Available online at www.sciencedirect.com
SCIENCE DIRECT®
PERGAMON
Control Engineering Practice 12 (2004) 99–111
CONTROL ENGINEERING PRACTICE
www.elsevier.com/locate/conengprac

# ProcGraph: a procedure-oriented graphical notation for process-control software specification

Giovanni Godena*

*Department of Computer Automation and Control, J. Stefan Institute, Jamova 39, 1000 Ljubljana, Slovenia*

Received 18 January 2002; accepted 5 January 2003

**Abstract**

This paper presents a procedure-centric approach to process-control software specification. A high-level, domain-specific specification notation called ProcGraph was developed, based on three types of diagrams that describe the modelled system using a domain-oriented hierarchical structure of interdependent procedural control entities and state-transition diagrams. The state-transition diagrams describe the behaviour of the procedural control entities. The processing content of the states is defined by sequential programs. In order to illustrate the presented models and notation, some segments of a real process-control application that deals with the drying process in the production of titanium dioxide are presented.


*Keywords:* Requirements analysis; Software specification; Domain analysis; Conceptual representations; Programmable logic controllers; Process automation; Systems methodology; Finite state machines; Industrial control; Control technology

## 1. Introduction

As a result of the increasing complexity of computer-based process control and the growing demand for high-quality products and processes, there is an ever-increasing need for a systematic approach to the software development process. Indeed, this systematic approach should be extended to the entire software life-cycle, and particularly to the activities of requirements analysis and specification.

Over the past two decades we have seen two main approaches to software development. In the 1980s, structured methods, which were based on the so-called functional decomposition, were very popular. These methods brought many benefits; however, their main drawback was that they separated the data from the related operations. With the appearance of object-oriented methods the structured approach rapidly lost favour, to the extent that even the word ''structural'' began to acquire a pejorative meaning. Clearly though, there is obviously nothing wrong with the structure, which was and will remain the main complexity-managing mechanism of every method.

Object methods brought with them encapsulation and information hiding (object-based), which substantially improved data integrity. They also brought the concept of classification (object-oriented) along with all the accompanying concepts that improve reusability. Object methods undoubtedly represent an improvement over the previous approach. During the last decade we have encountered various object-oriented approaches, and in the last few years they all have converged to form the object-oriented modelling language named Unified Modelling Language (UML).

However, there are also some problems with applying object-oriented methods to the domain of process-control software. The first problem is related to the decomposition method, i.e. with determining the relevant entities for modelling. Failure to determine the right entities is probably the main source of problems in the development of process-control software.

The primary purpose of the optimum decomposition should be to expose the goal, i.e. the responsibility (Beck & Cunningham, 1989) of the parts of the control system under development, which results in better abstraction than exposing the means for achieving that goal. Thus the optimum decomposition is to the relevant entities

*Department of Systems and Control, J. Stefan Institute, Jamova 39, 1000 Ljubljana, Slovenia. Tel.: +386-1-477-3619; fax: +386-1-425-7009.

*E-mail address:* giovanni.godena@ijs.si (G. Godena).

0967-0661/03/$ - see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/S0967-0661(03)00002-9

from the problem domain, which are not valves and pumps, but procedural control entities, e.g. operations or activities. In spite of this, most of the methods found in the literature do not primarily expose the goals, rather, they expose the means for achieving those goals. This is also (if not primarily) true for the object-oriented approaches, which are nowadays widely accepted. Several authors consider that object orientation itself leads to optimum system modelling (Davidson & Mc Whinnie, 1996; Rossetti et al., 1995; Storr, 1997; Wheeler, 1995), but this is not completely true. These approaches most frequently define relevant entities among tangible entities, i.e. physical devices. This gives object models of functional system decomposition—those devices perform their intrinsic functions. The main problem of modelling the control system with equipment entities lies in the resulting high coupling between the software modules controlling single equipment entities. The only reason why this may be hard to realise from the literature is the fact that the examples given, as a rule, are too simple to be of practical relevance.

Of course, it is not the intention of the author to negate the advantages of using object technology, but it is—as its name implies—just a technology, and using it to represent a sub-optimum model of decomposition can only lead to a sub-optimum solution. It seems that in this respect most of the process-control-software community is squarely behind the business-software community, which replaced the function-centric view of their domain with the process(procedure)-centric view (Jacobson, Ericsson, & Jacobson, 1995). Yet there is a section of the process-control-software community that regularly uses the decomposition to procedural control entities as the main design abstraction. It is the batch-control-software community, which bases its work on the batch-process control standard (ANSI/ISA, 1995). The abstraction based on the decomposition to procedural control entities is also used by process (chemical) engineers, who describe their processes in terms of procedures and not in terms of equipment entities.

An interesting example is in Abou-Haidar, Fernandez, and Horton (1994), where the authors first state that the optimum decomposition is to equipment entities, but later in the paper they give an example of an equipment entity statechart, which is in fact the statechart of an operation (procedural control entity), though not complete. It seems as if the authors chose a better decomposition without being aware of the fact. The same is also true for the approach, presented in Godena (1997), where the system was already decomposed to procedural control entities, but the author referred to them as "control functions".

The second problem with applying object-oriented methods to the domain of process-control software is related to the semantic distance between the OO abstraction and the languages currently used in process control, which are defined by the IEC 61131-3 standard (IEC, 1993). Due to the existence of this large semantic distance, it would be very difficult to achieve seamless mapping between specification and implementation. Consider, for example, UML (Booch, Jacobson, & Rumbaugh, 1999). It is a rich and complex notation (consisting of nine types of diagrams), which is general enough to be applicable to most domains. But this strength of UML is also its main weakness when considering its use in PLC-based process control. Taken as a whole, the author finds it too general and too complex for such a narrow domain. And, furthermore, taking only a small subset of UML would reduce the available expressive power too much. What we need is a domain-specific, graphical notation, consisting of only a few types of diagrams, which is strongly related to the process-control domain, particularly to its procedural control entities (PCEs). These PCEs should be the main elements of the notation. The decomposition of PCEs to other PCEs at a lower hierarchical level should also be provided for. The behaviour of the PCEs at the lowest level should be described by some kind of extended finite state machines (FSMs). An important aspect of the notation is the concept of the mutual dependencies of PCEs (including synchronisation). The notation should give support for minimising the coupling between PCEs, which is the most important attribute of good modularization. Other notations, e.g. UML, Schlaer-Mellor (Schlaer & Mellor, 1992), define messages between objects as the means of achieving synchronisation, but this is too general and too unrestricted. In the author's opinion the notation should limit as much as possible the number of allowable types of coupling i.e. the types of dependence relations between PCEs. These dependence relations should also be part of the graphical notation and, as a consequence, appear explicitly at a very high level in the model, which is surely the best method of reducing the extent of such dependencies and consequently minimising coupling.

The aim of this paper is to present a high-level, domain-specific specification notation called ProcGraph. The paper is structured as follows. First, a description of the domain of continuous process control is given. The next section presents the domain-oriented, high-level specification notation called ProcGraph. Then, an illustration of an engineering application based on the presented notation is given. The automation of the titanium dioxide suspension drying process as a part of the titanium-dioxide production process was chosen as the example. Finally, the presented notation is compared with some similar notations, the benefits of introducing the proposed notation are discussed, and the plans for future work are presented.

## 2. The domain of continuous process control

The domain of process control can be divided into the following three subdomains: control of the continuous, batch and discrete processes. Each of these subdomains can again be divided into more subdomains, e.g. the subdomain of basic control (achieving and maintaining a desired state of the process equipment or of the process) and the subdomain of procedural control (performing process-oriented activities).

In this paper the discussion is limited to the subdomain of continuous process control. The reason being that this subdomain is simple enough to be handled in detail, while it is also complex enough to give a real picture of the presented approach. The resulting models are very similar to those that would be obtained for the domains of sequential and batch process control.

### 2.1. Domain models for continuous process control

From the viewpoint of the software, the continuous processes are less complex than, for example, the batch processes. The main potential sources of complexity for the continuous-process-control software are the changes of their state, e.g. starting, stopping and, as a major source of complexity, the reactions to exceptional situations. These potential sources of complexity should therefore be considered with particular care while building the conceptual model of continuous-process-control software.

Fig. 1 shows three models describing various aspects of continuous process control: the model of procedural control, the physical model and the process model. The relation between the entities of the three models is the following: the entities from the procedural control model, *combined with the* entities from the physical model, *provide process functionality to carry out* the entities from the process model. These models were adopted from ANSI/ISA (1995), adapted for continuous processes, and mapped to the specification notation.

### 2.2. Entity model

The physical model represents the decomposition of the process equipment to equipment entities—equipment groups and equipment elements. Each equipment element is described by (or is classified to) a corresponding equipment-element type. The physical model *contains* basic control. The basic control is aimed at achieving and maintaining a determined state of the process equipment or the process. Here we are dealing with PID (Proportional-Integral-Derivative) control loops, sequential control and process monitoring. This model, therefore, belongs to the subdomain of process-equipment engineering and represents the world of discourse between the system analyst and the control/instrumentation engineer.

The procedural control model is composed of the *procedure*, which is decomposed into parallel (concurrent) *operations* (to model an implicit concurrency in the process). The operation is a procedural control element,

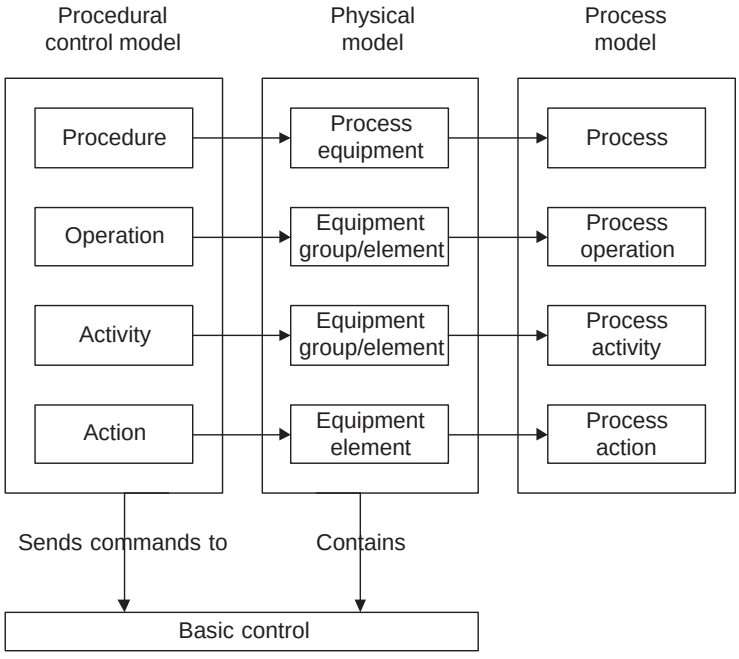


Fig. 1. Three models of continuous process control.

which defines the execution of some process-oriented activity, resulting in physical or chemical changes to materials or their energy state. This model, therefore, belongs to the subdomain of process engineering and represents the world of discourse between the system analyst and the process/chemical engineer. Each operation is further decomposed into *parallel activities* (to model an implicit concurrency in the operation) and each activity can also be decomposed into parallel subactivities. At the lowest decomposition level each activity is composed of a number of *actions* that perform *process actions* by sending commands to and accepting signalisations from the *basic control*. Technological requirements and constraints determine the decomposition of an operation to the activities. During the decomposition, special regard should be paid to the criteria of good modularization (high internal cohesion of the entities and low coupling between them). Good modularization enables us to manage successfully the complexity of the development, the use and the maintenance of the control system. Fig. 3 shows the relations between the entities of the physical model and the procedural control model (the definitions of the symbols appearing on the entity relationship (ER) diagrams are given in Fig. 2).

## 3. ProcGraph specification notation for continuous processes

The ProcGraph model only contains information about the procedural control entities (right-hand part of Fig. 3) and not about the basic control in equipment entities. The reason for this decision is that in the procedural control there is much more complexity and diversity between the different applications than in basic control, where we usually can simply reuse equipment-element type blocks, i.e. blocks performing basic

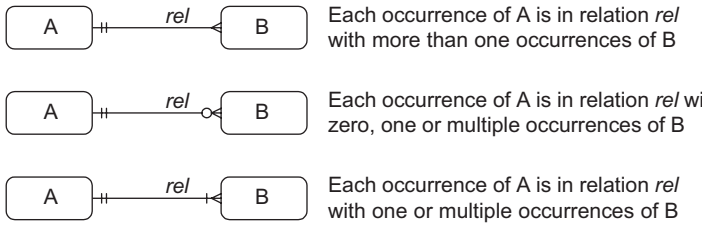


Fig. 2. Symbol definitions for ER diagrams.

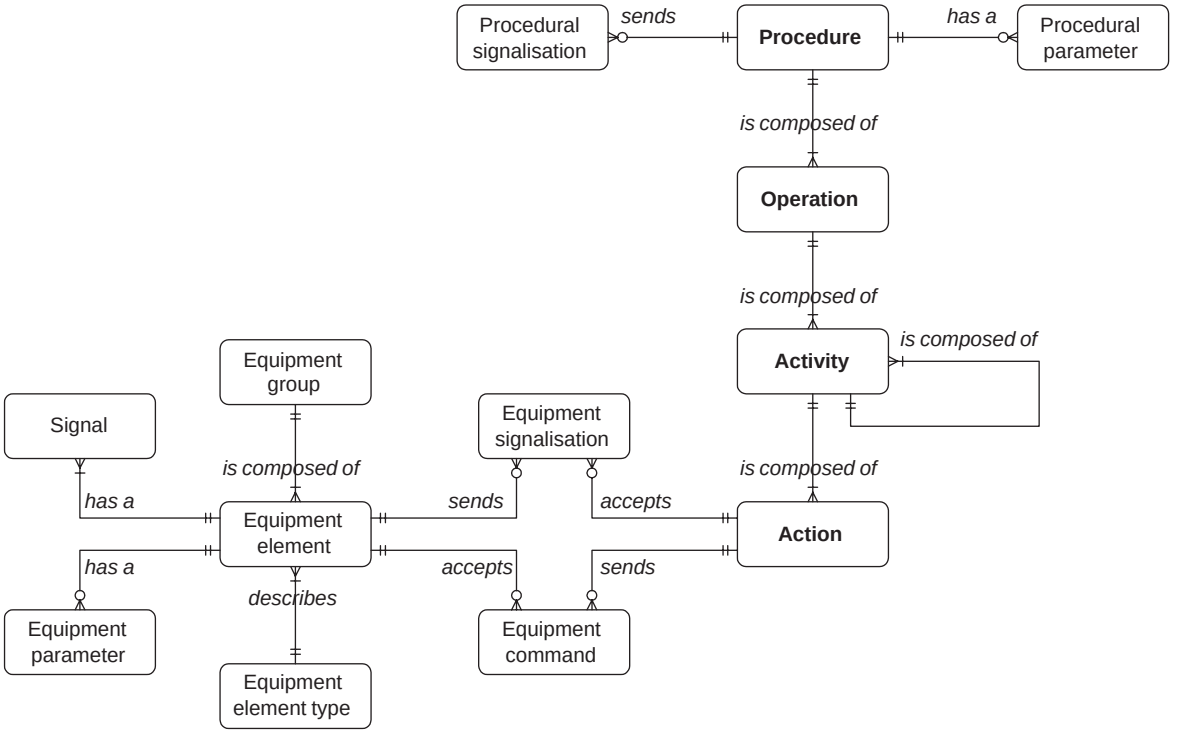


Fig. 3. Relations between entities of the physical model and the procedural control model.

control. Libraries of these basic control blocks are already becoming a part of modern development environments (e.g. SIEMENS, 1999).

The ProcGraph notation consists of four different notation elements—three diagram types and a symbolic pseudo-language. The three diagram types are aimed at presenting three types of information relating to procedural control entities (PCEs): the hierarchy of PCEs, the behaviour of these PCEs and the behaviour dependencies between PCEs. In order to minimise the number of dependency types, which was one of the main goals, only two types of dependencies were defined—the condition of transition and the propagation of transition.

### 3.1. Diagram notations

The three types of diagrams are the Procedural Control Entities Dependencies Diagram (PCEDD), the Procedural Control Entities State Transition Diagram (PCESTD) and the Procedural Control Entities Dependencies State Transition Diagram (PCEDSTD). The PCEDD shows the dependencies between PCEs at the highest level. With a PCEDD, an example of which is shown in Fig. 4, we can see PCEs and the existence, the type and the direction of their dependence relations. On the other hand, what we cannot see at this level, are the concrete dependent states and transitions. In PCEDD, as well as in PCEDSTD, the conditional dependence is denoted by a solid line with an arrow oriented towards the dependent PCE. The same is true for the propagation dependence, the only difference being that in this case we use a dashed line. In Fig. 4, the upper relation means that some transition(s) of the PCE **A** are dependent on some state(s) of the PCE **B** (i.e. they are only allowed if the PCE **B** is in a determined state). The lower relation in Fig. 4 means that some transition(s) of the PCE **C** occur as a propagation of some state(s) of the PCE **D**.

The PCEDD has two decompositions to the lower-level diagrams. The explosion of a PCE is a PCESTD, i.e. the state-transition diagram of that PCE, which does not show the state dependencies with other PCEs. The main features of PCESTDs are:

1. *States* with their Entry, Loop and Exit *sequences of actions*.
2. State *transitions* with their *causes* (e.g. completion, operator command, auto-transitions). Two types of transitions were defined: transition on completion (denoted by a line ending with an outlined arrow) and transition on event (denoted by a line ending with a filled arrow). Only states are allowed to have actions—transitions may not have actions; the underlying state-machine model is in fact an extension of a Moore-type state machine (Hopcroft & Ullman, 1979).
3. *Nested states* (superstates, which are composed of substates, at the lowest level there are elementary states). The purpose of nested states may be to incorporate conceptually related entities or, as the most important purpose, to avoid the repetition of information by closing into a superstate the actions and/or transitions and/or interactivity dependence relations common to a number of states. Note that a superstate is in fact concurrent with its active substates at all nesting levels (there may be as many concurrently active states as the number of nesting levels). The ProcGraph notation does not include the notion of the initial substate. In fact, all TO transitions are drawn explicitly to elementary states, while superstates may only have FROM transitions.

An example of a part of a PCESTD is shown in Fig. 5. The PCESTD part consists of a superstate $S_A$, which consists of two substates: $S_{a1}$ and $S_{a2}$. *Cause*($T_A$ : $S_{a1} \rightarrow S_{a2}$) is the cause of the (on event) transition $S_{a1} \rightarrow S_{a2}$. The way of noting the actions performed

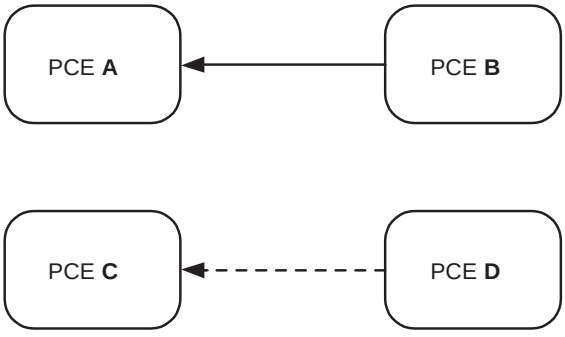


Fig. 4. PCEDD.

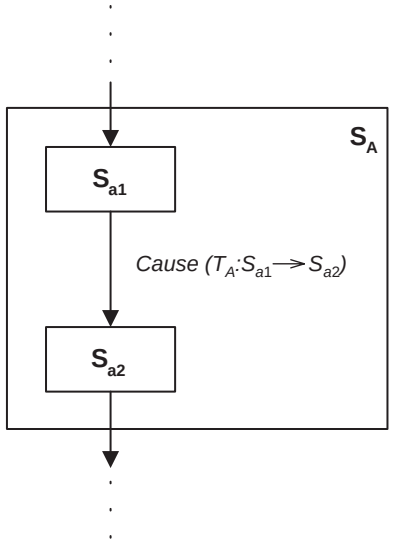


Fig. 5. PCESTD.

during the states (superstates and elementary states) and the logic defining the causes of transitions is described in the next section.

*Cause* contains only the PCE's internal causes of transition, while the conditional-causal relations with other PCEs, which were introduced in the PCEDD, are not shown in greater detail in the PCESTD. These conditional-causal relations are defined in detail in the PCEDSTD, which is the explosion of the connections (dependence relations) between two PCEs in the PCEDD. It shows which transition of one PCE is dependent on which state of another PCE. Consequently, in order to model the behaviour of a PCE entirely, the information is needed from its PCESTD and from as many PCEDSTDs as there are PCEs in conditional-causal relations with that PCE. Before considering the PCEDSTD, let us give a brief definition of the dependence relations between PCEs. These are the following relations, where *Cond* is a condition, $T$ is a transition and $S$ is a state:

1. *Condition of a state transition*: the state transition of the PCE **A** from state $S_{a1}$ into state $S_{a2}$ is only allowed if the PCE **B** is in some determined state $S_{b1}$. The relation is shown by

$$Cond(T_A : S_{a1} \rightarrow S_{a2}) = (S_B = S_{b1}). \quad (1)$$

2. *Propagation of a state* of some PCE **A** is a transition of the operation **A** from state $S_{a1}$ into state $S_{a2}$, caused by the fact that some other PCE **B** is in such a state $S_{b1}$ that does not allow the PCE **A** to remain in the state $S_{a1}$. The relation is shown by

$$S_B = S_{b1} \Rightarrow T_A^{propag} : S_{a1} \rightarrow S_{a2}. \quad (2)$$

As a consequence of relation (2), the following relations often occur:

- the state $\overline{(S_{b1})}$ of the PCE **B** is a condition of the transition of the PCE **A** into state $S_{a1}$; the relation is shown by

$$Cond(T_A : S_{ax} \rightarrow S_{a1}) = (S_B \neq S_{b1}), \quad (3)$$

- the state $\overline{(S_{a1})}$ of the PCE **A** is a condition of the normal (non-automatic) transition of the PCE **B** into state $S_{b1}$; the relation is shown by

$$Cond(T_B : S_{bx} \rightarrow S_{b1}) = (S_A \neq S_{a1}), \quad (4)$$

- from (4) and (2) it also follows that the propagation of the PCE **A** from state $S_{a1}$ into state $S_{a2}$ cannot really be caused by any sort of transition of the PCE **B** into state $S_{b1}$, but only by auto-transition or propagated transition; the relation is shown by

$$T_B^{auto\,OR\,propag} : S_{bx} \rightarrow S_{b1} \Rightarrow T_A^{propag} : S_{a1} \rightarrow S_{a2}. \quad (5)$$

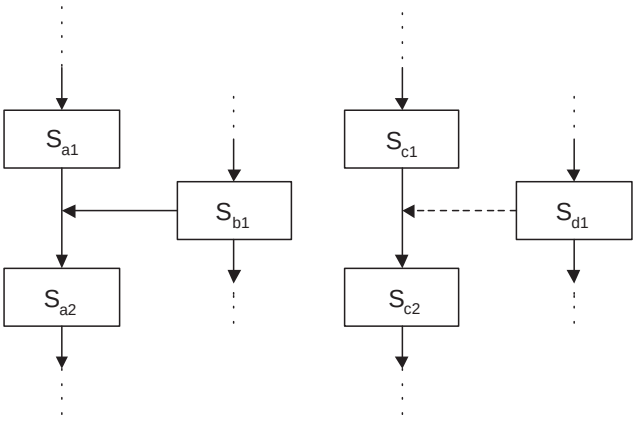


Fig. 6. PCEDSTD.

State propagation (as well as state auto-transition) can be immediate or with a delay. The delayed propagation is denoted by the symbol Δ.

The most frequent case of state propagation of a PCE is the propagation of (auto)stopping of one PCE to the stopping of another PCE. Such stopping is referred to as *domino stopping*. PCEs in such a relation form the so-called *domino-chains* of PCEs, which determine, as a consequence of (3) and (4), the sequence of starting and of the normal (non-automatic) stopping of PCEs inside them.

An example of a part of a PCEDSTD is shown in Fig. 6. The left-hand part of the figure shows that state $S_{b1}$ is a condition for the transition $S_{a1} \rightarrow S_{a2}$, whereas the right-hand side shows that state $S_{d1}$ causes (is propagated into) the transition $S_{c1} \rightarrow S_{c2}$.

### 3.2. Defining the processing of the states and the transitions

In order to describe elementary states, the causes of their out-transitions and their sequences of actions should be specified. As mentioned above, only states are allowed to have actions—transitions may not have actions. The actions of the states are grouped into Entry, Loop and Exit actions. All actions can be instantaneous (e.g. issuing a command to an equipment entity) or durable (e.g. waiting for an equipment entity to perform a command). The difference between Entry, Loop and Exit sequences is that the Entry sequence is executed only once upon activation of the state, the Exit sequence is executed only once before deactivation of the state, and the Loop sequence is being executed cyclically all the time while the activity is in a certain state.

With the nesting states the order of execution is as follows: upon entry into a state that is nested within one or more superstates first execute all Entry actions, beginning with the outermost state and ending with the

innermost state, then cyclically all Loop actions, always beginning with the outermost state and ending with the innermost state, and before exit all Exit actions, beginning with the innermost state and ending with the outermost state.

The sequence of actions and the causes of transitions can be defined by using one of the higher-level languages defined by the IEC 61131-3 standard (e.g. Structured Text or Sequential Function Chart). This option becomes particularly interesting when considering the possibility of upgrading the IEC 61131-3-based development environments with the ProcGraph notation, which would enable the automatic generation of code from the specifications whose higher-level part is given in ProcGraph notation.

For describing the sequences of actions and the causes of transitions semi-formally, a symbolic pseudo-language was introduced; the symbols of which are given in Table 1.

## 4. An example of a continuous-process-control application

### 4.1. Introduction

Titanium dioxide ($TiO_2$) is a white pigment that is widely used in paint and enamel production. The production of the $TiO_2$ pigment in the *Cinkarna* chemical works consists of a succession of several processes. Many of these processes were subjected to computer automation and optimisation in recent years (Černetič, Šubelj, & Selič-Podgoršek, 1994; Godena, 1997; Žele & Juričić, 1997; Šel et al., 1999). One of the processes in the last stage of the production is the continuous process of drying the $TiO_2$ suspension. A simplified technological scheme of the $TiO_2$ drying process is shown in Fig. 7. The $TiO_2$, in the form of a suspension with a concentration of about 650 g/l, enters the drying process in storage vessel A, having been washed in the previous stage. From the storage vessel A, the suspension is pumped into the rotating vacuum-dryer B, where it is dried to approximately 50% $TiO_2$. After vacuum drying the suspension is transported to the dispergating vessel C and from there it is pumped to the drying chamber (turbo-dryer) D, where the water is evaporated instantaneously by the effect of the hot flue gases from the thermo-aggregate E. The dried pigment is transported forward by pneumatic transport, powered by the fan F or G, to the silos H and I. Once inside, some of the pigment falls to the bottom, and some remains on the bag filters at the top of these silos, separated from the wet flue gases. The gases proceed through the scrubbing system (Venturi scrubber J and water-gas separator K) to the chimney L. From the silos H and I, the pigment is transported by the screw-conveyors transport system M to the next process—micronization.

### 4.2. ProcGraph requirements model

The procedural control was decomposed into four operations:

1. Evaporation and Pneumatic Transport—EPT. This operation performs the central part of the process—drying in the turbo dryer, pneumatic transport, separation on bag filters and flue-gases scrubbing. The operation is composed of three concurrent subactivities: the core activity EPT.Core and two auxiliary activities: Shaking, which controls the compressed-air shaking of the bag filters; and Dispergation, which manages the dispergator at the entry to the drying chamber (primarily level control).
2. Rotating vacuum drying (RVD).
3. Heat generation (HG).
4. Screw conveyer transport (SCT).

The operations (with subactivities) and their mutual dependencies are shown by the PCEDD in Fig. 8.

In Fig. 8 we can see only the existence and direction of the state dependencies, while the concrete dependent states and transitions are not shown at this level. So, for example, we can see that the SCT operation influences the EPT operation in two ways:

- a certain state of the SCT is a condition for a certain transition of the EPT.Core, and
- a certain state of the SCT causes (propagates to) a certain transition of the EPT.Core.

In the opposite direction, a certain state of the EPT.Core is a condition for a certain transition of the SCT.

As defined before, each PCE from the PCEDD explodes to its PCESTD, which models the PCE's behaviour. As an example, let us take the PCESTD of the EPT.Core activity, which is shown in Fig. 9. From the PCESTD in Fig. 9 we can see that, besides the "standard" states (Stopped, Starting, Running, Stopping), there is also the state Washing. The operation enters this state on an operator command or by auto-transition as a consequence of a disturbance in feeding the suspension into the drying chamber (failure of the pump or the high-level alarm in the dispergating vessel). The operation also enters the Washing state in the case of auto-stopping of the SCT operation—in this case with a delay (as can be seen from the PCEDSTD in Fig. 10). During the Washing state only the "drying" of the water takes place (there is only water passing through the drying chamber). The name of the Washing state follows from the fact (which can be seen from the PCESTD in Fig. 9), that a normal (non-automatic) transition from the Running state to the Stopping state

Table 1
Symbols of the pseudo-language

| Notation | Meaning |
|---|---|
| 1. Commands | |
| ↑ | Drive (motor) ON |
| ↓ | Drive (motor) OFF |
| ◎ | Open a discrete element (valve, damper) |
| ◉ | Close a discrete element (valve, damper) |
| ◐ | Set continuous element state (◐ := ×) |
| 2. Signalisations | |
| ↥ | Drive (motor) ON |
| ↧ | Drive (motor) OFF |
| ↻ | Rotation |
| Ɛ | Error |
| ○ | A discrete element (valve, damper) is OPEN |
| ● | A discrete element (valve, damper) is CLOSED |
| ◐ | Check continuous element state (◐ = ×) |
| 3. Modes | |
| L | Local |
| R | Remote |
| RA | Remote-Auto |
| RM | Remote-Manual |
| 4. Operators | |
| =, >, <, ≥, ≤ | Relational operators |
| ¬ | Negation (logical NOT operator) |
| & | Conjunction (logical AND operator) |
| v | Disjunction (logical OR operator) |
| ⇒ | Implication (truth of the left side implies the truth of the right side) |
| Δ (Time) | Delay in the duration of Time |
| ∇ (Cond) | Waiting for condition Cond |
| $\nabla_L$ (Cond,Time) | Waiting for condition Cond—waiting time limited to Time |
| T | State transition |
| SETTIMER (*name*, *time*) | The timer *name* is initialised to the duration *time* and started. |
| TIMEOUT (*name*) | The function returns the value *true*, if the timer *name* is out. |
| RESETTIMER (*name*) | The procedure resets the timer *name*. |
| DURATION (Cond,Time) | The function returns value TRUE, when the value of Cond is true in the continuous duration of at least Time |
| 6. Action sequence flow control | |
| IF $cond_1$ THEN $action_1$<br>ELSIF $cond_2$ THEN $action_2$<br>…<br>ELSIF $cond_m$ THEN $action_m$<br>ELSE $action_n$ ENDIF; | Executes one of m actions depending on the corresponding conditions, if no condition is true, $action_n$ is executed. |
| EXCEPTIONS<br>…<br>ENDEXCEPT | The statements inside the construct are executed concurrently with statements inside the same BLOCK construct as the EXCEPTIONS construct. The construct EXCEPTIONS contains primarily commands for checking the arising of exceptions; it can also contain alarm messages, state transition commands, and various *last wish* statements. The EXCEPTIONS construct has been described extensively in Godena and Colnarič (2000). |
| BLOCK *Name*<br>…<br>ENDBLOCK *Name* | The aim of the construct is to limit the scope of the command (primarily limiting the scope of the construct EXCEPTIONS). |

is only possible through the Washing state, during which the washing of the turbine at the entry to the drying chamber really occurs (otherwise, it would be necessary to change the turbine before re-starting the operation).

In Fig. 9 we can also see three superstates, Drying, Non-drying and Operating. The purpose of all three superstates is to represent all the features (actions and transitions) common to the corresponding substates.

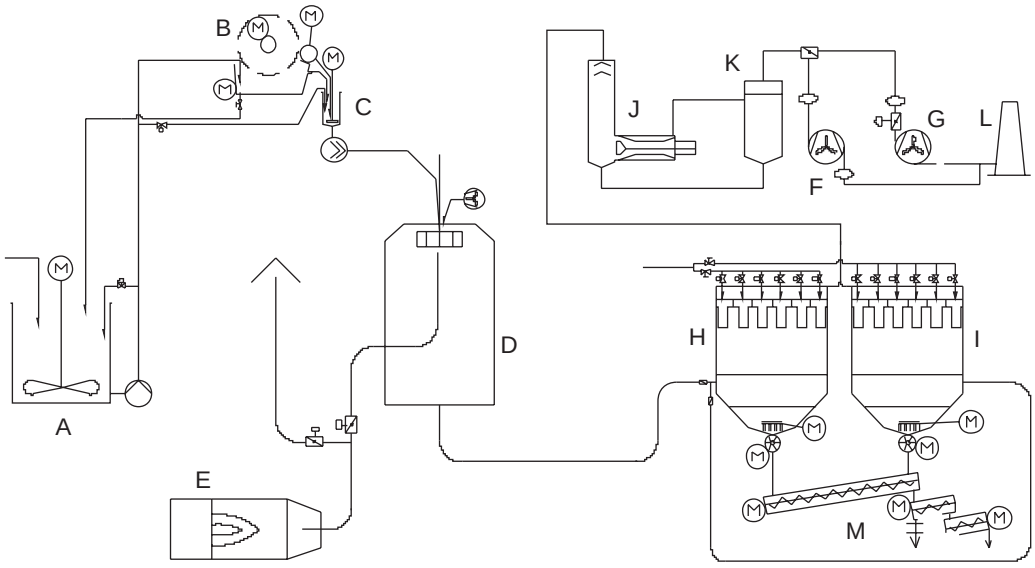


Fig. 7. $TiO_2$ suspension drying process.

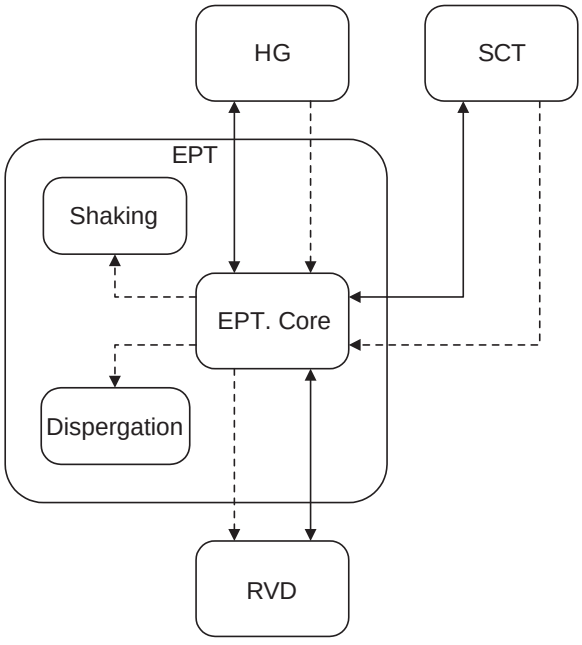


Fig. 8. Drying process PCEDD.

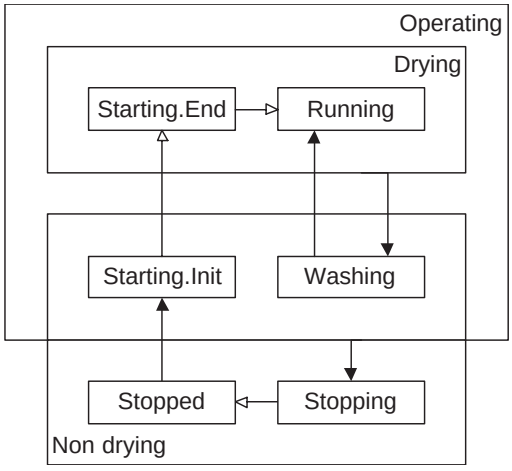


Fig. 9. EPT.Core activity PCESTD.

This is illustrated by the excerpts from the symbolic pseudo-code specifications of the superstates Drying and Operating, which are given in Listings 1 and 2, respectively. The superstates Drying and Non-drying also have an additional purpose of simplifying the notation of the interactivity dependence relations, as explained in the discussion of Fig. 10.

A detailed description of the conditional-causal relations between the operations EPT (its subactivity EPT.Core) and SCT is given by the PCEDSTD in Fig. 10. As mentioned before, the PCEDSTD represents an explosion of the block of inter-dependencies between two PCEs from the PCEDD. In Fig. 10 we can see the following dependencies:

- the state Running of the SCT operation is a condition for the transitions of the EPT. Core activity from the state Stopped to the state Starting, and from the state Washing to the state Running,
- the superstate Non-operating of SCT operation causes (is propagated to) the transition of the EPT.Core activity from the superstate Drying to the state Washing—here the symbol $\Delta$ indicates that it is a *delayed propagation* (the propagation only occurs after the SCT operation has been continuously, for a determined duration, in the superstate Non-operating),
- the superstate Non-drying of EPT.Core activity is a condition for the transition of the SCT operation from the state Running to the state Stopping.

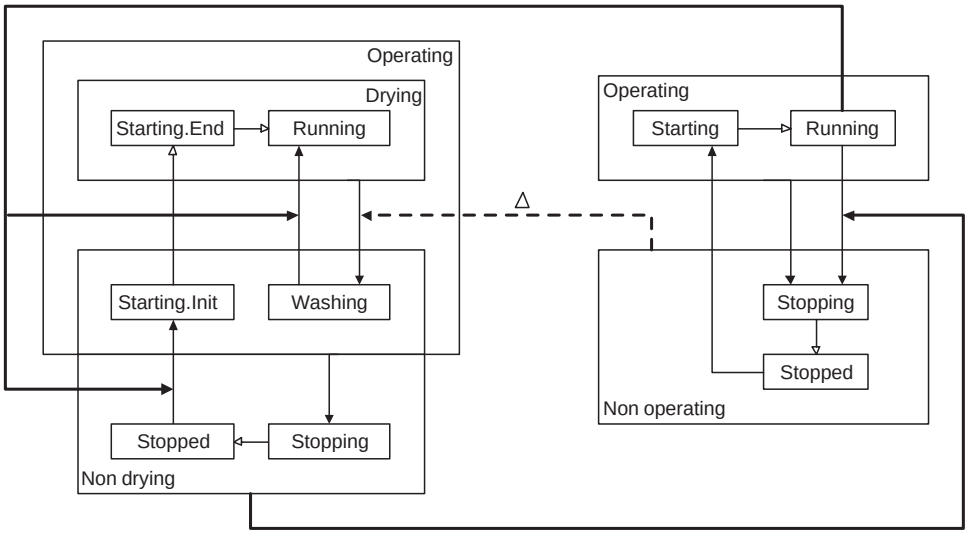


Fig. 10. EPT.Core versus SCT PCEDSTD.

```
Actions:
        :
IF M85[¬RA ∨ ε] THEN Alarm ENDIF;
        :
Transitions:

P40.50[¬RA ∨ ε] ∨ Dispergation\LT93101=MaxAW ⇒ T: Washing
```

Listing 1. Drying superstate of EPT.Core activity.

```
Actions:
        :
IF LT93030≤PP\EPT\LT93030\AL THEN Alarm ENDIF;
IF PT92656≤PP\EPT\PT92656\AL THEN Alarm ENDIF;
        :
Transitions:

P40.52[¬RA ∨ ε] ⇒ T: Stopping
```

Listing 2. Operating superstate of EPT.Core activity.

Fig. 10 illustrates very clearly one of the purposes of the nested states. So, for example, without the superstate Non-drying, the conditional dependence of the transition of the SCT operation from the Running state to the Stopping state would have a source at all four states of the superstate Non-drying of EPT.Core activity: Washing, Stopping, Stopped and Starting.Init. With regard to the causal dependence of the transition of the EPT.Core activity from the superstate Drying to the state Washing from the superstate Non-operating of SCT operation, without superstates there would be four dependence connections instead of one. At the diagram level, without superstates, there would be nine dependence connections instead of the existing four.

In this application, all the inter-operation state dependencies are of the domino-chains type, as defined before, and can be determined by observing the concrete dependent transitions in the PCEDSTD diagrams. So, for example, from the EPT.Core↔SCT PCEDSTD in Fig. 10 we can see that the stopping of the SCT is propagated to the stopping of the EPT.Core, which implies the existence of a domino-chain. The direction of the chain is denoted by the propagation (dashed) lines in Fig. 8. From this figure we can also see that the order of starting of operations runs in the direction from the top two operations (HG and SCT) in any order, through EPT to RVD. The stopping goes in the opposite direction: first RVD, then EPT and, finally, HG and SCT in any order.

Fig. 11 shows the relations between the EPT.Core and the Shaking activities. This is a typical case of relations between a main activity of an operation and one of its subactivities, where the main activity triggers (starts or stops) the subactivity. The requirements are as follows: The Core activity should start the shaking activity on its starting. At the stopping of the EPT.Core activity, Shaking activity should shake unconditionally (i.e. regardless of the current dP) for a determined time. After that the unconditional shaking stops and both activities terminate.

Translating these requirements to the analysis model gives the PCEDSTD in Fig. 11. This figure is in fact an explosion of the connections (dependence relations) between EPT.Core and Shaking PCEs in Fig. 8. It consists of the PCESTDs of both PCEs and of dependence relations between their concrete states and transitions. Note also that the PCESTD of the EPT.Core PCE was given above as example of PCESTD, whereas the PCESTD of the Shaking PCE (as well as all others) was not given. We can see in Fig. 11 that the main activity EPT.Core starts the Shaking activity upon entry in its Starting state, i.e. that the Starting state of the activity EPT.Core is propagated into the transition from Stopped to Regular Running (regular means depending on the dP on the

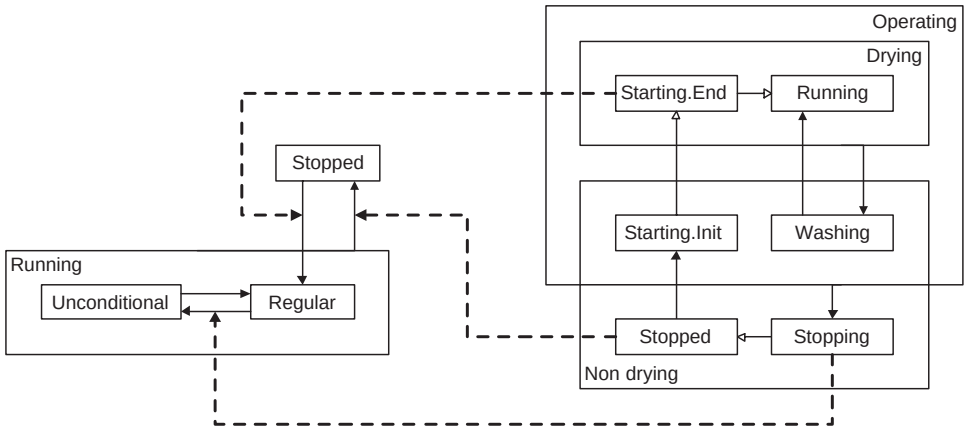


Fig. 11. EPT.Core versus Shaking PCEDSTD.

filter bags). At the end, we can see that the EPT.Core activity transition into Stopping state is propagated into transition to Unconditional Running of the Shaking activity. After a determined time, i.e. the duration of the unconditional shaking (specified in the processing definition of the EPT.Core activity Stopping state), in the EPT.Core activity a transition from Stopping to Stopped state occurs, which is propagated to the transition from Running to Stopped state of the Shaking activity. Note that the transition to Stopped state of the Shaking subactivity is from the Running superstate and not from the Unconditional substate, though the only possible transition is from the Unconditional substate. This solution makes the model more robust to eventual future changes. For example, consider the situation of adding a new transition in the EPT.Core activity from Operating superstate to a new state Fast Stopping, which performs stopping without unconditional shaking (this may become necessary in the case of some critical exception arising that would disallow shaking). The described solution allows the shaking subactivity to remain unchanged on such a change in requirements.

For an illustration of the specification of states at the lowest level, the (simplified) symbolic pseudo-code specifications of the states Starting.Init (Listing 3) and Starting.End (Listing 4) of the activity EPT.Core are given.

## 5. Comparison with similar approaches

It is interesting to compare the ProcGraph notation with similar notations. Two similar notations were found in the literature. One is SpecCharts (Vahid, Narayan, & Gajski, 1995), and the other is Statecharts (Harel, 1987), which is also one of the nine UML notations.

All three notations are based on a hierarchical/concurrent FSM model. However, there are also many substantial differences between them.

ProcGraph is more domain specific and at a higher level of abstraction, while SpecCharts and Statecharts are more general and at a lower level of abstraction. ProcGraph is based on three types of domain-specific diagrams, SpecCharts and Statecharts are based on only one type of diagram. ProcGraph includes two classes of processing description—the domain oriented hierarchical structure of procedural control entities (e.g. operations) and state-transition diagrams, which describe the behaviour of procedural control entities. SpecCharts and Statecharts define only behaviours (states), which may be decomposed to sequential or concurrent (orthogonal) sub-behaviours; in the context of process-control software this results in the problem of the mixing of abstraction levels (regarding procedural control) on the same diagram.

```
V96605[◉,∇●]                          (* close valve V96605, wait until closed *)
∇(LT93101≥PP\EPT\InitLT93101)         (* wait until level LT93101 reaches initial value *)
K96145.6.7[◎,∇○]                      (* open dampers K96145.6.7, wait until opened*)
P40.52[↑,∇↕]                          (* start pump P40.52, wait until running *)
TC90023.SP:=PP\EPT\TC90023\InitSP     (* set temperature controller setpoint *)
TC90023.MODE:=Auto                    (* set temperature controller to auto mode *)
K96150[◐:=PP\EPT\K96150\Openness]     (* set damper K96150 openness *)
∇(TT90018≥PP\EPT\TT90018\InitTarget)  (* wait until temp. TT90018 reaches target value *)
```

Listing 3. Starting.Init state of EPT.Core activity.

```
V96605[◎]                                          (* open valve V96605 *)
P40.50[↑]                                          (* start pump P40.50 *)
LC93101.SP:=PP\EPT\LC93101\SP                      (* set LC setpoint *)
LC93101.MODE:=Auto                                 (* set LC to auto *)
TC90023.SP:=RAMP(PP\EPT\TC90023\Slope)             (* send ramp to TC SP *)
∇(TT90023≤PP\EPT\TC90023\SP+PP\EPT\TC90023\Treshold)  (* wait for temperature *)
EXCEPTION
   ON LC93101[¬A] DO Warning END
   ON TT90018≤PP\EPT\TT90018\AL DO Alarm END
ENDEXCEPTION
```

Listing 4. Starting.End state of EPT.Core activity.

In ProcGraph the dependencies between procedural control entities are limited and given explicitly in the model; in SpecCharts and Statecharts these dependencies are implicit, i.e. hidden in the conditions, which is, in our opinion, one of their biggest weaknesses.

In Statecharts there is no notation for specifying the processing contents—there are only actions of states, which may start some activities that are not part of the notation. In SpecCharts the processing is a part of the notation, but in this notation only leaf states have contents (i.e. may be described by a sequential program). In ProcGraph all states, including superstates, may perform sequences of actions, described by a sequential program (or pseudo-code).

## 6. Conclusion and further work

The paper presented a procedure-oriented graphical notation for a process-control software specification called ProcGraph. The notation stems from field experience and has successfully been used in several industrial projects, for example more than ten subprocesses of the $TiO_2$ production process, or the process of polyvinyl acetate glues production. The main advantage of the presented notation in comparison with other approaches is its orientation to procedural control entities, resulting in better specification abstraction. The introduction of ProcGraph in the analysis and specification phase of the development process has had significant benefits both for the products and for the process. The product integrity is improved: the number of errors is reduced significantly in all life-cycle phases, particularly in the coding phase, where in some cases we achieve a near zero defect goal; the software maintainability is much better; the software robustness is very good owing to the powerful and, at the same time, simple abstraction that makes it possible to develop a very precise and extensive specification of handling exceptional situations. Further, the cost is reduced (by reducing development time), the documentation is standardised and consistent. The engineering process is more disciplined, and the spread of the domain knowledge inside and between the development teams is more efficient.

The approach is also very promising in terms of the future development of the automatic support of the development process. The goal is to achieve the (partially) automatic generation of code from a specification. It is possible to expand the existing development environments, based on the IEC 61131-3 standard, to support the ProcGraph notation. In the resulting environment the specification and the programming phases would be unified—the developer would start programming on a higher level of abstraction by using the ProcGraph notation, while the processing content of the states and the transitions (now specified by using symbolic notation) would be given in the IEC 61131-3 languages, e.g. Structured Text, Sequential Function Chart or Ladder Diagram.

Currently, a prototype tool is being developed, which translates a ProcGraph graphical model to a structure of function block diagrams in the MELSEC MEDOC+ environment (Mitsubishi, 1998). A description of the tool is in Kandare, Godena and Strmčnik (2003). Of course, to achieve the maximum benefit, the ProcGraph notation should become an integral part of the development environment.

## Acknowledgements

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

## References

Abou-Haidar, B., Fernandez, E. B., & Horton, T. B. (1994). An object-oriented methodology for the design of control software for flexible manufacturing systems. *Proceedings of the second workshop on parallel and distributed real-time systems* (pp. 144–149). Los Alamitos: IEEE Computer Society Press.

ANSI/ISA-S88.01 (1995). *Batch Control, Part 1: Models and Terminology*.

Beck, K., & Cunningham, W. (1989). A laboratory for teaching object-oriented thinking. *Proceedings of the OOPSLA89 Conference*, New Orleans.

Booch, G., Jacobson, I., & Rumbaugh, J. (1999). *The unified modelling language user guide*. New York: Addison-Wesley Publishing Company.

Černetič, J., Šubelj, M., & Selič-Podgoršek, N. (1994). Batch chemical treatment of $TiO_2$: Process design and control. *Computers & Chemical Engineering*, *18*, 195–199.

Davidson, C. M., & Mc Whinnie, J. (1996). The use of object oriented methods in PLC control system software development. *Proceedings of the 12th international conference on CAD/CAM robotics and factories of the future* (pp. 212–218). London: Middlesex University Press.

Godena, G. (1997). Conceptual model for process control software specification. *Microprocessors and Microsystems*, Vol. 20 (pp. 617–630). Amsterdam: North Holland.

Godena, G., & Colnarič, M. (2000). Exception handling for PLC-based process control software. *Microprocessors and Microsystems*, Vol. 24 (pp. 407–414). Amsterdam: North Holland.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, Vol. 8 (pp. 231–274). Amsterdam: North Holland.

Hopcroft, J. E., & Ullman, J. D. (1979). *Introduction to automata theory, languages, and computation*. Reading, MA: Addison-Wesley.

IEC 61131-3 International standard (1993). *Programmable controllers—Part 3: Programming languages*.

Jacobson, I., Ericsson, M., & Jacobson, A. (1995). *The object advantage*. Wokingham, England: Addison-Wesley Publishing Company.

Kandare, G., Godena, G., & Strmčnik, S. (2003). A new approach to PLC software design. ISA Transactions, 42, Melville, NY.

Mitsubishi electric Europe (1998). *MELSEC MEDOC plus IEC Programming and Documentation System Reference Manual.* Germany: Ratingen.

Rossetti, V., Rizzetti, G. C., Wohlgemuth, W., Wheeler, S., Citron, G. C., & Tresse, S. (1995). Heracles—automation design and control made easy, computer applications in production and engineering. *Proceedings of CAPE '95* (pp. 479–486). London: Chapman & Hall.

Schlaer, S., & Mellor, S. J. (1992). *Object lifecycles*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall Building.

Šel, D., Milanič, S., Strmčnik, S., Hvala, N., Karba, R., & Šuk-Lubej, B. (1999). Experimental testing of flexible recipe control based on a hybrid model. *Control Engineering Practice*, *7*(10), 1191–1208.

SIEMENS (1999). SIMATIC PCS7, Technological Blocks Manual.

Storr, A. (1997). Modelling and Reuse of Object-oriented Machine Software. *Proceedings of the European Conference on Integration in Manufacturing* (pp. 475–484). Dresden, Germany: Technical University of Dresden.

Vahid, F., Narayan, S., & Gajski, D. D. (1995). SpecCharts: A VHDL front-end for embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *14*(6), 694–706.

Wheeler, G. C. W. S. (1995). Application of object-oriented techniques to the design and operation of factories. *IEE Colloquium on Intelligent Manufacturing Systems*, Digest No. 1995/238, IEE, London, 3/1-6.

Žele, M., & Juričić, Đ. (1997). Fault detection of dosing paths in a $TiO_2$ production unit. *Copernicus Project*, 7684CT94-02337.