8th International IFAC Symposium on
Dynamics and Control of Process Systems
Preprints Vol.2, June 6-8, 2007, Cancún, Mexico

COMPUTER AIDED DESIGN OF PROCEDURAL PROCESS CONTROL SOFTWARE

Gregor Kandare and Stanko Strmčnik

J. Stefan Institute, Jamova 39, SI-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. *Copyright © 2007 IFAC.*

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

1. INTRODUCTION

Process control software is becoming ever more complex and difficult to develop and maintain. The reason lies in the very nature of control systems, which are designed to control machines, devices and processes. If the software is not appropriate, those machines, devices, processes, etc. can go out of control and can cause big material damage or even ecological disasters. Reliability, safety and real-time reactions are therefore among the most important attributes of this kind of software (Frey and Litz, 2000).

A further issue is complexity of control software systems. In order to cope with complexity and ensure software quality, a systematic approach to the software development has to be undertaken. In the field of business software development, a software engineering approach with the use of modelling languages and tools has successfully been used for several years and software engineering has developed to a mature technical discipline (DeMarco, 1978; Wieringa, 1998; Booch, *et al.*, 1999; Ludewig, 2003). In the process control software projects, the development of software in many cases still begins with programming, without previous modelling. The problem is that the programming languages for process control devices such as programmable logic controllers (Lewis, 1998) do not reach an adequate level of abstraction and are therefore unsuitable for very complex systems. However, some reports of the use of software engineering methods in process control software development can also be found in the literature (Davidson, *et al.*, 1998; Edan and Pliskin, 2001). Nevertheless, a shortfall exists in the offer of computer automated software engineering tools (CASE) for process control software development, especially tools that automatically generate source code for process control devices.

The paper is organised as follows. In the next section, the ProcGraph modelling language is presented. In the paper only a short definition of ProcGraph syntax is given. A detailed description of formal syntax as well as semantics can be found in (Kandare, 2004). Afterwards, the description of mapping procedure of models into source code is given. In the subsequent section, a software modelling tool that enables model building and automatic code as well as documentation generation is presented. After that, an example of application of the modelling tool in development of software for an industrial process control system is described. Finally, some conclusions are drawn.

2. THE PROCGRAPH MODELLING LANGUAGE

ProcGraph is a specialized modelling language for the design of procedural process control software (Kandare, 2004; Godena, 2004). 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, which corresponds to subject domain-oriented decomposition (Wieringa, 1998). 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 (Krueger, 1992). Seamlessness has many benefits, one of which is that communication between the specialists in the software development process (process engineers, system modellers 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 irreversible one. It often occurs that a change made in a certain development phase requires 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.

The ProcGraph modelling language encompasses the following types of diagrams:

- *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 (behaviour) of conceptual components – procedural control entities.
- *Entity dependency diagram* (EDD) portrays causal and conditional dependencies between conceptual components (procedural control entities).

2.1 Procedural control entities diagram - PCED

Procedural control entities diagram depicts the concurrent procedures of the control system. At the same time, this represents an architectural decomposition of the control software. 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 Fig. 1.

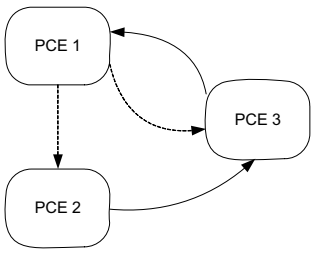

Fig. 1. Procedural control entities diagram of a procedural control system.

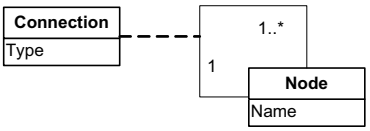

Fig. 2. Syntax of procedural control entities diagram.

Syntax model of procedural control entities diagram (<*PCED*>) 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 the syntax description it can be seen that each node has a name and can be connected to one or more other nodes. Each connection has a type (full or dashed).

The syntax can also be defined using the following six-tuplet:

$$<PCED> = \{V_E, P_E, \rho, iz, po, tp\} \quad (1)$$

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. A comprehensive formal model of ProcGraph syntax and semantics is described in (Kandare, 2004)

2.2 State transition diagram - STD

Procedural control systems produce real-time reactions to output stimuli, therefore they can be characterised as reactive systems. The most appropriate model for describing the behaviour of such systems is the *state transition diagram* (Harel, 1987). Fig. 3 shows a sample state transition diagram representing the dynamics of procedural control entity PCE 3 from Fig. 1.
Syntax of state transition diagrams (<*STD*>) is depicted in Fig. 4. Each node representing a state can

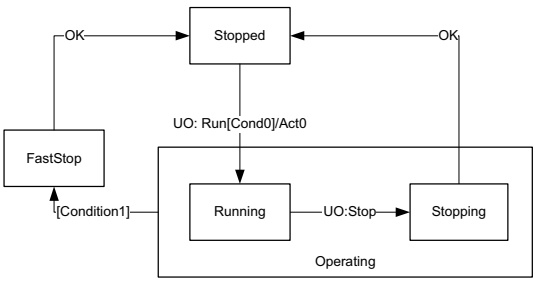

Fig. 3. A sample state transition diagram.

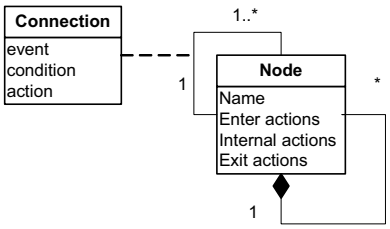

Fig. 4. Syntax of state transition diagrams.

be connected to one or more other nodes with connections representing state transitions. Each connection is adorned with a description of the event that fires the transition, the condition which has to be fulfilled and the action that is executed when the transition fires. Furthermore, nodes can contain other nodes, thus building a state hierarchy.

Another way of defining the STD syntax is by using the expression

$$\langle STD \rangle = \{V_S, P_S, iz, po, nad, I, P, O, \lambda\} \qquad (2)$$

V_S is a set of nodes representing the states, P_S a set of connections depicting the transitions, *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 λ a function that attributes a logical expression to each transition from the set P_S. Input and output signal descriptions are used in logical expressions that define transition event and condition, as well as the actions executed when a corresponding transition is activated.

2.3 Entity dependency diagram (EDD)

Entity dependency diagrams contain 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 a full arrow, while a causal relationship is indicated by a dashed arrow.

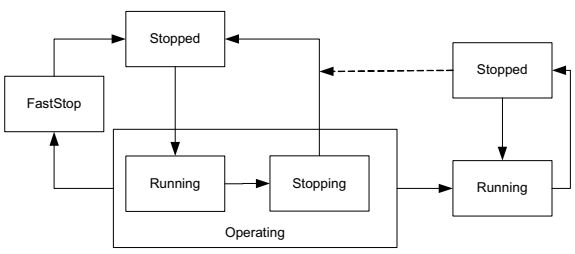

Fig. 5. A sample entity dependency diagram.

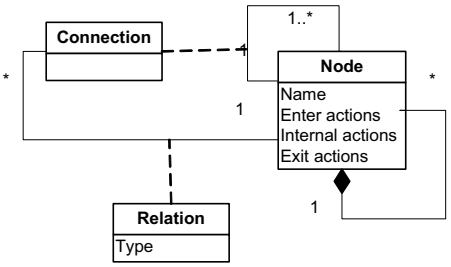

Fig. 6. Syntax of entity dependency diagrams.

Fig. 5 shows an EDD depicting state transition diagrams and dependencies of procedural control entities PCE 3 and PCE 1.

Fig. 6 shows the syntax model of entity dependency diagram syntax. The syntax is similar to the syntax of state transition diagrams. The difference is that in addition to connections between nodes there also exist connections (representing dependencies) that originate in nodes and sink in connections.

The expression

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

also describes the syntax of entity dependency diagrams. Compared to the expression (2), the expression (3) contains a set of conditional relationship connections (Q) and functions *izp* and *pop* that return the connection source node and sink transition connection, respectively. On the other hand, EDD does not contain the information about transition events, conditions and actions as this information is already included in the STD.

3. MAPPING OF PROCGRAPH MODELS TO SOURCE CODE

In the software development process, the modelling phase is followed by the programming (coding) phase. It is preferable to have similar abstractions and maintain their hierarchy throughout the whole development process. ProcGraph is designed to reflect the abstractions of the problem domain. In this article the focus is put on programmable logic controllers as the implementation devices for procedural process control. The destination language therefore has to be one of the PLC programming languages. Considering the seamlessness issue, the most appropriate language for this purpose seems to be the Function block diagram (FBD), which uses function blocks as language constructs that are similar to the main ProcGraph elements such as procedural control entities and states. Furthermore, FBD allows hierarchical decomposition of programs (the body of a function block can contain code composed of other function blocks). Thus, the hierarchy of the ProcGraph models can remain preserved in the FBD code. To transform ProcGraph

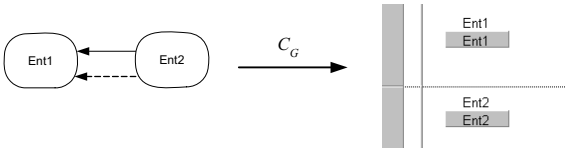

Fig. 7. Mapping of procedural control entities.

models into source code, exact mapping rules based on the ProcGraph and programming language syntax and semantics have to be defined.

ProcGraph modelling language consists of three different types of diagrams, which can be described with the following expression:

$$<ProcGraph> = <PCED> + <STD> + <EDD> \quad (4)$$

The mapping function of ProcGraph models into Function block diagram language can be written as follows:

$$C_G: <ProcGraph> \rightarrow <FBD> \quad (5)$$

Each node of procedural control entities diagram is mapped to its corresponding function block (Fig. 7).

The procedural control entities have corresponding state transition diagrams describing their dynamics. Therefore the body of each function block, representing an entity in the code, contains an algorithm implementing the state transition diagram. The algorithm is yet another piece of FBD code, where states are represented with function blocks. A snippet of code representing the state transition diagram in Fig. 3 is shown in Fig. 8.

The state transition mechanism together with the state actions are implemented within the bodies of function blocks representing the states. Furthermore, the function blocks representing substates have to be called within the body of the corresponding superstate thus maintaining the hierarchy of the model.

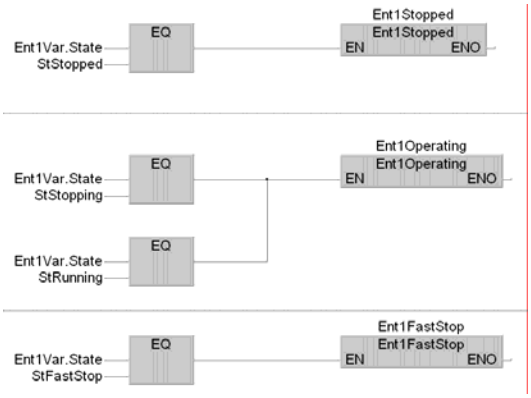

Fig. 8. Mapping of state transition diagrams.

4. A SOFTWARE MODELLING TOOL

A software modelling tool has been developed in order to support the software development process. The main functions of the modelling tool are to provide a graphical editor for ProcGraph models, to automatically generate function block diagram code and documentation from the model. Fig. 9 shows a schematic representation of the modelling tool.

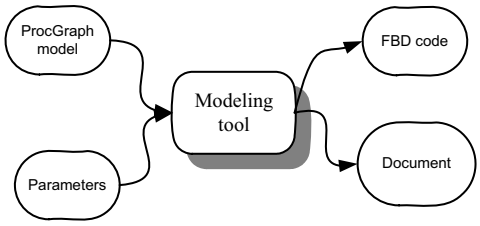

Fig. 9. Inputs and outputs of the modelling tool.

One of the most important functionalities of the tool is automatic generation of function block diagram code from ProcGraph models. In generating program code in a graphical programming language such as FBD, the generator has to create not only the correct content but also the right form (graphical image) of the code, which is more difficult than generation of code in textual programming languages such as C or Java. 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 code generation process is illustrated in Fig. 10. Firstly, the user creates and edits the ProcGraph model with the help of a graphical editor. When the model is completed, code generation can be activated. The resulting product of the code generation process is FBD source code, which has to be compiled to executable code using an appropriate compiler. Finally, the executable code can be uploaded to the programmable logic controller.

Compared to manual mapping of models to code, automatic code generation is considerably faster. Furthermore, manual coding introduces a number of errors which do not occur in automatic mapping.

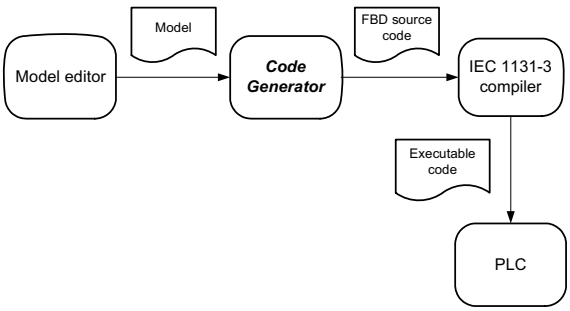

Fig. 10. Code generation process.

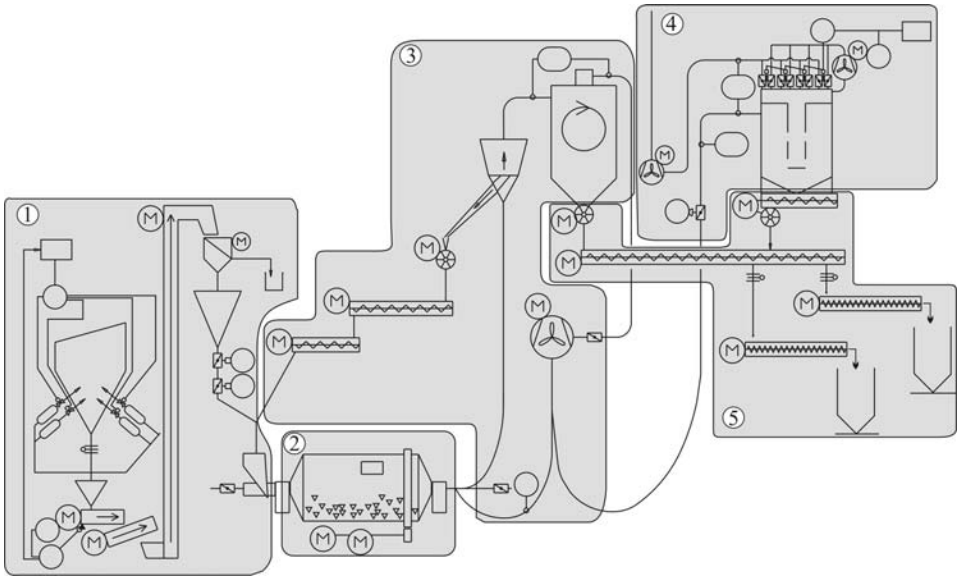

Fig. 11. Ore grinding process.

5. EXAMPLE OF APLICATION OF THE MODELLING TOOL

To demonstrate the use of the modelling tool described in the previous section, a control system of an ore grinding process shown in Fig. 11 is employed. The process 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 onto a belt scale. From the belt scale, the ore is transported by a conveyor belt to the elevator and from there to a vibration sieve. The sieved ore falls through a funnel and a damper system into the grinding mill. In the rotating mill, the ore is ground. 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.

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.

Fig. 12 shows a screenshot of the main window of the modelling tool with the procedural control entities editor containing the five procedural control entities and relationships between them. Each procedural control entity has a corresponding state transition diagram, describing the entity's dynamics. If the user clicks on the corresponding procedural control entity in the PCED editor, the state transition diagram editor window opens (Fig. 13). Once the complete model is built, automatic code generation can be started. Fig. 13 shows a state transition diagram in the modelling tool and the resulting FBD code that implements it. The resulting code was created automatically by the tool. Fig. 13 shows 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.

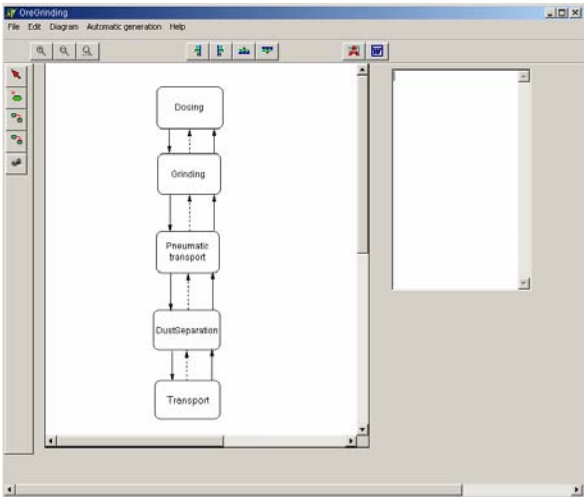

Fig. 12. Screenshot of the PCED editor.

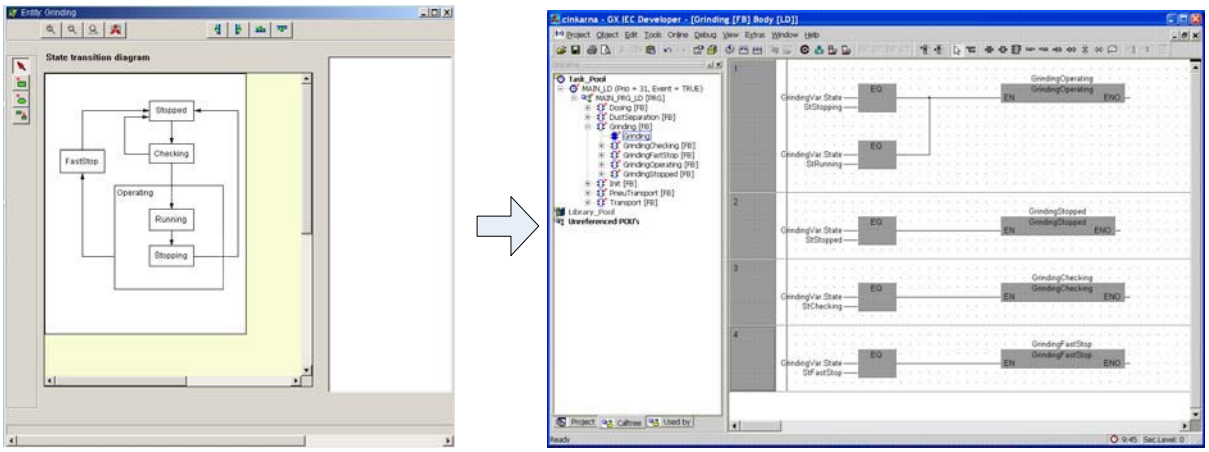

Fig. 13. Screenshot of the STD editor and generated FBD code.

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 manpower resources. In this article it is shown 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.

Automatic code generation significantly reduces the duration of software development process and at the same time improves the quality of the product-procedural control software.

Modern control applications very often consist of distributed control systems. For that reason, our future work will be focused on modelling and code generation for such systems, following the guidelines of the IEC 61499 standard.

ACKNOWLEDGMENT

The financial support by the Ministry of Education, Science and Sport of the Republic of Slovenia are gratefully acknowledged.

REFERENCES

Booch G., J. Rumbaugh and I. Jacobson (1999). *The Unified Modeling Language User Guide*, Addison Wesley, Boston.

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

DeMarco, T. (1978). *Structured Analysis and System Specification.* Prentice Hall, Englewood Cliffs.

Edan, Y. and N. Pliskin (2001). Transfer of Software engineering Tools from Information Systems to Production Systems. *Computers & Industrial Engineering*, **39**(1), 19-34.

Frey, G. and L. Litz (2000). Formal methods in PLC programming, *Proc. IEEE Conference on Systems Man and Cybernetics SMC 2000*, Nashville.

Godena, G. (2004). ProcGraph: a procedure-oriented graphical notation for process-control software specification. *Control Engineering Practice*, **12**(1), 99-111.

Harel, D. (1987). Statecharts: A Visual Formalism for Complex Charts. *Science of Computer Programming*, **8**(3), 231-274.

Kandare, G. (2004). *Computer aided design of procedural control software for programmable logic controllers.* PhD Thesis, University of Ljubljana.

Krueger, C. W. (1992). Software reuse. *ACM Computing Surveys*, **24**(2), 131-184.

Lewis, R.W. (1998). *Programming industrial control systems using IEC 1131-3*. The Institution of Electrical Engineers, London.

Ludewig, J. (2003). Models in Software Engineering – an Introduction. *Software and Systems Modeling*, **2**(1), 5-14.

Wieringa, R. (1998). A Survey of Structured and Object-Oriented Software Specification Methods and Techniques, *ACM Computing Surveys*, **30**(4), 459-527.