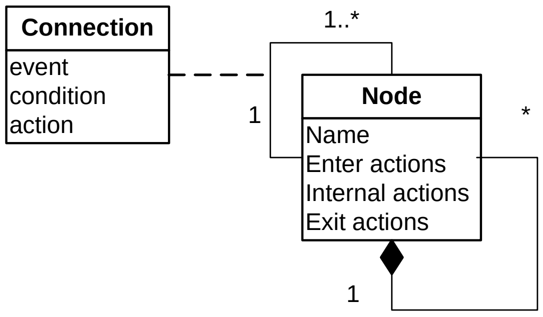


Figure 6. Syntax of the state transition diagrams

Syntax can also be described by the 9-tuple (2).

$\langle STD \rangle = \{ V_S, P_S, iz, po, nad, \mathtt{I}, \mathtt{P}, \mathtt{O}, \lambda\}$ (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, $\mathtt{I}$ a set of input signal descriptions, $\mathtt{P}$ a set of parameter descriptions, $\mathtt{O}$ a set of descriptions of output signals and λ 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.

Entity dependancy diagram

An entity dependancy 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. Whereas 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.

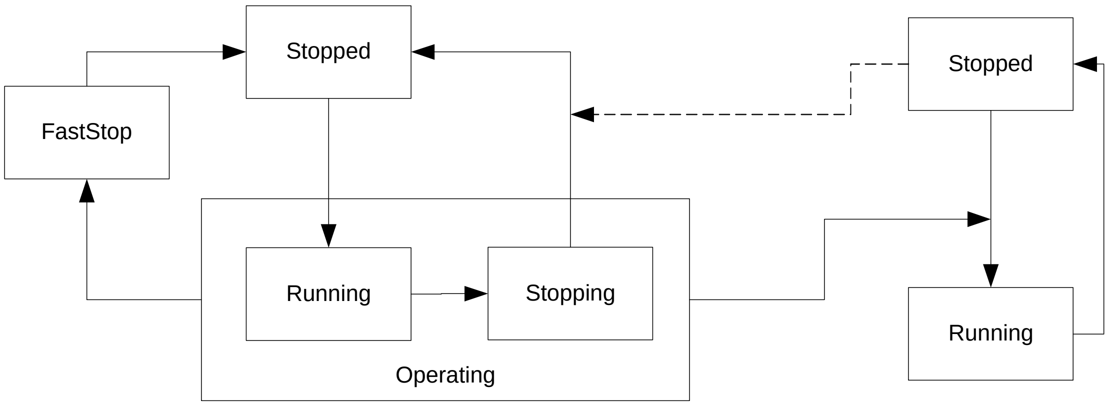


Figure 7. A sample entity dependency diagram

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

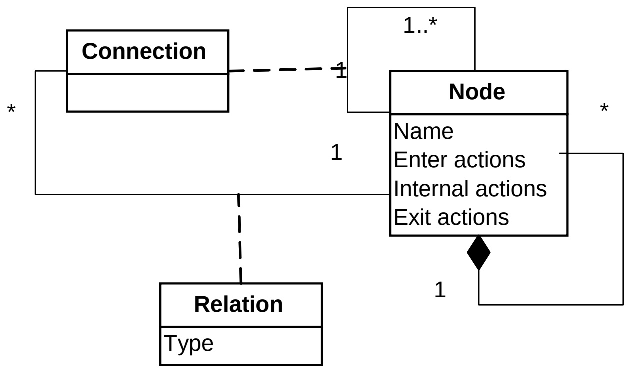


Figure 8. Syntax of the entity dependency diagrams

The syntax can be described by the following 8-tuple:

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

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.

5. MAPPING OF MODELS TO SOURCE CODE

In order to achieve seamless transitions between the development phases of software product, the abstractions (entities) used in the models of the separate phases have to be as similar as possible. In this manner, the effort needed to advance from one phase to another is minimized.

As we have seen in Section 4, 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. In regard to the freedom of choice of models and abstractions, we are not as free in this phase as we are in the modeling phase. While in the modeling phase we can design and adapt the modeling language so that it suits our needs, this is not feasible in the programming phase. The "model" which results from the programming phase is actually program source code in one of the programming languages. Programming languages have fixed syntax and semantics that cannot be altered 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 the ones defined by the IEC 61131-3 standard ([4]). 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 are an extension of Ladder diagram, which was the first language used in PLC programming. Programs in Ladder diagram look a lot like electrical schemes with contacts and coils. Beside those elements, the main components of FBD programs are function blocks, which are "wired" together. Thus, function blocks can be imagined as a kind of integrated circuits.

For the description of function block diagram syntax, the following triplet can be used:

$$\langle FBD \rangle = \{\mathsf{B}, \mathsf{L}, \mathsf{V}\}, \qquad (4)$$

where:

- B represents a set of function blocks,
- L a set of variables and
- V a set of connections.

Furthermore, the syntax of Function block diagram language is depicted in Figure 9.

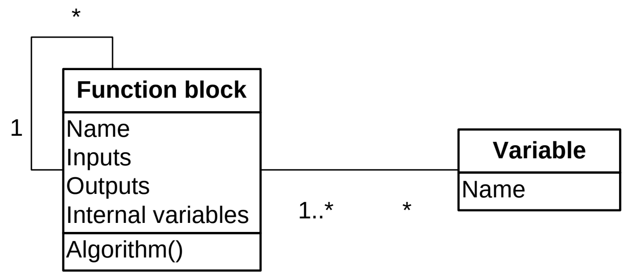


Figure 9. Syntax of Function block diagrams

The syntax definition in Figure 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.

Definition of the mapping function

As we have seen in Section 2, a ProcGraph model consists of three different types of submodels (diagrams), which can be described by the following equation:

$\langle ProcGraph\rangle = \langle PCED\rangle + \langle STD\rangle + \langle EDD\rangle.$ (5)

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

$C_G: \langle ProcGraph\rangle \rightarrow \langle FBD\rangle.$ (6)

The expression (6) is general and describes the transformation of the entire ProcGraph model into function block diagram source code.
Respecting the seamlessness principle, the nodes of ProcGraph models are mapped to function blocks. As we have seen in Section 4, the nodes of ProcGraph models represent procedural control entities in procedural control entities diagrams and states in state transition diagrams. Figure 10 shows an example of how two entities of a procedural control entities diagram map to corresponding function blocks in Function block diagram code.

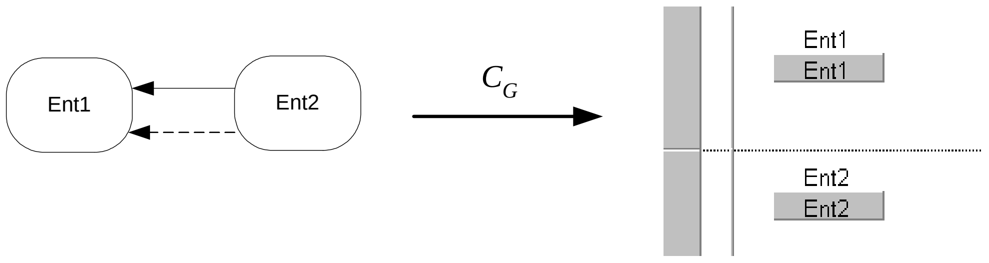


Figure 10. Mapping of procedural control entities diagrams

Each entity in the model corresponds with a state transition diagram, which describes its dynamics. For example, the dynamics of the entity *Ent1* in Figure 10 is described by the state transition diagram in Figure 4. Therefore, the algorithm of each function block depicting an entity is an implementation of the entity's state transition diagram. Figure 11 shows the implementation code of the state transition diagram shown in Figure 4.

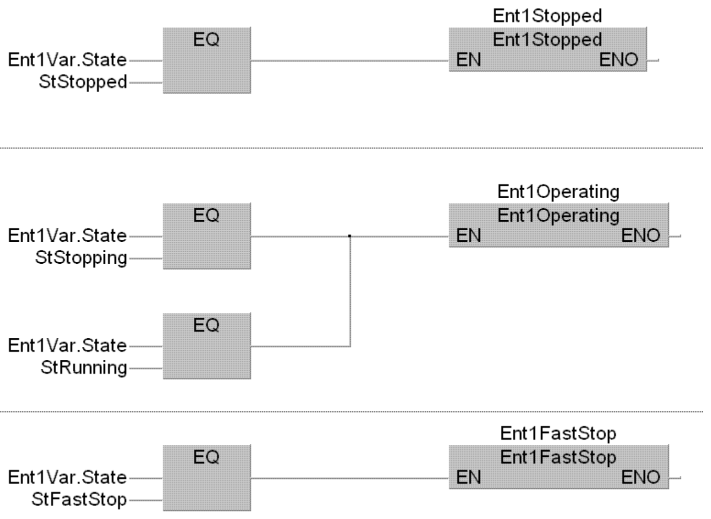


Figure 11. Mapping of state transition diagrams

In order to ensure the seamlessness principle, the hierarchy of programe code elements (function blocks) is equivalent to the hierarchy of model elements (procedural control entities and states). In Figure 11, consequently, there appear only function blocks depicting the states of the highest hierarchy level of STD from Figure 4. The substates are implemented by function blocks that are called (activated) at a corresponding level of function block hierarchy. The *Operating* state of the STD in Figure 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 depict the

Running and *Stopping* states (Figure 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. Besides the function blocks representing the substates, the code in Figure 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.

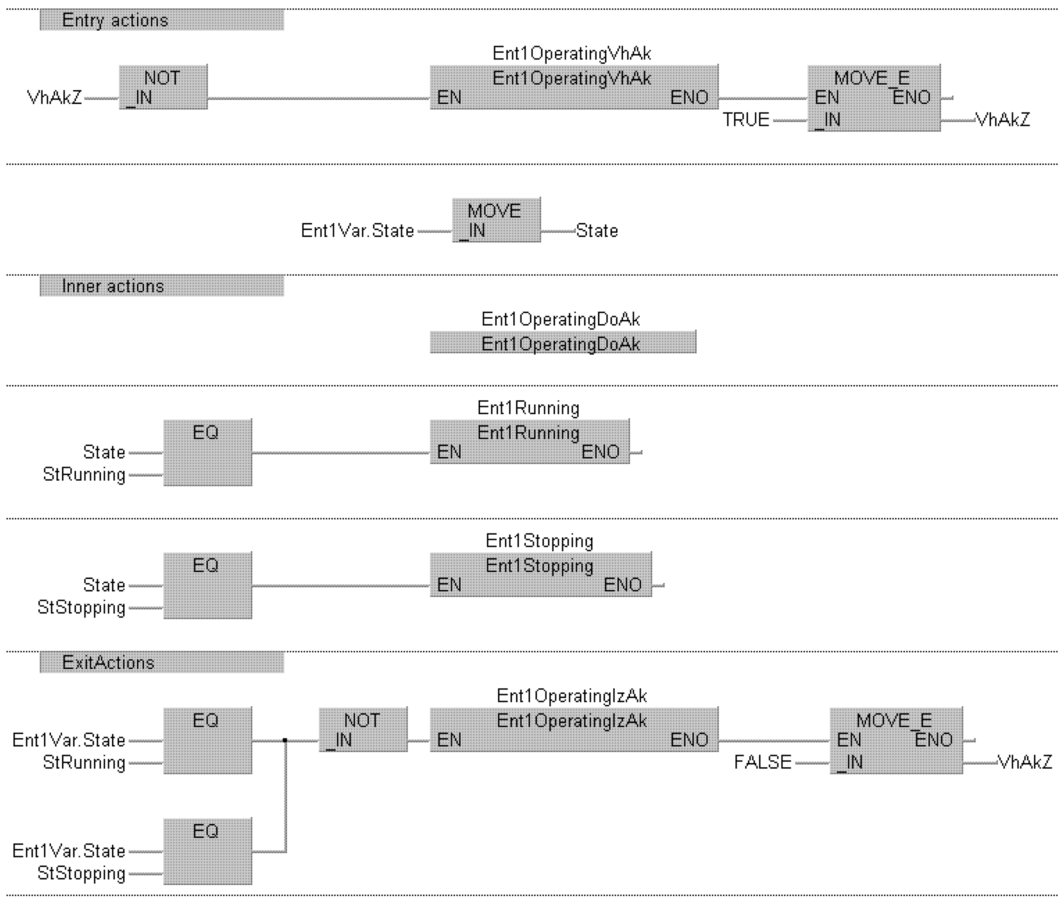


Figure 12. Algorithm of the *Ent1Operating* function block

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 such names 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 have realized that manual programming is a very 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 programming manpower can be reduced and development times can be significantly shortened. Furthermore, the amount of errors is decreased.

Automatic code generators resemble compilers – they both transform 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 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 Figure 13.

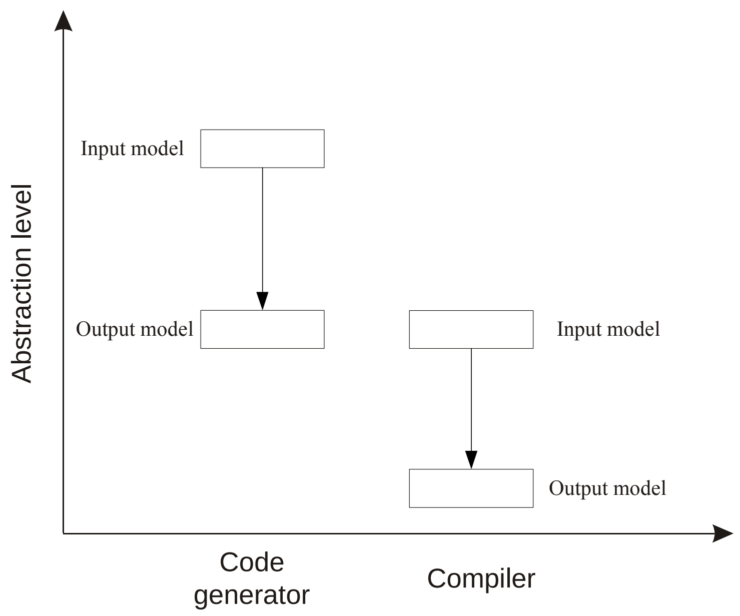


Figure 13. Code generator and compiler abstraction levels

7. APPLICATION OF A SOFTWARE MODELLING TOOL

In order to demonstrate the use of the modeling tool a control system of an ore grinding process is employed ([18]). The process (Figure 14) is 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 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.

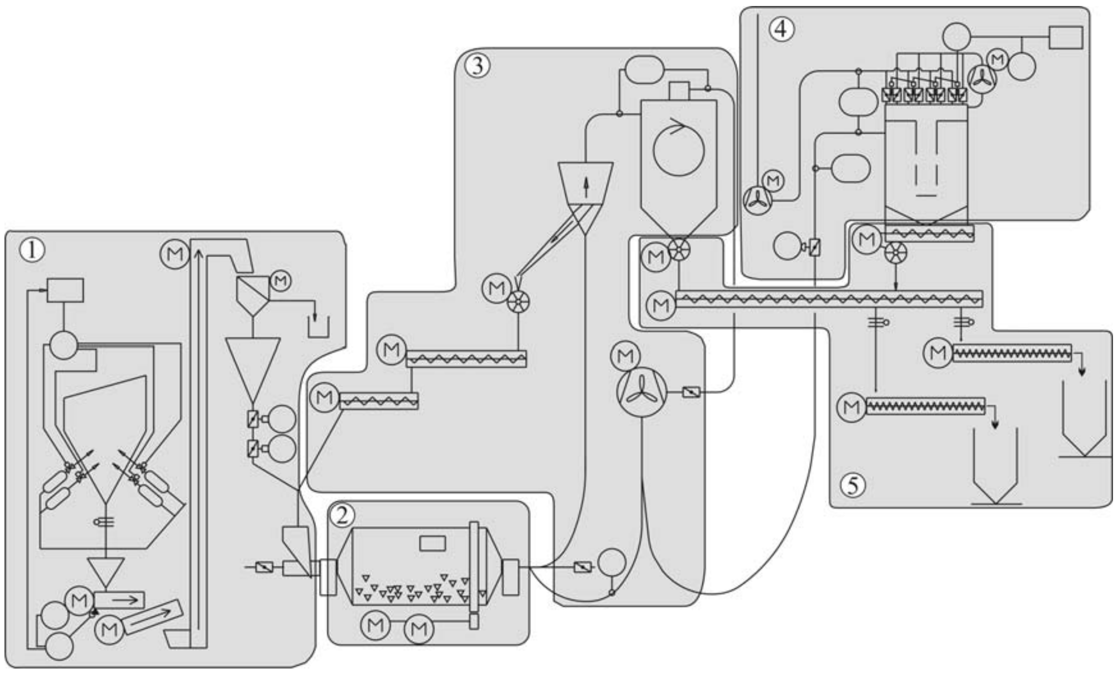


Figure 14. Ore grinding process

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 following five procedural control entities: Dosing, Grinding, Pneumatic transport, Dust separation and Silo transport.

The screenshot in Figure 15 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 mentioned above. Connections between the nodes in Figure 15 indicate that there exist some causal and conditional dependencies between the entities.

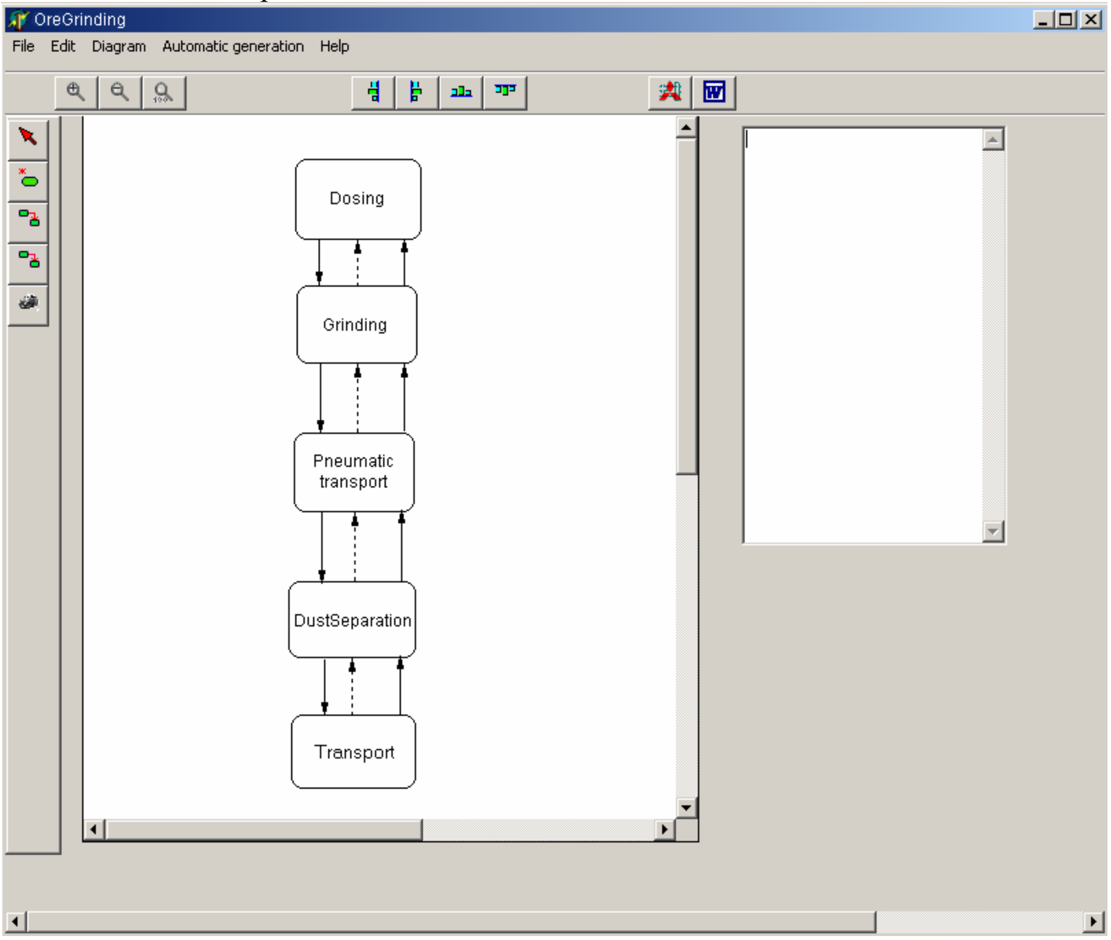


Figure 15. Screenshot of the main window of the modeling tool

From the procedural control entities diagram, the user can proceed to editing of the 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 (Figure 16).

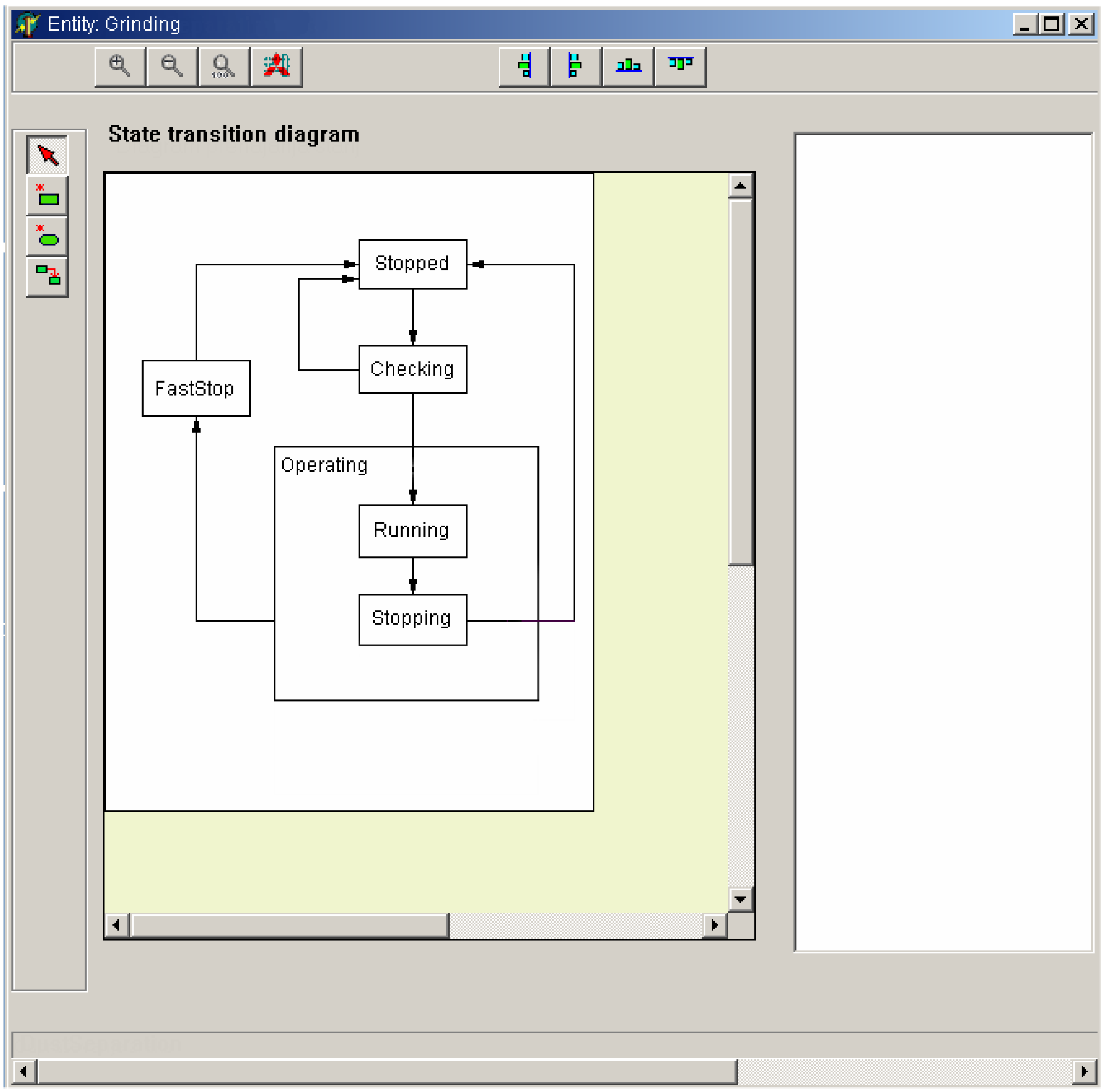
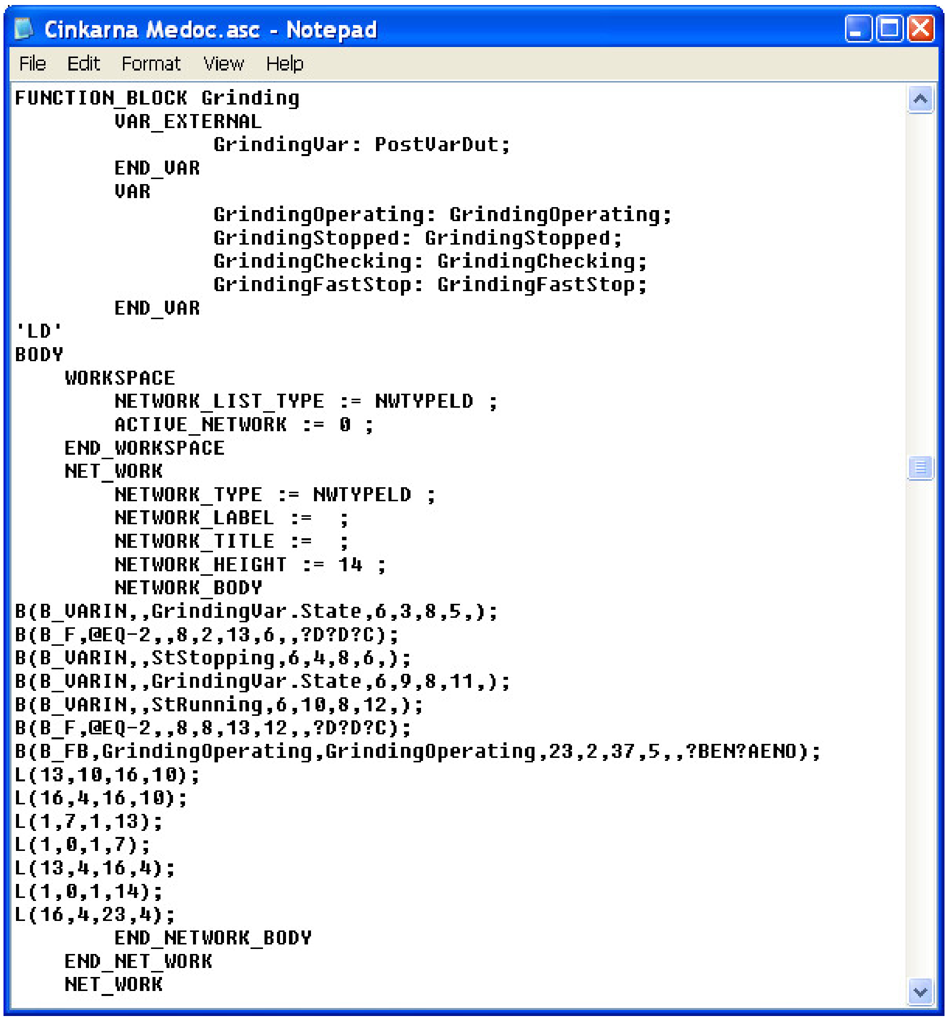


Figure 16. State transition diagram editor

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, a number of actions are executed. The actions have to be described in structured text in a special 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 Figure 16 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. This is done by way of using a series of window dialogs.
When editing of the model is completed, code generation procedure can be activated. The result of the code generation procedure is a text file (Figure 17) describing graphical as well as textual parts of the FBD source code.



```
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
```

Figure 17. A fragment of the generated text file

The actual FBD code that incarnates the model (Figure 4, Figure 7) is obtained by importing the text file into the IEC 61131-3 programming environment. The result is shown in Figure 18. 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.

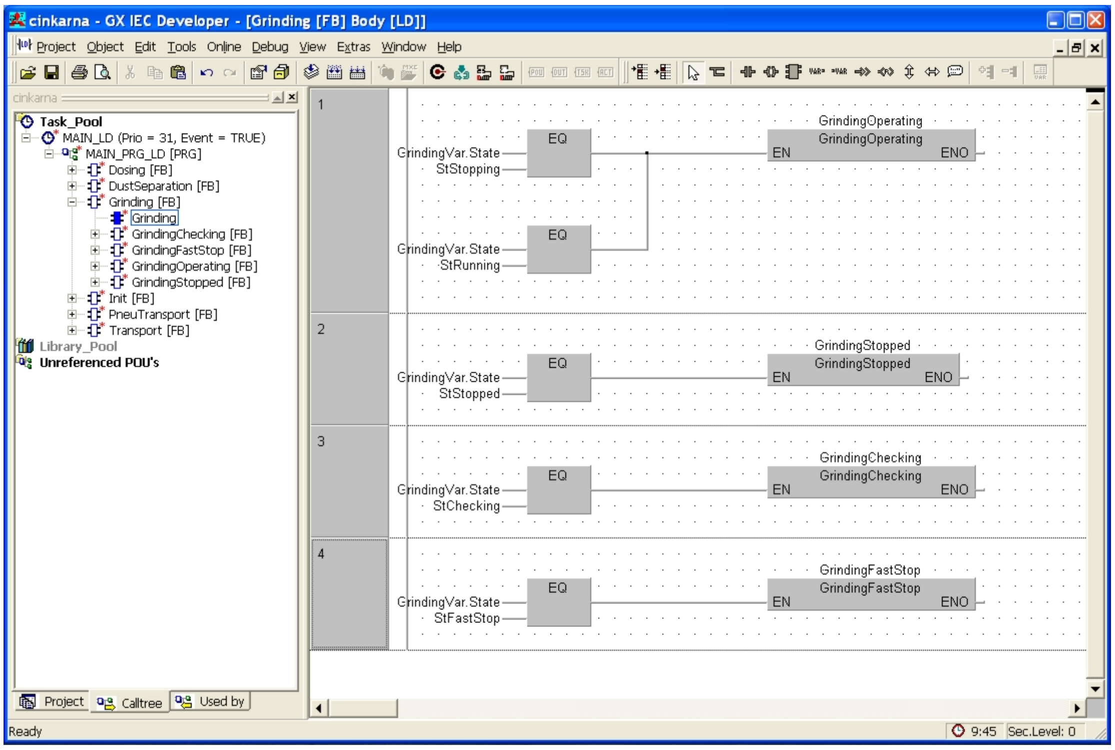


Figure 18. Example of generated FBD program code

8. CONCLUSION

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 manpower resources. In this article we show that the rules of the model-to-program code conversion can be precisely defined and hence automated. This can be done by implementing a domain-specific code generator (synthesizer). The code generator uses code patterns, which also contributes to the standardization and reusability of the generated code. In the code generation process, the appropriate patterns are used and filled with the corresponding content.

REFERENCES

[1] G. Frey and L. Litz, "Formal methods in PLC programming," in *Proc. IEEE Conference on Systems Man and Cybernetics SMC 2000*, Nashville, 2000, pp. 2431-2436.

[2] C. M. Davidson, J. McWhinnie and M. Mannion "Introducing Object Oriented Methods to PLC Software Design," in *Proc. International Conference and Workshop: Engineering of Computer-Based Systems (ECBS '98),* Jerusalem, Israel, 1998, pp. 150-157.

[3] ISA (ANSI/ISA-S88.01.1995), *Standard Batch Control; Part 1: Models and Terminology*, Instrument Society of America, 1995.

[4] R.W. Lewis, "Programming industrial control systems using IEC 1131-3," London, UK: The Institution of Electrical Engineers, 1998.

[5] Y. Edan and N. Pliskin, “Transfer of Software engineering Tools from Information Systems to Production Systems,” *Computers & Industrial Engineering*, vol. 39, no.1, pp. 19-34, 2001.

[6] F. Bonfatti, G. Gadda and P.D. Monari, “Re-usable Software Design for Programmable Logic Controllers,” in *Proc. Workshop on Languages, Compilers & Tools for Real-Time Systems (LCT-RTS 1995),* La Jolla, California, 1995, pp. 31-40.

[7] K. Fischer and B. Vogel-Heuser, “UML in der Automatisierungstechnischen Anwendung – Stärken und Schwächen,” *Automatisierungstechnische Praxis,* vol. 44, no. 10, pp. 63-69, 2002.

[8] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide*, Boston: Addison Wesley, 1999.

[9] C. W. Krueger, “Software reuse,” *ACM Computing Surveys*, vol. 24, no. 2, pp. 131-184, 1992.

[10] J. P. Gray and B. Ryan “Technologies and Techniques for rapid CASE tool development,“ in *Proc. International Database and Applications Symposium*, Cardiff, UK, 1998, pp. 188-201.

[11] D. Troy and R. McQueen, “An Approach for Developing Domain Specific CASE Tools and Its Application to Manufacturing Process Control,” *Journal of Systems and Software,* vol. 38, no. 2, pp. 165-192, 1997.

[12] D. Barstow, “Domain specific automatic programming,” *IEEE Transactions on Software Engineering*, vol. 11, no. 11, pp. 1321-1336, 1985.

[13] G. Godena, “ProcGraph: a procedure-oriented graphical notation for process-control software specification,” *Control Engineering Practice*, vol. 12, no. 1, pp. 99-111, 2004.

[14] R. Wieringa, “A Survey of Structured and Object-Oriented Software Specification Methods and Techniques,” *ACM Computing Surveys*, vol. 30, no.4, pp. 459-527, 1998.

[15] D. Harel, “Statecharts: A Visual Formalism for Complex Charts,” *Science of Computer Programming*, vol. 8, no. 3, pp. 231-274, 1987.

[16] D. Harel, H. Lachover, A. Namaad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring and M. Trakhtenbrot, “STATEMATE: A Working Environment for the Development of Complex Reactive Systems,” *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 403-413, 1990.

[17] M. Glinz, “Statecharts for Requirements Specification – As Simple As possible - as Rich as Needed,” in *Proc. ICSE Workshop Scenarios and state machines: models, algorithms, and tools*,” Orlando, Florida, 2002.

[18] G. Kandare, G. Godena and S. Strmčnik, “A new approach to PLC software design,” *ISA Transactions*, vol. 42, no. 2, pp. 279-288, 2003.

[19] R. F. Paige, J.S. Ostroff and P.J. Brooke, “Principles for modeling language design,” *Information and Software Technology*, vol. 42, no. 10, pp. 665-675, 2000.

[20] J. Ludewig, “Models in Software Engineering – an Introduction,” *Software and Systems Modeling*, vol. 2, no. 1, pp. 5-14, 2003.