AUTOMATIC PROGRAMME SYNTHESIS

Gregor Kandare

Department of Systems and control
Jožef Stefan Institute
Jamova 39, 1000 Ljubljana, Slovenia

Abstract: This article is dealing with the challenge of automating the software development process from user requirements to programme code. The first part of the process – building a model from requests tends to be quite a difficult for the computer to handle it. The reason for this is that the requests are most often given in a natural language, which is full of ambiguities. Considering fact that the models are mostly formal, automating the second part – translation of models into code results to be feasible with computers.

Keywords: Code generation, process control, sequential control.

1 Introduction

Modern control systems cover an ample set of functions and enclose a broad set of hardware devices, ranging from simple sensors and actuators, controllers to complex computer systems. Consequently, control software is becoming ever more complex and difficult to develop and maintain. In the process control domain, there exists a broad spectrum of software like e.g. software for digital PID controllers, fuzzy logic, batch process control software, scheduling, supervision and fault detection software, etc.. The issues regarding control software are manifold ([8]). In the first place, control software has to be reliable ([10]). Control software is amongst others used in extremely critical applications such as control systems of nuclear power plants and airplanes. Failures in such systems can cause ecological disasters and loss of lives. Another issue is complexity of control software systems. In order to cope with complexity, a systematic approach to the software development has to be undertaken.
As mentioned above, there exist various very diverse types of control software. However, in this article we limit ourselves to a small subset of them - software for procedural control of continuous processes controlled by programmable logic controllers. Beside batch processes, continuous processes are most common in chemical industry. Design of batch process control systems and their organisation is well covered by the ISA S88 standard. There also exist various software tools for batch process control. In the domain of continuous processes this is not the case, which leaves many things to be done in this field.
Several solutions for tackling the problems of programmable logic controller software development have been proposed in the literature ([1],[3],[6],[9],[10]). The solutions

are mainly based on lifecycle view of software. According to this view, software is considered as any other product, which 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 terminates with operation and maintenance phases. The intermediate phases are development phases of the product, which can be further divided into modelling and realisation phases. In the modelling phases, software engineering methods and tools for analysis and design are used. Modelling languages such as UML represent the methods and computer automated software engineering tools (CASE) are used as tools.

An important issue in procedural control software design process is also 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) play in development of other products. The main objective of CASE tools is to automate the development phases of software lifecycle and transitions between the phases. 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, one of the most important capabilities of CASE tools is that they can automatically generate programme code from the models. The most important advantages of automatic code generation are that the mapping from model to final product requires no human effort and as good as no time. Another benefit is that due to automation of the mapping process the probability of mapping errors decreases radically. In this article, a process of automatic code generation for PLCs is presented.

2 The ProcGraph modelling language

ProcGraph is a modelling language specialized for design of procedural process control software. A more detailed description of the language can be found in ([1]).

ProcGraph models consist of three types of diagrams, each describing a particular view of the system to be built:

- *Entity diagrams (<ED>)* depict conceptual decomposition of the system as well as relationships between conceptual components.
- *State transition diagrams (<STD>)* describe the dynamical view (behaviour) of the conceptual components.
- *Entity dependency diagram (<EDD>)* portrays causal and conditional dependencies between the conceptual components.

3 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 for transition from one phase to another is minimised.

ProcGraph modelling 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 modelling phase if followed by the programming (coding) phase. As regards to the freedom of choice of models and abstractions, in this phase we are not as free as we are in the phase of modelling. While in the modelling phase we can design and adapt the modelling language so that it suits our needs, in the programming phase this is not feasible. The reason for this is that the "model", which results from the programming phase, is actually programme source code in one of the programming languages. Programming languages have fixed syntax and semantics. Furthermore, in the choice of the target programming language, we are limited by the hardware platform.

In the case of programmable logic controllers, available programming languages are in the majority of cases the ones defined by the IEC 61131-3 standard (**Error! Reference source not found.**). 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 diagrams 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.

3.1 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 written as follows:

$$\langle ProcGraph\rangle = \langle ED\rangle + \langle STD\rangle + \langle EDD\rangle \tag{3.1}$$

Using expressions **Error! Reference source not found.** and (3.1), we can define the mapping function of ProcGraph models into Function block diagram language:

$$C_G: \langle ProcGraph\rangle \rightarrow \langle FBD\rangle \tag{3.2}$$

The expression (3.2) is general and describes the transformation of the entire model into function block diagram source code.

4 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 this popularity is the fact that software development companies have realised that manual programming is a very time-consuming and error-prone task. Moreover, if the software model built by the analyst/designer has sufficiently low abstraction level, the programming itself is a relatively mechanical task which can easily be taken over by the computer.
In automating the programming process, costs of the programming manpower can be saved, development times can be significantly reduced and also the amount of errors is decreased.
Automatic code generators resemble the 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 the 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.

4.1 Automatic code generation from ProcGraph models

The process of conversion of software model to the end product of software development – executable code, is shown in Figure 1.

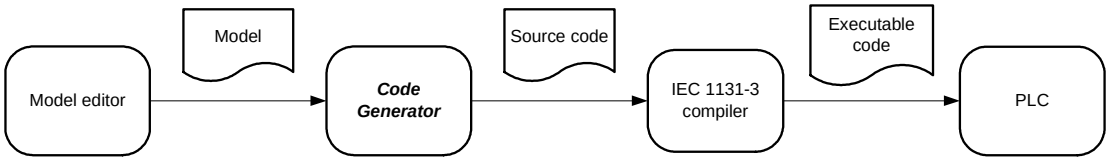

Figure 1: Conversion of software models into executable code

As noted in the preceeding section, we see that the input into the code generator is the model (in our case ProcGraph model). On the output of the code generator, FBD programme code is generated. A rough description of the code generation procedure is depicted in Figure 2. First, definitions of data structures are created, subsequently function blocks representing procedural control entities are generated. Furthermore, a hierarchy of function blocks representing states is constructed and finally, a definition of global variables is performed.

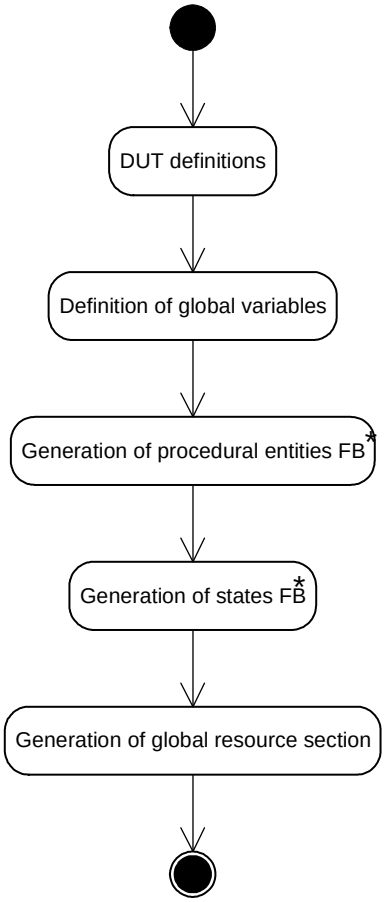

Figure 2: The automatic code generation algorithm

Each step of the algorithm from the Figure 2 is further decomposed into more atomic procedures which finally in atomic actions.
The code generator creates a text file that describes both the graphical as well as textual segment of the source code in function block diagram. Creating the graphical part of the code is a more demanding task, because issues such as placement of elements in the scheme and routing of connections have to be taken into account.

5 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 low abstraction level of the PLC programming languages. The programming process is time-consuming as well as extremely error-prone and consequently consumes a lot of manpower resources. In section 3 of 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 standardization and reusability of the generated code. In the code generation process, the appropriate patterns are used and filled with the corresponding content. The most important advantages of automatic code generation are evident – significant reduction

of software development time and consequential cost reduction, improvement of quality and therewith reliability of software.

References

[1] G. Godena, "ProcGraph: a procedure-oriented graphical notation for process-control software specification", *Control Engineering Practice*, vol. 12, pp. 99-111.

[2] G. Kandare, G. Godena and S. Strmčnik, "A new approach to PLC software design"

[3] Bonfatti, F., G. Gadda and P.D. Monari: Re-usable Software Design for Programmable Logic Controllers, *Proceedings of the workshop on Languages, Compilers & Tools for Real-Time Systems (LCT-RTS 1995),* str. 31-40, La Jolla, California, Junij 1995.

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

[5] Chirn, J.-L. and D.C. McFarlane: Petri Nets Based Design of Ladder Logic Diagram, *Proceedings of the UKACC International Conference on CONTROL 2000*, University of Cambridge, UK, 2000.

[6] Davidson, C.M., J. McWhinnie and M. Mannion: Introducing Object Oriented Methods to PLC Software Design, *Proceedings of the International Conference and Workshop: Engineering of Computer-Based Systems (ECBS '98),* str. 150-157, Jerusalem, Izrael, Marec-April 1998.

[7] Dierks, H. und J. Tapken: MOBY/PLC: Eine graphische Entwicklungsumgebung für SPS-Programme, *Automatisierungstechnik,* Vol. 49, No. 1, str. 38-44, Januar 2001.

[8] Edan, Y. and N. Pliskin: Transfer of Software engineering Tools from Information Systems to Production Systems, *Computers & Industrial Engineering*, Vol. 39, No.1, str. 19-34, Februar 2001.

[9] Fischer, K. und B. Vogel-Heuser: UML in der Automatisierungstechnischen Anwendung – Stärken und Schwächen, *Automatisierungstechnische Praxis*, Vol. 44, No. 10, str. 63-69, Oktober 2002.

[10] Frey, G. and L. Litz: Formal methods in PLC programming, *Proceedings of the IEEE Conference on Systems Man and Cybernetics SMC 2000,* str. 2431-2436, Nashville, Oktober 2000.