Chapter 12
A New Approach to Control Systems Software Development

Giovanni Godena, Tomaž Lukman, and Gregor Kandare

12.1 Introduction

This chapter addresses the software development of large-scale control and automation systems that are based on the use of Programmable Logic Controllers (PLCs) and programming languages according to the IEC (International Electrotechnical Commission) 61131-3 standard [18]. The focus of the chapter is mainly on the development of control systems for continuous processes, although most results are also usable for batch and discrete processes. The control systems that can be found in the specified scope belong to a broad range of industrial sectors, including chemical, pharmaceutical, food processing and other industries. The classes of these control systems typically involve hundreds or thousands of signals, dozens of control loops, and have to cope with the hybrid nature of the processes. Interestingly, the complexity of the development, operation and maintenance of the software for such kinds of systems is not so much associated with basic control (continuous behaviour), aimed at achieving and maintaining the desired state of the process variables, but more with so-called procedural control (discrete behavioural control), which performs a sequence of activities that ensure that the goals of the system or process are achieved.

According to the estimates of Boeing and Honeywell, software development consumes 60–80 % of engineering effort [15], and procedural control software, as stated above, represents its major and most complex part. Therefore, it is of paramount importance how this software is developed and what its attributes are. Unfortunately, software engineering (SE) state-of-the-practice in the domain of industrial process control has a low maturity level compared to SE practices in other domains (e.g., business information systems). Because of this low maturity, SE in this domain is

G. Godena (✉) · T. Lukman
Department of Systems and Control, Jožef Stefan Institute, Ljubljana, Slovenia

G. Kandare
Magna Steyr Battery Systems GmbH & Co OG, Graz, Austria

S. Strmčnik, Đ. Juričić (eds.), *Case Studies in Control*, Advances in Industrial Control,
DOI 10.1007/978-1-4471-5176-0_12, © Springer-Verlag London 2013

failing to address the steadily increasing complexity of control systems and the demands of the market: short time-to-market periods, high-quality software and an efficient development process [19]. The main causes of this immaturity are the following deficiencies [7, 10, 12]:

- a focus on coding and testing phases with little activity in the earlier software lifecycle phases;
- reliance on the programming skills of individuals instead of building on conventions, standards, rules and advanced software engineering concepts and technologies;
- the use of inadequate abstractions, which are mainly device-centric;
- a low degree of reuse of artefacts and knowledge, in particular those belonging to early lifecycle phases;
- a high degree of routine time-consuming and error-prone development tasks due to the low expressive power of the programming languages defined by the 61131-3 standard; and
- a low degree of development process automation.

In order to successfully overcome the issues mentioned above, methodologies for the engineering of process control software including tools that offer support for the whole development cycle are needed [9]. Many attempts to overcome the low level of maturity through a variety of different technologies (methods, tools, or processes) have been made.

The aim of this chapter is to present a new approach to process control software engineering, based on an innovative domain-specific modelling language, ProcGraph, and on the defined development process, supported by a tool suite implemented as an integrated development environment (IDE). The IDE includes automatic code generation. The approach builds on more than 20 years of experience in industrial process control software development and the application of the modelling language ProcGraph in more than 20 real industrial projects in the past 15 years.

The chapter is organised as follows. Modern approaches to process control software development and the actual challenges in the domain of process control software are discussed in Sect. 12.2. An overview of the domain specific modelling language ProcGraph is given in Sect. 12.3, followed by a description of the IDE and the software engineering process in Sect. 12.4. A sample application implementing the control system for the grinding of calcinated titanium dioxide process is presented in Sect. 12.5. The chapter concludes with a discussion of theory/practice issues and the conclusions.

12.2 Modern Approaches and Challenges to Process Control Software Engineering

In order to improve the engineering of process control software (or systems) several state-of-the-art approaches have been proposed. However, none of the proposed

approaches properly addresses all of the main challenges of the process control domain. An overview of the main approaches and challenges is given in the next two subsections.

12.2.1 State-of-the-Art

The following paragraphs describe the three main groups of state-of-the-art process control software/system approaches.

Application of the IEC 61499 Standard The most recent standard in industrial process control and automation systems is IEC 61499 [32], an extension of the ideas of IEC 61131-3 with support for the design phase and for distributed process control systems. IEC 61499 compliant programs are based on networks of function blocks (FBs). Every FB has data inputs and outputs that are used by the algorithms, as well as event inputs and outputs that are used by the execution control chart (ECC). An ECC is a specific kind of statechart that defines behaviour through the sequencing of the algorithm invocations. However, this approach has not been accepted by the industry due to several deficiencies, e.g., a lack of tool support and reference implementations [32], powerful market players do not support it [23], and the automatic generation of code for PLCs is not supported [32]. To the best of our knowledge, no significant improvement in quality or productivity has been reported, presumably because it does not cover the analysis phase [10], which in our opinion is its main deficiency, and it does not introduce any process-centric abstractions on its own.

At this point another approach should be mentioned that is similar to IEC 61499 in terms of the abstractions it uses and its expressive power, namely MATLAB Simulink/Stateflow. According to [6], there is a natural complementarity between Simulink/Stateflow and IEC 61499 Function Blocks, where Simulink/Stateflow provides a nice environment for modelling and simulation of control and embedded systems, while Function Blocks are good for designing distributed control systems. On the other hand, Simulink/Stateflow can also be used in the design phase due to its C or PLC (IEC 61131-3) code-generation feature. However, in our opinion, this environment, similar to IEC 61499, does not cover the analysis phase and does not introduce any process-centric abstractions, but is based on design-centric abstractions. The fact that it is not based on high-level, domain-specific elements on one hand makes it very general, and on the other results in its elements being at a low level of abstraction (with regard to a specific domain) and hence more implementation (programming) than problem oriented.

The Unified Modeling Language Approaches that use the Unified Modeling Language (UML) e.g., [4, 8, 14, 37] or a combination of UML and IEC 61499, e.g., [22, 33, 35], have been proposed. UML is a typical representative of General-Purpose Modelling Languages (GPMLs), which are intended to support modelling in a larger set of domains that sometimes can be very dissimilar. Consequently,

GPMLs tend to be extensive, complex and less expressive in a particular domain. This holds true for UML 2.2 [31], which consists of 14 diagram types and a very large set of modelling elements. UML can be specialised for a particular domain or subsets of domains with the UML profiles mechanism. Nevertheless, UML profiles are not suitable for use in the process control domain due to the following deficiencies: they complicate UML even more [16, 24]; they do not (easily) allow the deletion or omission of parts of the UML metamodel [16]; it is not possible to fully customise the concrete syntax, mainly because of the existing tools [3, 39]; and the default UML semantics strongly influence and constrain the semantics of new languages that are defined with UML profiles [21]. In general, process control software developers are unfamiliar with UML (and UML profiles) and perceive it as too complex and lacking the necessary expressiveness for this domain.

Model-Driven Engineering The last group of approaches (specific example approaches are [9, 29, 34]) is based on the idea of Model-Driven Engineering (MDE) and modelling languages that are defined independently of UML. MDE relies on the systematic and disciplined use of precise models[1] throughout the software/system lifecycle [25]. The essential idea of MDE is to shift the attention from program code to models, in a manner that regards models as the primary development artefacts. MDE employs modelling languages and model transformations which should be supported by sophisticated software tools. Through the automation of (parts of) the development process, MDE has the potential to increase productivity and software quality. Domain-Specific Modelling Languages (DSMLs) [28] enable the modelling of software and systems using concepts and abstractions that are common to a specific domain. DSMLs have important advantages (e.g., the inherent reuse of domain knowledge) compared to GPMLs. Despite the potential of the MDE and DSML concepts, the majority of the specific proposed approaches based on them use device-centric abstractions, whose negative influences are described in the next subsection. Several of the representatives are still conceptual and lack support software tools which would provide automatic PLC code generation.

12.2.2 The Main Challenges of the Process Control Domain

In spite of the relatively large variety of different approaches mentioned above, none of them has been widely adopted by industry [7] due to their technical limitations, complexity and, most importantly, inability to properly address the challenges of the process control domain. These challenges must, in our opinion, be seen as basic requirements for every software engineering approach in the process control domain. According to the literature and our own experience, the main challenges are as follows:

[1]The term models in this context does not represent mathematical models but models (mainly graphical) which are defined with the formalisms that enable modelling of the structure and the behaviour of software.

The Use of Adequate Abstractions The process control software should, in our opinion, be modelled through adequate, process-centric abstractions (e.g., technological processes, operations, and activities) that expose the goals, i.e., the responsibilities of the parts of the control system under development. Most of the methods found in the literature do not primarily expose the goals, rather they expose the means for achieving these goals [12]. These approaches can be considered to be device-centric, since they most frequently define relevant entities among the tangible entities, i.e., physical devices such as pumps and valves. Such device-centric models are more complex, less abstract, and introduce implementation details in the early phases of the development, which hinders the development of an optimal system. The main problem of modelling the control system software with equipment abstractions lies in the resulting high coupling between the software modules, each controlling a single equipment entity [12].

The Reuse of Artefacts and Knowledge Employing the reuse of high-quality artefacts and knowledge has the potential to improve process control software engineering. This holds true for the reuse of well-tested and field-proven basic software blocks and modules performing basic control functions (according to the ISA-88 standard, [1]), which are primarily dedicated to establishing and maintaining a specific state of process equipment and process variables, and are the same or similar across a broad range of different processes. On the other hand, most of the process control software complexity and the needed development effort concern procedural control, which performs high-level, process-oriented activities, using the elements of basic control, in order to perform process-oriented tasks. Therefore, reusing elements of procedural control has a much higher potential than reusing elements of basic control. But, since procedural control varies substantially among individual projects, the focus should be moved to the reuse of knowledge and artefacts at a high abstraction level belonging to the early phases of the lifecycle where the similarity among individual projects is much higher.

Acceptance of Advanced SE Concepts by Process Control Software Developers Process control software developers are mainly electrical engineers with experience in the field of PLC programming [30]. The problem is that the developers are predominantly accustomed to using low-level constructs and languages. It is hard for them to use more abstract high-level constructs, models and modelling languages, which are essential for taming the increasing complexity of today's process control systems. What makes the situation even worse is that the existing modelling languages are extensive, complex and too general, which makes them inexpressive in the process control domain. Consequently, they are hard to use by process control developers.

Automated Software Engineering During the whole development cycle there are several activities that can be performed in a routine manner despite their complexity and volume. These activities, which are highly likely to be subject to human error, are amenable to automation. An empirical study performed by Colla et al. [7]

has shown that process control software development organisations do not employ automation during the development lifecycle. It is hard to transform the deliverables of the early development phases into the implementation of a process control system [7], because of the wide semantic gap between the high-level constructs of the modelling languages and the low-level constructs of the programming languages of the IEC 61131-3 standard. It is even more difficult to develop automatic transformations if inexpressive GPMLs such as UML are used. If the specifications are defined in a purely informal way (either textual or diagrammatic), which is predominant in the state-of-the-practice, it is impossible to automate the transformation [7].

The approach presented in this chapter addresses the identified challenges in various ways in order to increase the long-term possibilities of being adopted by process control software developers. The main elements of the approach are the domain specific modelling language ProcGraph, the defined development process and the tool-support for the (partial) automation of the development process.

12.3 Domain-Specific Modelling Language: ProcGraph

12.3.1 Background

The idea of developing the ProcGraph language was closely related to our experience gained with the realisation of control systems for demanding industrial processes. Our work in this area mainly consists of activities of the early development phases, such as requirements analysis and specification, and additionally a number of activities spanning multiple phases, such as quality assurance or verification. These activities are usually performed in close cooperation with engineering enterprises active in the domain of control systems development. Already in the early 1990s we were facing the problem of inappropriate abstractions and notations used in analysis and specification. Most of the problems that were encountered in this area originated from the general purpose nature of the methods for system analysis and specification that were popular at that time, and the CASE (Computer-Aided Software Engineering) tools supporting these methods. For this reason, we then started with the introduction of some modifications of the established analysis and specification methods. For example, we upgraded the well-known and at that time very popular Ward–Mellor method [38] with a metamodel which introduced domain specifics into the method [11]. The upgraded method was successfully applied in a number of industrial projects. Further development proceeded in the direction of domain engineering and in this context especially towards the development of a domain-specific process-oriented modelling language. Since at that time we were mainly involved in control software development for continuous processes, our emphasis in developing the language was placed on the domain of continuous process control.

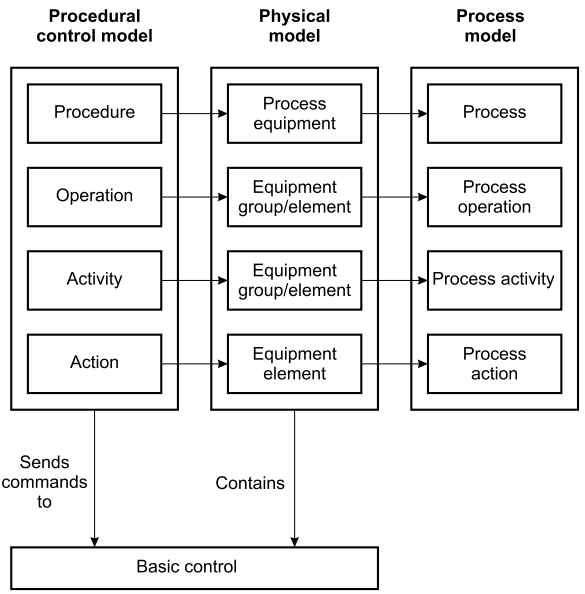


Fig. 12.1 The three models of continuous process control

12.3.2 The Domain of Continuous Process Control

The domain of process control can be divided into the sub-domains of control of continuous, sequential, batch and discrete processes. In this chapter the discussion is limited to the sub-domain of continuous process control. The resulting models are quite similar to the models for the domains of batch or sequential process control.

12.3.2.1 The Models of the Continuous Process Control Domain

With regard to software, continuous processes are less complex than, for example, batch processes. One of the complexity sources in continuous processes lies in the state-transition nature of the procedural control software. The main complexity source, however, can be found in reactions to exceptional situations, which should therefore be considered with particular care while building the conceptual model for continuous process control software. Figure 12.1 shows the three models which describe various aspects of continuous process control: the procedural control model, the physical model and the process model. Between the individual entities of the three models there is the following relation: the entities of the procedural control model, *in combination with* the entities of the physical model (or with their use), *provide process functionality to carry out* the entities of the process model. We adopted these models from [1] and adapted them to continuous processes.

12.3.2.2 Entities Model

The physical model represents a decomposition of the process equipment to individual entities—equipment groups and equipment elements. Each equipment element is defined by a corresponding equipment element type, i.e., it is classified into a proper equipment element type. The physical model *contains* the basic control. The purpose of basic control is to achieve and maintain the desired state of the process equipment or the process. It includes regulatory control and process equipment sequential control and monitoring. The physical model therefore belongs to the sub-domain of process equipment design and represents the universe of discourse between the system analyst and the specialists in control technology and process equipment. Although this part of process control systems may contain very complex elements (e.g., demanding control loops), it is generally much simpler (from the applicative software point of view) than the procedural control part of these systems. One of the main sources of the higher complexity of procedural control software is its significantly greater variability and consequently the lower availability of corresponding reusable components. On the other hand, for basic control there already exist libraries of reusable components that are an integral part of the integrated development tools for the software development of industrial controllers [27].

At this point let us mention that between the level of basic control and the level of controlled physical equipment entities there is one more very important level, namely the level of *safety interlocks*, which is responsible for personnel and environmental protection in the event of exceptional situations. This part of control functions is, by definition, separated from higher level control activities and no other control activity should intervene between this level and the controlled physical equipment.

The procedural control model contains the procedural control entities, which are the most important part of the process control software, since they are responsible for executing control algorithms by performing process oriented activities. The highest level procedural control entities are the *operations*, which perform larger process-oriented activities, resulting in physical or chemical transformation of the material or energy. The procedural control model therefore belongs to the sub-domain of process engineering and represents the universe of discourse between the system analyst and the process/chemical engineer. Each operation can be decomposed into *concurrent activities* (for modelling the implicit concurrency inside the operation), and each activity may be further decomposed into concurrent sub-activities. At the lowest level of such decomposition each activity is composed of *actions*, which execute *process actions* by issuing commands to and accepting signalisations from the *basic control*.

During the decomposition of the procedural control, particular attention must be paid to the criteria of good modularisation (high internal cohesion of the entities and the lowest possible coupling between them). Good modularisation is crucial for the successful management of the complexity of the control systems development, use and maintenance.

The conceptual model of continuous process control must also include the human operator and his interactions with the control system. The model including the

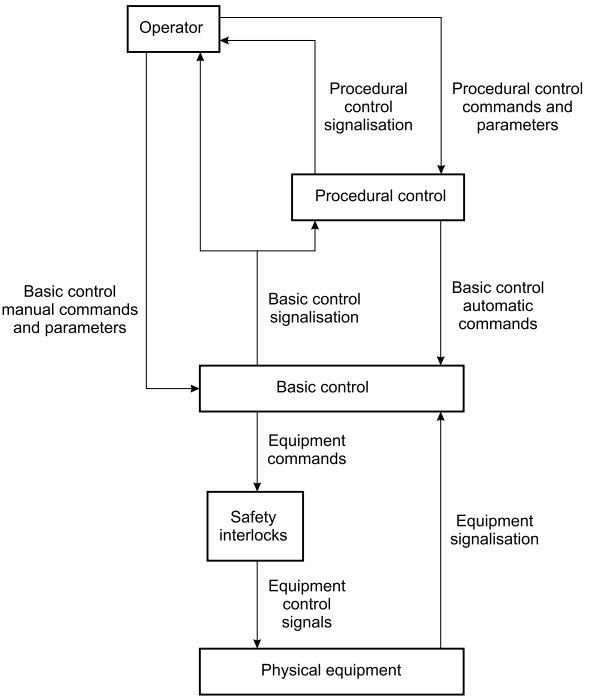


Fig. 12.2 Connections between the control system entities

information flow between the operator, the procedural control, the basic control, the safety interlocks and the physical equipment entities is shown in Fig. 12.2.

Figure 12.3 shows the conceptual model of the continuous process control domain. The model details the relations between the physical model entities and the procedural control entities using an Entity Relationship (ER) diagram. The definitions of the symbols appearing in the ER diagram are given in Fig. 12.4.

12.3.3 ProcGraph Language Requirements

Due to the aforementioned fact that in procedural control there is much more complexity and variability between the individual applications than is the case in basic control, the central part of a ProcGraph model should be information about the procedural control, and not information about the underlying basic control.

In determining the syntax and semantics of the modelling language ProcGraph we began with the following principal requirements:

1. The language must include elements that are suitable for describing the dynamic behaviour of reactive systems, since process-control systems are a subset of real-time systems, and hence reactive systems.

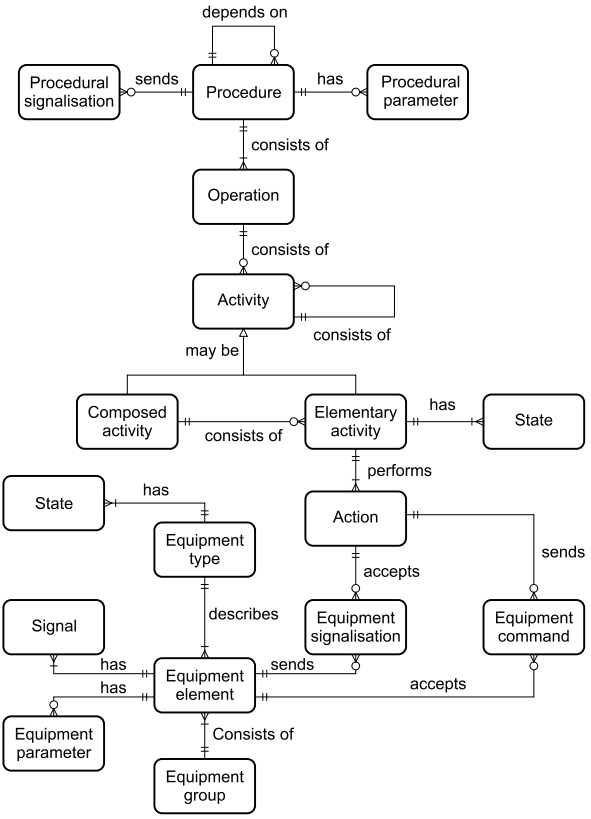


Fig. 12.3 Entities of the physical and procedural control models

2. The highest-level abstractions of the language should be goal-oriented and problem-oriented, which means that the highest-level model elements should represent the high-level procedural control entities.
3. The language must be designed so that the coupling between the individual elements of the model is minimised and explicitly visible at a high level in the model.

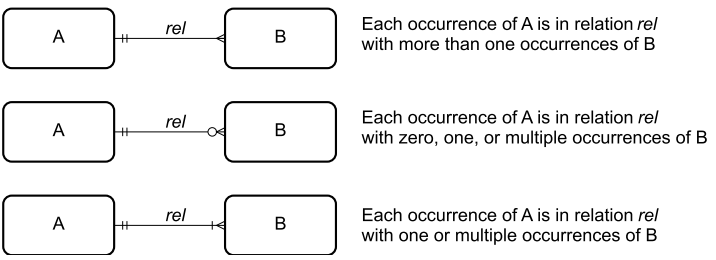


Fig. 12.4 Symbol definitions for the ER diagram

Given the above-listed principal requirements, the following lower level requirements were determined. A specific graphic modelling language is needed, including a smaller number of diagrams, that is closely related to the domain of process control, especially its procedural control entities. These entities should be the main elements of the language. The modelling language must also allow for decomposition of the procedural control entities into new entities at a lower hierarchical level. The behaviour of the procedural control entities at the lowest hierarchical level should be described by a variant of an extended finite-state machine (FSM). Another important aspect in the design of the modelling language that should be properly addressed is the notation of the synchronisation and the interdependence of the procedural control entities. The language must support the developer in minimising the coupling among the procedural control entities, which is the most important attribute of good modularisation. Other known notations (such as UML and Schlaer–Mellor [26]) implement the synchronisation by inter-object communication, which is too general and, in particular, too unlimited. In our view, the modelling language should as much as possible limit the number of possible types of coupling, i.e., the number of different types of dependency relations between the procedural control entities. These dependency relations must, of course, also be part of the graphical notation, and so appear explicitly and at a very high level in the model. This is, in our opinion, the best way to maintain good control of the number of these dependencies and, therefore, also to minimise the coupling.

12.3.4 Main Language Elements and Diagram Types

Based on the requirements stated in the previous section, the main information types and language elements were defined. The ProcGraph language consists of three different diagram types and several processing sequences written in a special symbolic language or in IEC 61131-3 Structured Text language. The three types of diagrams are as follows: a procedural control entities diagram (hereinafter referred to as an entities diagram, ED), a procedural control entity state-transition diagram (hereinafter referred to as a state transition diagram, STD), and a procedural control entities state-transition dependencies diagram (hereinafter referred to as a state dependencies diagram, SDD).

The graphical part of a ProcGraph model, expressed by means of the three abovementioned diagram types, represents a high-level behavioural structure, which has to be filled with specific, finely granulated processing definitions. Only this processing performs the intended functionality of the control system, including equipment control, sequential control and control loops of any kind, while the purpose of the graphical part of the language is just to provide the high-level, domain-specific framework that supports an optimal modularisation of the software.

The example excerpts of a ProcGraph model shown in Fig. 12.5 will be used as an aid to explain the elements of this language.

The root diagram is an entities diagram (ED), which contains Procedural Control Entities (PCEs) (i.e., operations, activities, or sub-activities) and potential composite

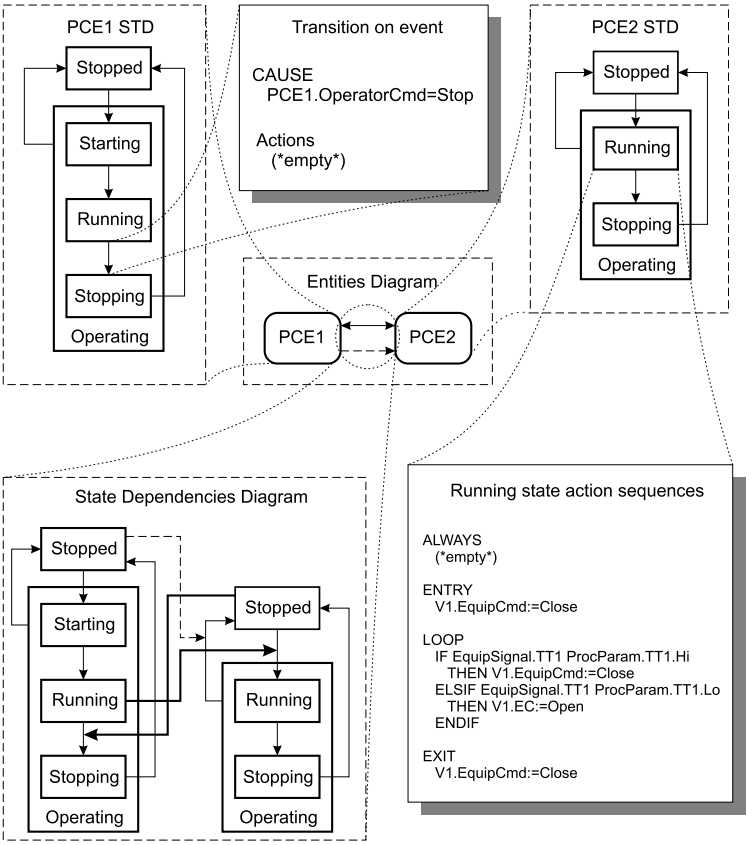


Fig. 12.5 Schematic structure of a ProcGraph model

dependencies between them. A PCE can be either elementary or a superPCE, which is decomposed into another ED containing a set of its subPCEs. An example ED with two elementary PCEs and a composite dependency is shown in the centre of Fig. 12.5.

The next diagram type is an STD, which defines the behaviour of a particular elementary PCE. Such a diagram consists of states and transitions, where state hierarchies of super-states and sub-states in the same diagram are possible. The super-states are aimed at avoiding the repetition of information by grouping the actions and/or transitions and/or dependency relations common to a number of (their sub-) states. The state for which the action sequences are currently being executed is called an active state. Each PCE has only one active elementary state at a time. In the case this state is contained in any super-states, then all of them are also active. An STD is an extended FSM, which differs from other extended FSM variants (e.g., Statecharts and a UML state diagram) as regards the following details:

- it includes two types of states, namely durative and transient;
- the processing is organised into a richer set of action sequence types, namely entry, loop, exit, transient, always and specific entry (a sequence of the latter type is located in a transition, but represents the specific entry actions of that transition's target state);
- all action sequences have a duration;
- overlapping super-states;
- two transition types (i.e., the transition on event and the transition on completion).

The transition on event, which is denoted by a filled arrowhead, is fired when the source state of the transition is one of the PCE's active states and the firing condition value becomes true. The transition on completion, which is denoted by an empty arrowhead, is fired when the source state of the transition has finished its processing. An example STD is shown in the upper-left and another in the upper-right corner of Fig. 12.5.

All the action sequences are defined either with ProcGraph's symbolic language[2] or with the ST language (which was used in Fig. 12.5) of the IEC 61131-3 standard.

The SDD is an explosion of a composite dependency, which exactly defines the mutual behaviour dependencies between the two PCEs it connects. An SDD consists of the STDs of two interdependent PCEs and a set of elementary dependencies. The lower-left corner of Fig. 12.5 shows an example SDD. A dependency can be either a conditional dependency, which is denoted by a normal line with a filled arrowhead, or a propagation dependency, which is denoted by a dashed line with a filled arrowhead. The transition that is the sink of a conditional dependency can only be fired when the source state of this dependency is active. For example, in Fig. 12.5 the transition between 'Stopped' and 'Running' of PCE2 can only be fired if 'Running' is the active state of PCE1. The transition that is the sink of a propagation dependency is fired when the source state of this transition and the source state of the dependency are both active. For example, in Fig. 12.5 the transition between 'Operating' and 'Stopped' of PCE2 is fired when 'Stopped' is the active state of PCE1 and 'Operating' is the active state of PCE2.

Three additional features of the SDD are delayed propagations, combined multi-source dependencies, and proxy states from other STDs. A delayed propagation occurs with a given delay after the causative event occurs and is denoted by the symbol Δ. A multi-source dependency is an incoming conditional or propagational dependency that is a result of a logical expression containing conditional or propagational dependencies as operands. The symbol for the AND operator is $\wedge$, and the symbol for the OR operator is $\vee$. A proxy state from another STD is used to combine multi-source dependencies from different STDs.

All the behaviour dependencies between two PCEs that are defined in an SDD are summarised by the shape of a composite dependency, which shows a union of the defined elementary dependencies. For example, in Fig. 12.5 the composite dependency in the entities diagram shows that between PCE1 and PCE2 there are one

[2]This language is not presented in this book because it is not supported by the current version of the IDE. For more information, please see [7].

or more conditional dependencies in each direction and one or more propagational dependencies directed from PCE1 to PCE2, which can also be seen in the SDD.

12.3.5 Experiences Related to the Use of ProcGraph

ProcGraph has been successfully used over the last 15 years in more than 20 industrial projects, ranging in size (expressed in a commonly used number-of-signals-based metrics) from a couple of hundred to a couple of thousand signals. Some of the most interesting projects include PVA glue production, resin synthesis in paint and varnish production and several sub-processes of the titanium dioxide production process (ore grinding, ore digestion, hydrolisis, calcinate grinding, gel washing, chemical treatment, pigment washing, pigment drying, pigment micronisation). The introduction of the ProcGraph language in the analysis and specification phase of our development process has had important positive effects, both on the products and on the development process. Product integrity has improved, costs were reduced (due to the shorter development time) and the documentation is standardised and consistent. The engineering process is more disciplined, and the transfer of the domain knowledge between the development groups is more efficient.

By introducing the ProcGraph language into our development process, we have partly addressed the first three of the five challenges listed in the previous section, i.e., the use of adequate abstractions, the reuse of artefacts and knowledge, and the acceptance of SE concepts by process control SW developers. However, the language was used for the specification of control software on "paper", and these specifications had to be manually transformed into PLC code. To really take advantage of the language, and at least partly address the other challenge of software development in the process control domain (i.e., automated software engineering) an additional development step was required. This step comprised the design and implementation of an IDE for ProcGraph and the definition of the software engineering process supported by this environment. The IDE and the engineering process will be presented in the next section.

12.4 The Integrated Development Environment and the Software Engineering Process

12.4.1 The Integrated Development Environment

The aim of the IDE is to support the process of modelling the procedural control processing in the form of ProcGraph constructs using appropriate graphical tools, and to enable automatic code generation from the ProcGraph constructs obtained in this way. The currently existing IDE is a prototype consisting of a *model repository*,

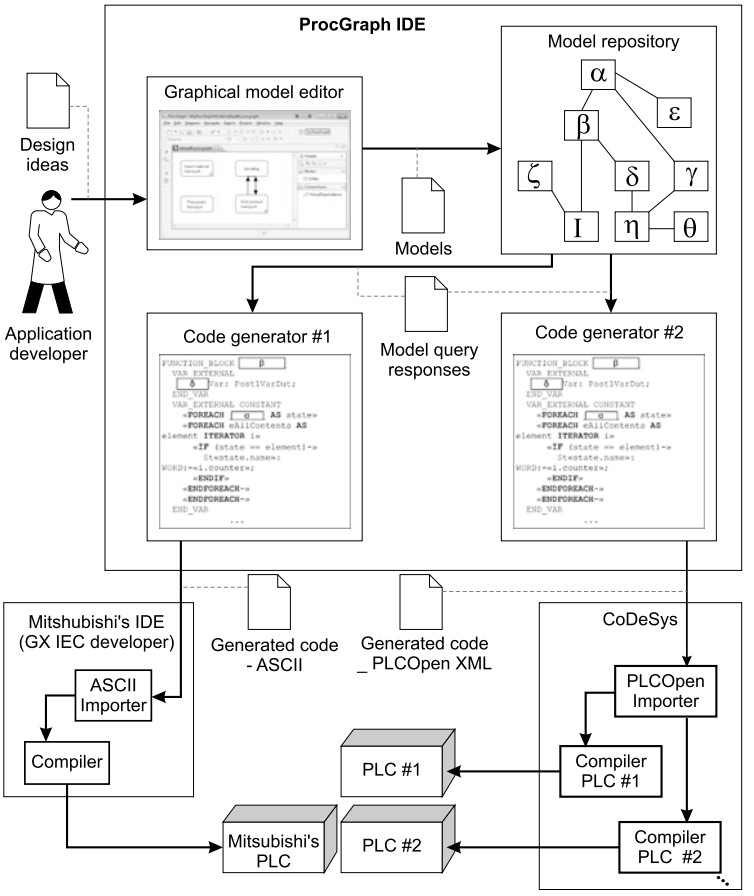


Fig. 12.6 The information flow between the components of the IDE

a *model editor* and two *code generators*. The components of the IDE and their interaction for the purpose of automatic PLC code generation are shown in Fig. 12.6. The sinks of the information flow chain are two external tools that use the generated code (the development environment for Mitsubishi PLCs and the CoDeSys tool [36]). The following subsections present each of the mentioned components.

12.4.1.1 Graphical Model Editor

The model editor basically serves as a user interface, with which users (i.e., process control software engineers) can construct the models through an intuitively understandable diagrammatic notation representing domain-specific abstractions. Screenshots of the model editor can be seen in the example in Sect. 12.5.

12.4.1.2 Model Repository

The model repository stores all the necessary data that has to be defined according to the ProcGraph DSML independently of its representation (i.e., the notation), but according to the structure of the models. It also has to enable the manipulation and querying of the ProcGraph models through services which can be invoked by other components (e.g., a code generator).

The model repository for our IDE was developed with EMF (Eclipse Modelling Framework) [5], which is the central tool for creating structural model repositories for the Eclipse platform. EMF automatically generated the model repository for ProcGraph, based on the formal definition of ProcGraph.

12.4.1.3 Code Generator

The purpose of the code generator is to automatically generate PLC source code from ProcGraph models. The code generator eliminates human coding errors, which are common when model transformations are carried out manually by developers. According to the IEC 61131-3 standard, there are five procedural-imperative programming languages which can represent the source code for PLCs: instruction list (IL), structured text (ST), ladder diagram (LD), function block diagram (FBD) and sequential function chart (SFC). The model transformation rules of the presented approach define how the ProcGraph models are transformed into code in a combination of FBD and ST, which are semantically nearest to the ProcGraph language. Currently eight high-level model transformation rules have been defined. An example of a model transformation rule which defines how STDs are transformed into the target language is presented in Algorithm 12.1.

Algorithm 12.1 Each STD is transformed into an FBD, where each top-level state (i.e., a state that has no super-state) is transformed into an FB instance in a separate network. Because only one elementary state (together with all its super-states) of a PCE can be active, guards are placed at the beginning of the networks to ensure that only one FB instance is active at the same time:

- If the FB instance is a transformed elementary state, only one guard (it checks if this elementary state is the currently active one) is generated.
- If the FB instance is a transformed super-state, then a guard for each of the elementary sub-states of that super-state is generated (the guards are joined together with a logical OR).

The code generator was realised in the openArchitectureWare tool [13]. This tool includes a code generation engine based on code generation templates. A code generation template consists of a static (i.e., invariant) and a dynamic part. During the generation process the dynamic parts are filled in with the information that was queried from the ProcGraph model. Currently our IDE has two code generators: one for the GX IEC Developer [20] from Mitsubishi, because of our long cooperation with this company, and one for the PLCOpen format, because it is supported by an increasing number of PLC IDEs (e.g., CoDeSys [36]).

Algorithm 12.2 shows a code generation template excerpt which creates an FBD for each PCE in the root ED. It first creates a variable for each FB in the VAR block and then starts to draw the graphical part of the FBD. A separate network is created for each PCE in the block «`FOREACH subEntities AS Entity-`». The function B creates an FB of the type user function block (denoted by B_FB), with the FB name and the instance name taken from the acronym of the current PCE queried from the model (denoted by «`getAcronym(Entity.name)`»), the x and the y coordinates of the upper-left corner of the FB, and the x (the width of the FB is calculated dynamically based on the length of the FB acronym, which is done by invoking the getWidth function) and y coordinates of the lower-right corner of the FB.

Algorithm 12.2

```
«DEFINE transformRootEntities FOR procgraph::Entity-»
PROGRAM MAIN_PRG_LD
(**)
(**)
        VAR
                «FOREACH subEntities AS Entity -»
                «getAcronym(Entity.name)»: «getAcronym(Entity.name)»;
                «ENDFOREACH-»
        END_VAR
'LD'
BODY
        WORKSPACE
                NETWORK_LIST_TYPE := NWTYPELD;
                ACTIVE_NETWORK := 0;
        END_WORKSPACE
        «FOREACH subEntities AS Entity-»
        NET_WORK
                NETWORK_TYPE := NWTYPELD;
                NETWORK_LABEL :=;
                NETWORK_TITLE :=;
                NETWORK_HEIGHT := 6;
                NETWORK_BODY
                        B(B_FB,«getAcronym(Entity.name)»,«getAcronym(Ent
                        ity.name)»,12,2,2 + «geWidth(getAcronym(Entity.name).
                        length.toString())»,4,);
                        L(1,0,1,6);
                END_NETWORK_BODY
        END_NET_WORK
        «ENDFOREACH-»
END_BODY
END_PROGRAM
«ENDDEFINE»
```

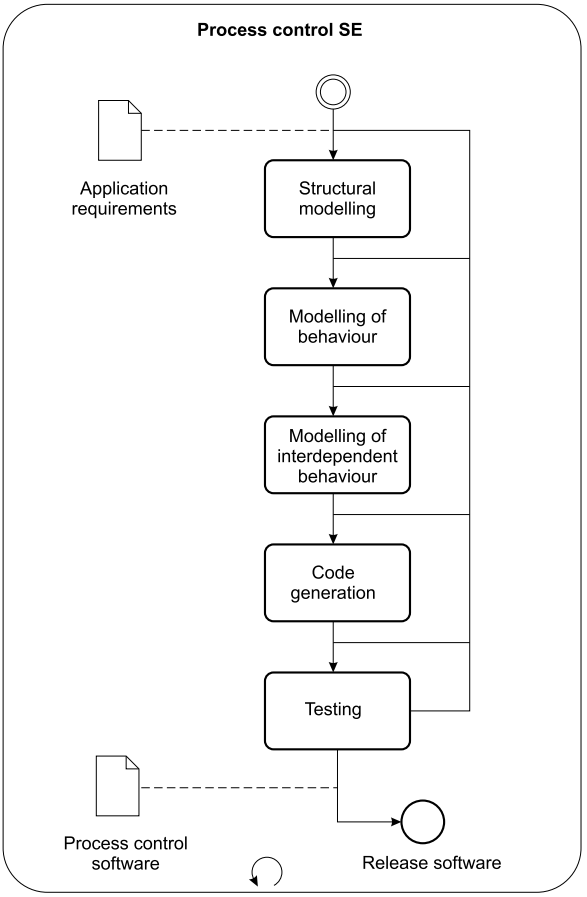


Fig. 12.7 The software engineering process

12.4.2 Software Engineering Process

The process control software engineering process is composed of the development activities that are shown in Fig. 12.7, and relies on the provided IDE. The inputs into this process are the requirements for the process control system/software. The requirements that are needed by the presented approach consist of a Piping and Instrumentation Diagram (P&ID) according to ISA S5.1 [2], and supporting documents (e.g., informal functionality and safety requirements).

The next subsections describe each development activity.

12.4.2.1 Structural Modelling

At the beginning of the structural modelling development activity we analyse the requirements to identify the main operations that should control the considered technological process. Each operation should be named according to the goal it pursues.

The next step is the device allocation for each operation. To avoid potential problems, each equipment entity should be controlled by only one operation. The initial operation list and their allocated equipment entities may and probably will change during the development process. Essentially, the operations of the technological process were chosen well if they are highly cohesive and weakly coupled.

After the operations have been identified, the complex or extensive ones should be decomposed into activities and, if needed, further into sub-activities that have behaviour which is moderately complex and consequently not hard to manage.

In the ProcGraph IDE, all the identified operations (represented by top-level PCEs) are placed in the root ED and all the interdependent operation pairs are connected with composite dependencies. However, it is often not clear in the beginning which operations (if any at all) are interdependent. Therefore, these composite dependencies can also be drawn later, when they emerge during the modelling of the behaviour.

The result of this development activity is a partial ProcGraph model, with one or more initial EDs.

12.4.2.2 Modelling of Behaviour

Each elementary PCE has to be described by a STD, which defines the states of the PCE and the transitions between them. Typically, we start building the STD with a set of typical or standard states which comprises the elementary states *Stopped*, *Starting*, *Running*, *Stopping* and *Fast Stopping*, and the corresponding transitions. Once the needed typical states and transitions are introduced, the investigation of the information (of various kinds) that is common to more states results in the introduction of some super-states in order to avoid the repetition of information. The next task is to define the detailed behaviour through the action sequences of the states and transitions and the causes of the transitions. The action sequences define the processing needed to achieve a process-oriented goal, which is achieved by using the functions of individual devices as the means to achieve that goal. Note that the current IDE prototype supports the definition of the action sequences through the ST language.

The deliverable of this development activity is a refined ProcGraph model, with information about the independent behaviour of the identified software parts.

12.4.2.3 Modelling of Interdependent Behaviour

The aim of this development activity is to define the interdependent behaviour between pairs of elementary PCEs that are behaviourally dependent, which is modelled with SDDs. The system analyst first has to draw a composite dependency in the ED between the two interdependent elementary PCEs and then explode this dependency into a SDD.

The deliverable of this development activity is a complete model, which is used for code generation.

12.4.2.4 Code Generation

In this development activity, the code for a selected PLC platform (family or vendor) is automatically generated. The generated code is in the import format of the selected target PLC development environment. The currently developed code generators are mentioned in Sect. 12.4.1.3.

12.4.2.5 Testing

The last development activity is testing, which can be started after the generated code is imported into the target PLC development environment, where it is compiled and then downloaded onto the target platform. In this development activity an appropriate testing technique for PLCs (e.g., [17, 18]) should be followed. Our IDE and approach do not support reverse engineering, which means that all the changes in the code that are made manually are lost during the code regeneration. Therefore, it is recommended that the necessary changes are always made in the model, from which the code is then regenerated.

12.5 Sample Application

This section presents an example of applying the proposed approach to a process control scenario taken from an industrial project. The considered project is one of the above-mentioned in which the ProcGraph language was used in a "manual" manner (without tool support). In this section we present the reengineering of the considered project by using the extended approach and the new IDE. The example is presented through the development process activities that are described in Sect. 12.4.2.

12.5.1 Process Description and Control System Requirements

The aim of the project was to develop control software[3] for the grinding of calcinated titanium dioxide (or calcinate, in short), which is a sub-process of titanium dioxide (TiO_2) production. This system contained around 50 devices that included nearly 400 signals. The requirements for this system were supplied in the form of a P&ID, a signal list and an informal description of the functionality and the safety requirements. The simplified P&ID in Fig. 12.8, which is stripped of information about the devices and signals, shows a schematic of the technological process.

[3]Note that only software for procedural control is considered. Closed loop control is in this case limited to only a few simple control loops and will not be treated.

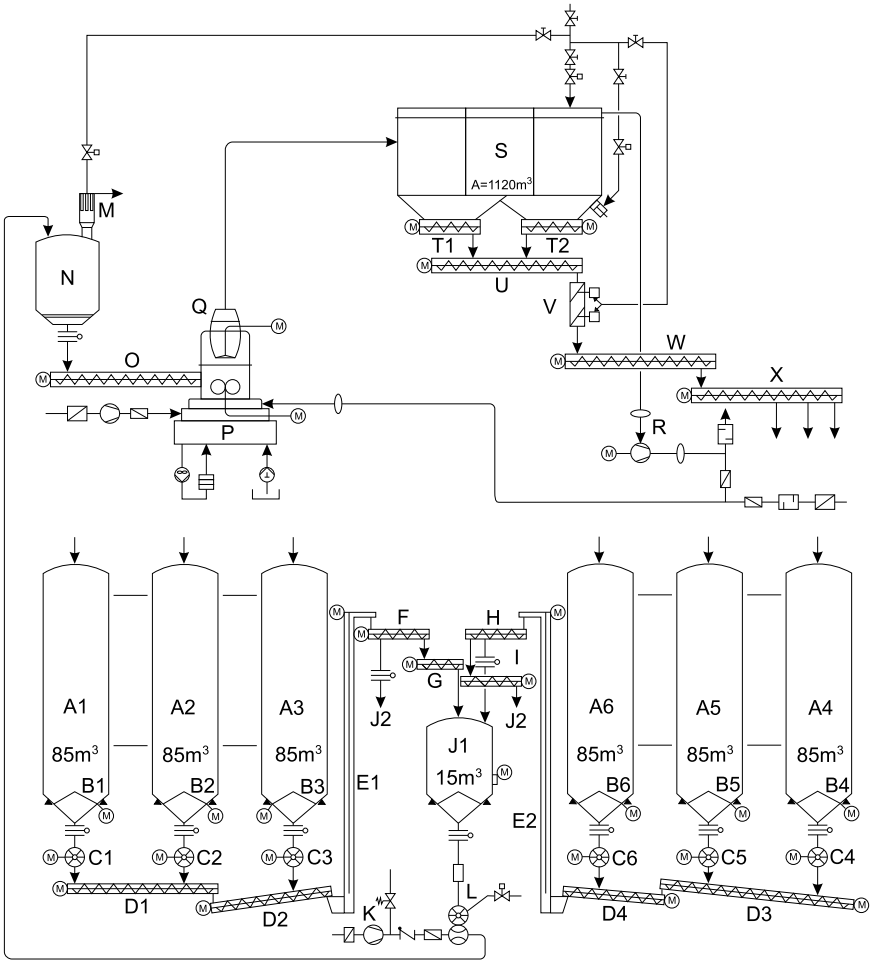


Fig. 12.8 Simplified technological scheme of the calcinate grinding sub-process

The calcinate grinding sub-process starts with the storage of the cooled calcinate in six silos, denoted as silos A1–A6. From each of these silos the material can be transported by means of vibrating sieves B1–B6, rotary valves C1–C6, screw-conveyors D1–D4, F, G, H and I, and elevators E1 and E2 into the intermediate silo J1. Any combination of the silos can be included in the calcinate dosing at a given time and the amount of material dosed from a particular silo can be controlled by the rotation speed of the rotary valves C1–C6. From the intermediate silo J1 the calcinate is pneumatically transported (powered by the rotary valve L and the fan K) to the silo N. The air flow provided by the fan R transports the grounded pigment from the mill into the selector Q, where the coarse fraction of the pigment is separated and fed back into the mill. The ground pigment is pneumatically transported

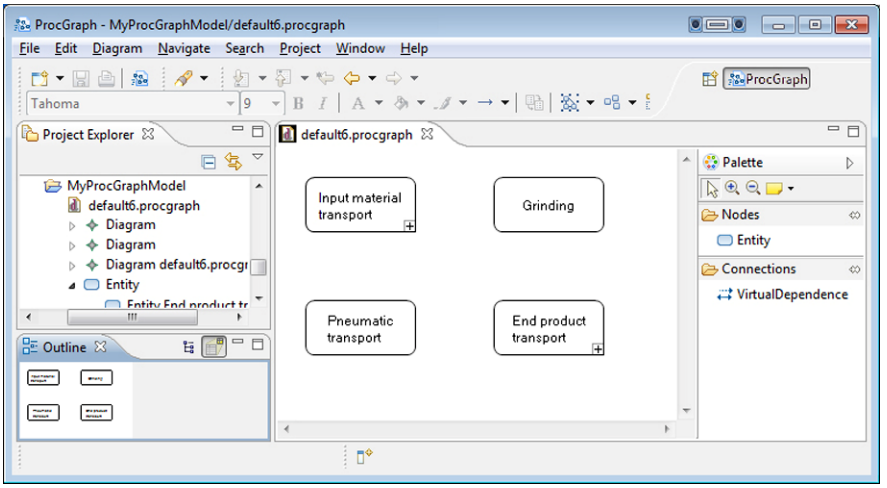


Fig. 12.9 The initial root ED for the sample process

into the bag filter S, where it is separated from the air. The ground pigment that is caught in the bag filter S is transported with the screw-conveyors T1, T2 and U through a double-hatch chamber (DHC) V to the screw-conveyors W and X, which transport it to the next sub-process (of the TiO_2 production process).

12.5.2 Development with the Proposed Approach

Due to the limited page space for this example, it is not possible to present it in its entirety; instead we had to choose between complete coverage with a few details (a broad and shallow approach) and partial coverage with more details (a narrow and deep approach). In order to give the reader the best possible insight into the presented system, we decided to show the high level view of all operations. The operations are elaborated to the level of individual STDs and SDDs, without detailed specification of the processing, except for one activity, for which we present excerpts of its low-level processing (through listings). Omitting the low-level details should not hinder a general understanding of the system considered.

We start with the construction of EDs. First we analyse the requirements and identify four operations of the calcinate grinding process, which are then placed in the initial root ED (Fig. 12.9): 'Input material transport' (using equipment A1 to I), 'Pneumatic transport' (using equipment J to M), 'Grinding' (using equipment N to R) and 'End product transport' (using equipment S to X). We decompose the 'Input material transport' and the 'End product transport' operations into activities that are placed in their respective subEDs. The reason why the 'Input material transport' is decomposed is the requirement "enable dosing an optional amount of the material from each silo among an arbitrary combination of them; also enable changing the combination of silos and their dosing speeds at any time". The 'End product

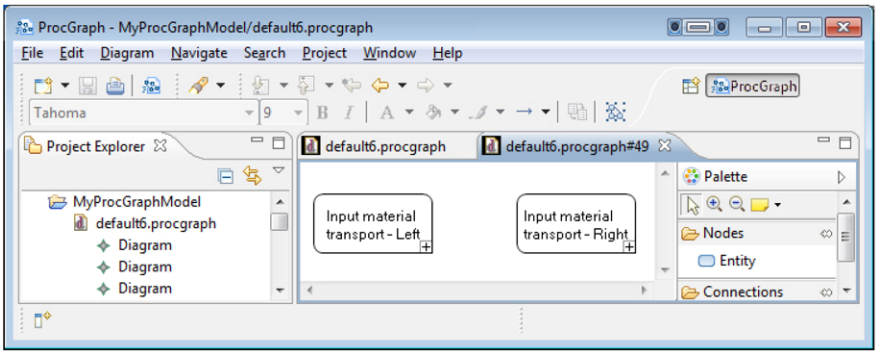


Fig. 12.10 The subED of the 'Input material transport' operation

transport' operation is decomposed according to the guideline that suggests the separation of cyclical behaviour and periodical behaviour into a main activity and an auxiliary activity. As mentioned above, we present only the high level view of all operations, while the only PCE for which we present the excerpts (through listings) that specify a part of the processing is one of the activities of 'Input material transport'.

The 'Input material transport' operation is decomposed into the 'Input material transport–Left' and 'Input material transport–Right' activities, which are shown in the subED in Fig. 12.10. Each of them presents one side of the 'Input material transport' operation.

The 'Input material transport–Left' activity is decomposed into four sub-activities (which are placed in the initial subED in Fig. 12.11), where the 'Screw-conveyor transport–L' sub-activity controls the transport path (consisting of screw-conveyors) into the next operation and each of the other three sub-activities perform the input of the calcinate from the three corresponding silos. It is assumed that each of the three material inputs has certain dependency relations with the 'Screw-conveyor transport' activity, but since the type of relation is not yet known, it is represented by "undefined-shape" (thick grey) connections in Fig. 12.11. Since the decomposition of the 'Input material transport–Right' activity from Fig. 12.10 is analogous to one of the 'Input material transport–Left' activities, its subED is neither shown nor discussed.

The 'Screw-conveyor transport–L' activity in Fig. 12.11 is not further decomposed into sub-activities (in other words, it is an elementary PCE); therefore the next step is to define its behaviour. This is done with the STD in Fig. 12.12, which was gradually constructed. Initially, the typical states and transitions, including the 'Operating' super-state, are placed in the diagram. Because the transition into 'Stopping' in this case is possible both from 'Starting' and 'Running', they are enclosed in the 'Executing' super-state to avoid the repetition of information.

At this point, it is time for the system analyst (or control software developer) to specify the operation's detailed behaviour by defining the state action sequences and the transition causes and action sequences expressed in the ST language. To

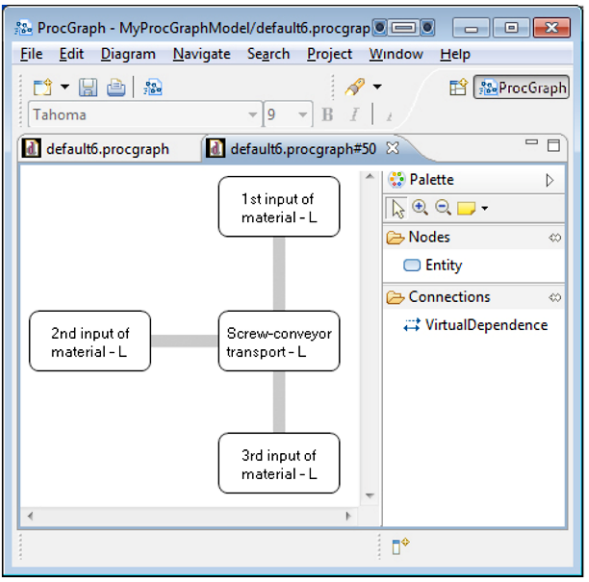


Fig. 12.11 The initial subED of the 'Input material transport–Left' activity

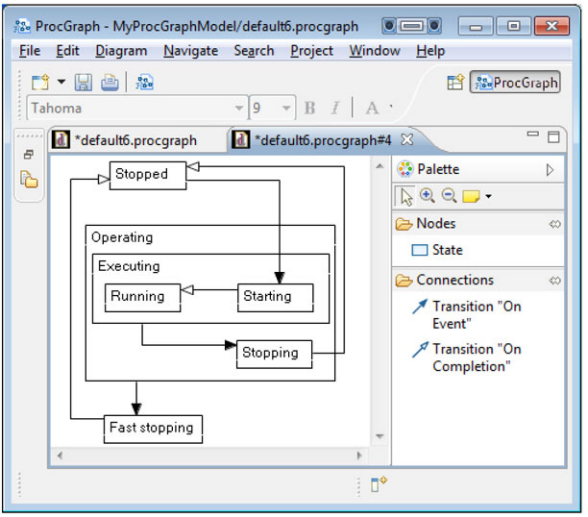


Fig. 12.12 The STD of 'Screw-conveyor transport–L'

demonstrate this, we present two state and one transition condition definitions for 'Screw-conveyor transport–L', namely the processing of the transient states 'Starting' and 'Stopping', and the conditions for the transition from the 'Operating' state to the 'Fast stopping' state.

The processing of the 'Starting' state, defined in Algorithm 12.3, sequentially turns on a set of devices and waits for confirmation of their activation by checking if their state is *on* and their drive shafts are rotating. These devices are the screw-conveyors G (M517) and F (M516), the elevator E1 (M659), and the screw-conveyor

D2 (M528). G, however, is turned on only if the destination silo is the silo J1 and not the silo J2, which is given by the PP_SCTL_Dest parameter. The inability of any of the listed devices to become activated in the expected time requires the auto-stopping of 'Screw-conveyor transport–L'. The possible causes for this are device failure, its non-remote-auto mode, or an interlock. Since these causes have to be detected not only in the Starting state, but also in the states Running and Stopping, this detection is assigned to their super-state 'Operating', more precisely to the condition of its transition to the 'Fast stopping' state (which is defined in Algorithm 12.5).

Algorithm 12.3
TRANSIENT:

```
CASE StepCounter OF
0: (* Check if the destination is the Silo J1*)
    IF (PP_SCTL_Dest = J1Silo) THEN

        (* Turn on screw-conveyor G*)
        M517_Command := ON;

        (* Check if screw-conveyor G is turned on and is rotating *)
        IF(M517_State = ON AND M517_Rotation = ON) THEN
            StepCounter := StepCounter + 1;
        END_IF;
    ELSE
        StepCounter := StepCounter + 1;
    END_IF;
1: (* Turn on screw-conveyor F*)
    M516_Command := ON;

    (* Check if screw-conveyor F is turned on and is rotating *)
    IF(M516_State = ON AND M516_Rotation = ON) THEN
        StepCounter := StepCounter + 1;
    END_IF;

2: (* Turn on elevator E1*)
    M659_Command := ON;

    (* Check if elevator E1 is turned on and is rotating *)
    IF(M659_State = ON AND M659_Rotation = ON) THEN
        StepCounter := StepCounter + 1;
    END_IF;

3: (* Turn on screw-conveyor D2*)
    M528_Command := ON;

    (* Check if screw-conveyor D2 is turned on and is rotating *)
    IF(M528_State = ON AND M528_Rotation = ON) THEN
        StepCounter := -1;
    END_IF;
END_CASE;
```

The processing of the 'Stopping' state, defined in Algorithm 12.4, sequentially turns off each device in the set and waits for confirmation that the current device is turned off. The devices in this set are the screw-conveyor D2 (M528), the elevator E1 (M659) and the screw-conveyors F (M516) and G (M517). After turning off D2, E1 and F, there is a delay to allow complete emptying of the succeeding transporting device, where the duration of each delay is given by the corresponding parameter (e.g., PP_SCTL_OffDelayM528AndM659 for D2). The delay after turning off F, however, occurs only if J1 is the destination silo.

Algorithm 12.4

TRANSIENT:

```
CASE StepCounter OF
0: (* Turn off screw-conveyor D2*)
     M528_Command := OFF;

     (* Check if screw-conveyor D2 is turned off *)
     IF(M528_State = OFF) THEN
          StepCounter := StepCounter + 1;
     END_IF;

1: (* Start delay timer *)
     DelayTimer(IN := TRUE, PT := PP_SCTL_OffDelayM528AndM659, );

     (* Check if the timer ran out *)
     IF(DelayTimer.Q) THEN
          StepCounter := StepCounter + 1;
     END_IF;

2: (* Turn off elevator E1*)
     M659_Command := OFF;
     (* Check if elevator E1 is turned off *)
     IF(M659_State = OFF) THEN
          StepCounter := StepCounter + 1;
     END_IF;

3: (* Start delay timer *)
     DelayTimer(IN := TRUE, PT := PP_SCTL_OffDelayM527AndM659, );

     (* Check if the timer ran out *)
     IF(DelayTimer.Q) THEN
          StepCounter := StepCounter + 1;
     END_IF;

4: (* Turn off screw-conveyor F*)
     M516_Command := OFF;

     (* Check if screw-conveyor F is turned off *)
     IF(M516_State = OFF) THEN
          StepCounter := StepCounter + 1;
     END_IF;
```

```
5: (* Check if the destination is the Silo J1*)
    IF (PP_SCTL_Dest = J1Silo) THEN

        (* Start delay timer *)
        DelayTimer(IN := TRUE, PT :=
        PP_SCTL_OffDelayM516AndM517, );

        (* Check if the timer ran out *)
        IF(DelayTimer.Q) THEN
            StepCounter := StepCounter + 1;
        END_IF;
    ELSE
        StepCounter := StepCounter + 1;
    END_IF;

6: (* Turn off screw-conveyor G*)
    M517_Command := OFF;

    (* Check if screw-conveyor G is turned off*)
    IF(M517_State = OFF) THEN
        StepCounter := -1;
    END_IF;

END_CASE;
```

Algorithm 12.5 specifies the auto-stopping conditions of the 'Screw-conveyor transport–L', which are detected in the 'Operating' state, precisely as it checks the condition for the transition to the 'Fast stopping' state. These failures are: the main switch is turned off, J1 is the destination silo and it is full (detected by the LS33809 sensor), or any of the material transporting devices (G when J1 is the destination silo, D2, E1 and F) is either not in remote-auto mode, its state is error or locked, or it is not rotating when its state is on.

Algorithm 12.5

CAUSE:

```
HS36890 = OFF OR
(PP_SCTL_Dest = J1Silo AND LS33809 = ON) OR
(PP_SCTL_Dest = J1Silo AND

    (M517_Mode <> RemoteAuto OR M517_State = Error
        OR M517_State = Locked OR (M517_State = ON AND
        517_Rotation = OFF))) OR
    (M528_Mode <> RemoteAuto OR M528_State = Error
        OR M528_State = Locked OR (M528_State = ON AND
        M528_Rotation = OFF)) OR
    (M659_Mode <> RemoteAuto OR M659_State = Error
        OR M659_State = Locked OR (M659_State = ON AND
        M659_Rotation = OFF)) OR
```

(M516_Mode <> RemoteAuto OR M516_State = Error
OR M516_State = Locked OR (M516_State = ON AND
M516_Rotation = OFF));

ACTIONS:
(∗ empty ∗)

The '1st input of material–L' sub-activity from Fig. 12.11 is further decomposed into an initial subED (shown in Fig. 12.13), which includes the '1st input of material–L–Core' main (cyclical) sub-activity and the 'Rotary valve control–1L' auxiliary (periodical) sub-activity. A composite dependency between these PCEs is drawn whose SDD is not yet defined, and therefore it has an "undefined shape". Based on the composite dependencies that are defined in Fig. 12.11, we also added 'Screw-conveyor transport–L' as a PCE proxy (i.e., an element that references the original activity, which is defined in the subED from Fig. 12.11) and connected it with a composite dependency to the '1st input of material–L–Core' sub-activity. SubEDs analogous to those shown in Fig. 12.11 are also created for the '2nd input of material–L' and '3rd input of material–L' sub-activities.

The next step is to define the behaviour of the '1st input of material–L–Core' sub-activity with the STD shown in Fig. 12.14. After adding the standard states and transitions in the STD, three special states are added, i.e., 'Starting–rotary valve on', 'Stopping–rotary valve off', and 'Starting of 'Screw-conveyor transport–L". The former two states are introduced to separate both starting and stopping into two states, at the point where the auxiliary activity has to be started and stopped. This separation enables the realisation of the starting and stopping by means of the propagation dependency mechanism. The 'Starting of 'Screw-conveyor transport–L" state is only a pure synchronisation state with empty action sequences, which only serves the purpose of issuing the start command to the 'Screw-conveyor transport–L' activity and then waiting for it to run, which is done by means of two propagation dependencies (one in each direction).

An investigation of specific auto-stopping causes reveals that the starting states have the same specific auto-stopping causes; therefore a 'Starting' super-state with one common auto-stopping transition is introduced.

The STD in Fig. 12.15 defines the behaviour of the 'Rotary valve control–1L' auxiliary periodic activity.

The next development step is the modelling of the interdependent behaviour between the sub-activities 'Rotary valve control–1L' and '1st input of material–L–Core'. This is done with the SDD in Fig. 12.16, which defines how '1st input of material–L–Core' turns on the 'Rotary valve control–1L' auxiliary activity by being in the 'Starting–rotary valve on' state, and stops it by being in the 'Fast stopping' or the 'Stopping–rotary valve off' state. If the IDE supports overlapping super-states, which is envisaged for future versions, the 'Fast stopping' and 'Stopping–rotary valve off' states could be included in a new super-state. In that case, only one propagation dependency from this super-state would be enough to stop 'Rotary valve control–1L' instead of the current two.

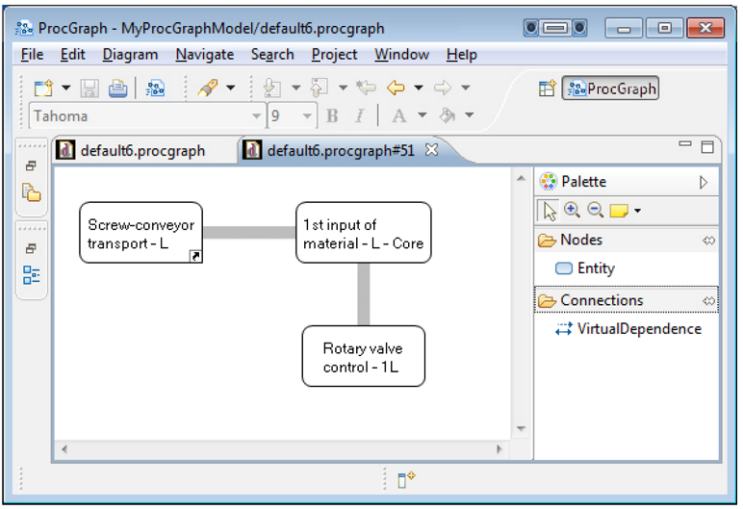
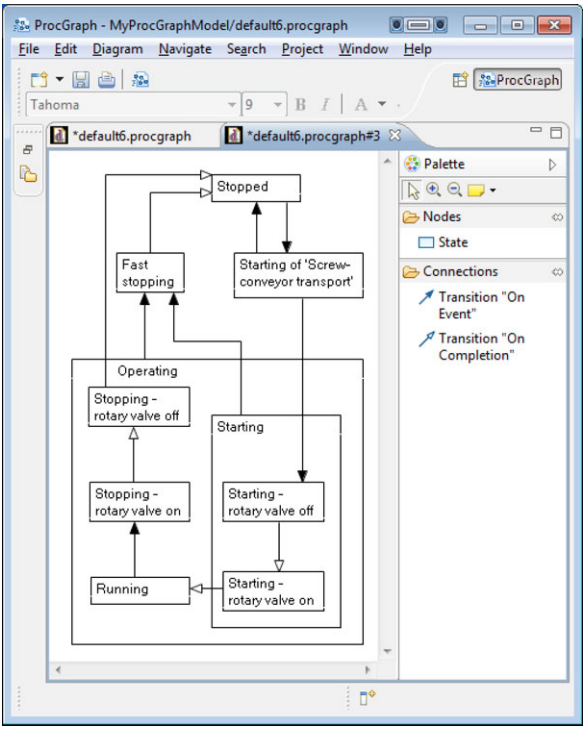


Fig. 12.13 The initial subED of the '1st input of material–L' activity

Fig. 12.14 The STD of '1st input of material–L–Core'



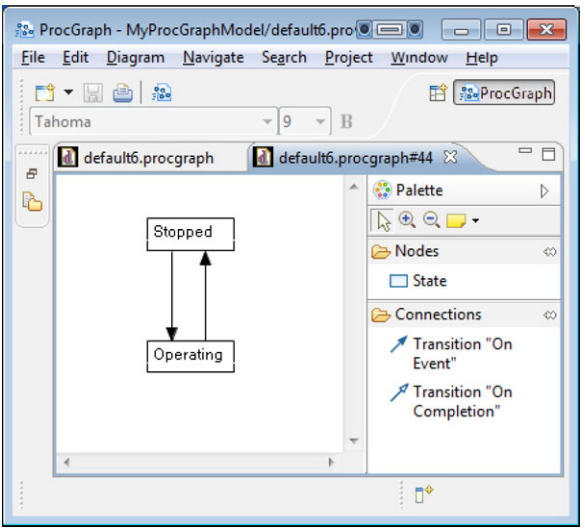


Fig. 12.15 The initial STD of 'Rotary valve control–1L'

Next is the modelling of the interdependencies between the '1st input of material–L–Core' sub-activity and the 'Screw-conveyor transport–L' activity, which is accomplished with the SDD in Fig. 12.17. The '1st input of material–L–Core' can only be started when 'Screw-conveyor transport–L' is not changing the state from 'Stopped' to 'Running' or vice versa (due to a previous propagation from '1st input of material–L–Core', since this is the only possible cause of such transition). During the starting of the '1st input of material–L–Core' activity, the 'Screw-conveyor transport–L' activity also has to be started by means of two-way propagation dependencies. When all material input activities ('1st input of material–L–Core', '2nd input of material–L–Core', and '3rd input of material–L–Core') become stopped, this propagates into the stopping of the 'Screw-conveyor transport–L' activity. The 'Fast stopping' state of the 'Screw-conveyor transport–L' activity causes (i.e., it is propagated into) the fast stopping of the '1st input of material–L–Core' activity.

A consequence of defining the interdependencies with the SDDs in Fig. 12.16 and Fig. 12.17 are changes to the EDs containing the composite dependencies that explode into these SDDs. Concretely, the initial subED from Fig. 12.13 changes into the final subED in Fig. 12.18, which now has two defined composite dependencies. These composite dependencies now have an appearance that shows the union of the elementary dependencies that are defined in their respective SDDs. The composite dependencies that span over various hierarchies of EDs change their appearance on all hierarchy levels. Concretely, the composite dependency between 'Screw-conveyor transport–L' and '1st input of material–L–Core' also changes its appearance in the ED in which the 'Screw-conveyor transport–L' is defined. After the two other input material sub-activities ('2nd input of material–L' and '3rd input of material–L') are defined in the same way as '1st input of material–L', the initial ED in Fig. 12.11 changes to the final one in Fig. 12.19.

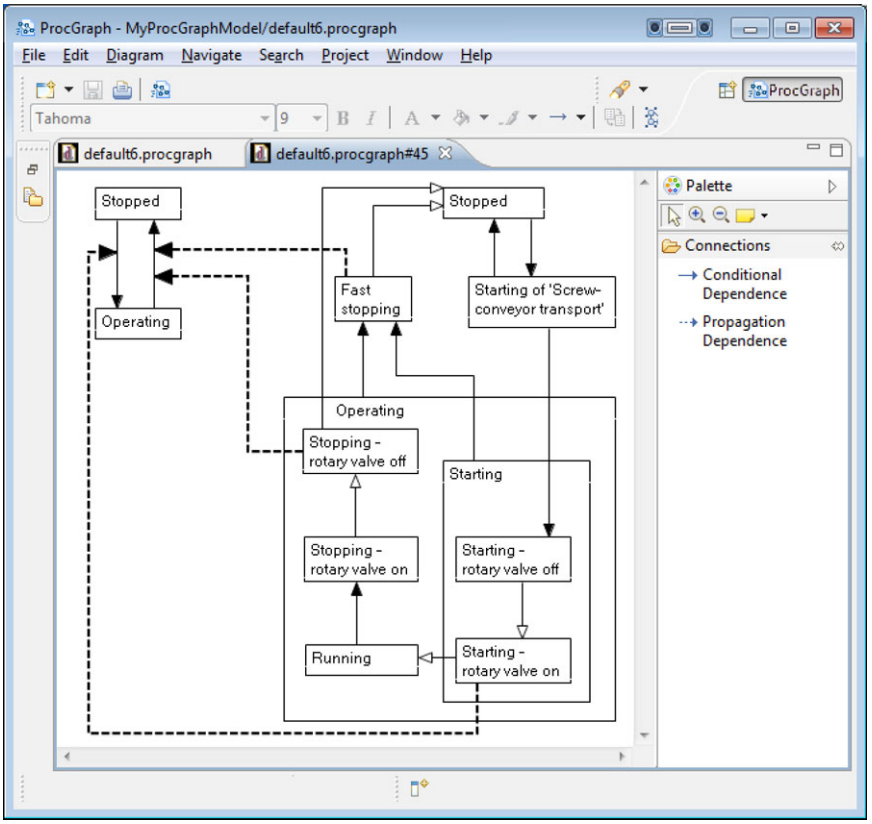


Fig. 12.16 The SDD showing the dependencies between the sub-activities 'Rotary valve control–1L' and '1st input of material–L–Core'

The next operation to be defined is the 'Pneumatic transport' operation. This operation is not decomposed into activities; therefore it is already possible to define its behaviour, which is done with the STD in Fig. 12.20. After introducing the typical states and transitions, the requirements are studied to identify additional states. Two different running states are added, whereby cases in which a too high pressure failure is detected (caused by an obstruction) are handled within the 'High pressure' state, which addresses the failure by closing the rotary valve L.

The starting and stopping sequences are each split into two elementary states which check different alarm conditions. Because two of the created states ('Starting–full alarms' and 'Stopping–alarms') and the state 'Running' have a common set of alarm conditions, these three states are enclosed in the 'Operating–alarms' super-state.

Each starting state has specific auto-stopping causes, therefore they both have their own additional transition to 'Fast stopping'.

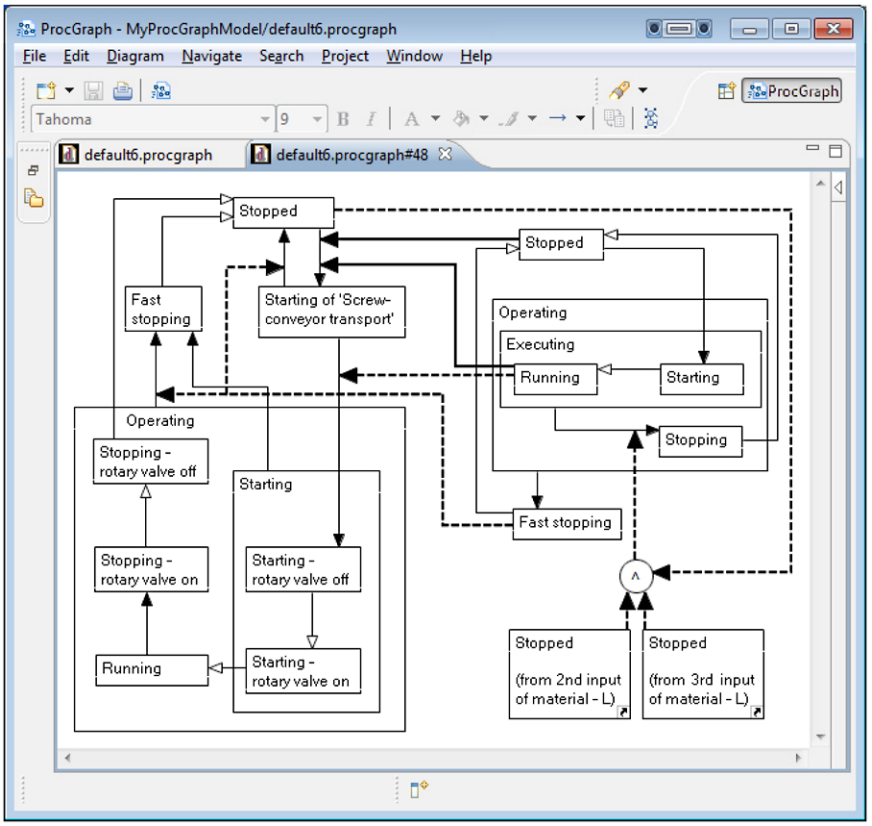


Fig. 12.17 The SDD of the composite dependency between the sub-activities '1st input of material–L–Core' and 'Screw-conveyor transport–L'

The behaviour of the 'Grinding' operation is specified with its final STD in Fig. 12.21. After inserting the standard states and the transitions between them, the specific requirements of this operation are studied and we discover that this operation needs two different running modes. Therefore, 'Normal operation' and 'Minimal load operation' sub-states are introduced. The latter is used to handle a situation in which an obstruction in the mill or in the pneumatic transport occurs, which requires the temporary stopping of the mill-feeding screw-conveyor. Another requirement is that this operation should not be turned on after a shutdown for the time interval given by a parameter (around 2 minutes). The easiest way to ensure this is to introduce two 'Stopped' sub-states: 'Stopped–start disabled', which becomes active first and 'Stopped–start enabled', which becomes active automatically after the required delay has run out.

Next, it becomes clear that some of the 'Operating' sub-states have common sets of additional failures; therefore they are enclosed in the super-state 'Additional AS'. Notice that the 'Starting' state is split into the 'Starting 1st part' and 'Starting 2nd

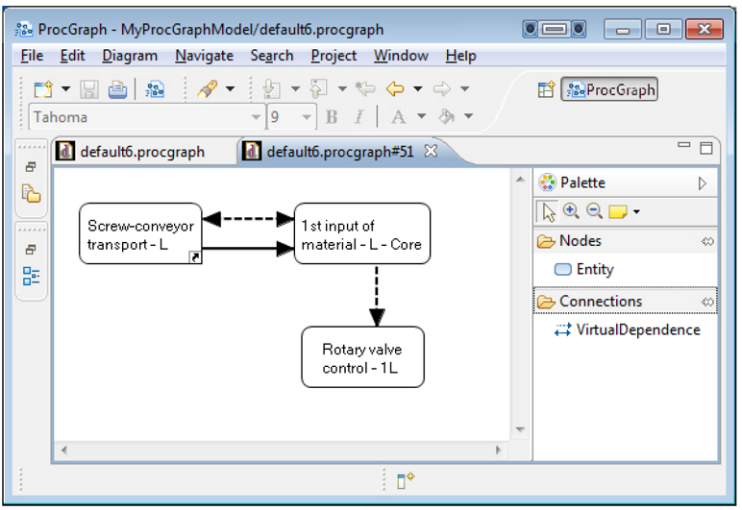
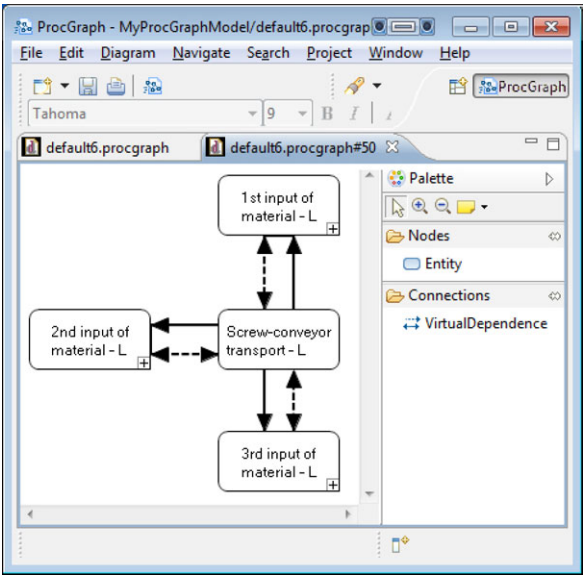


Fig. 12.18 The final subED of the '1st input of material–L' activity

Fig. 12.19 The final subED of the 'Input material transport–Left' activity



part', because the latter has additional auto-stopping causes. The 'Normal operation' state has even more additional auto-stopping causes, therefore it gets its own transition to the 'Fast stopping' state.

The 'Stopping' state is split into two separate states that should be monitored by different alarms. After looking at the commonalty of the alarms, two super-states are introduced: 'All states', which means that there is a subset of alarms that have

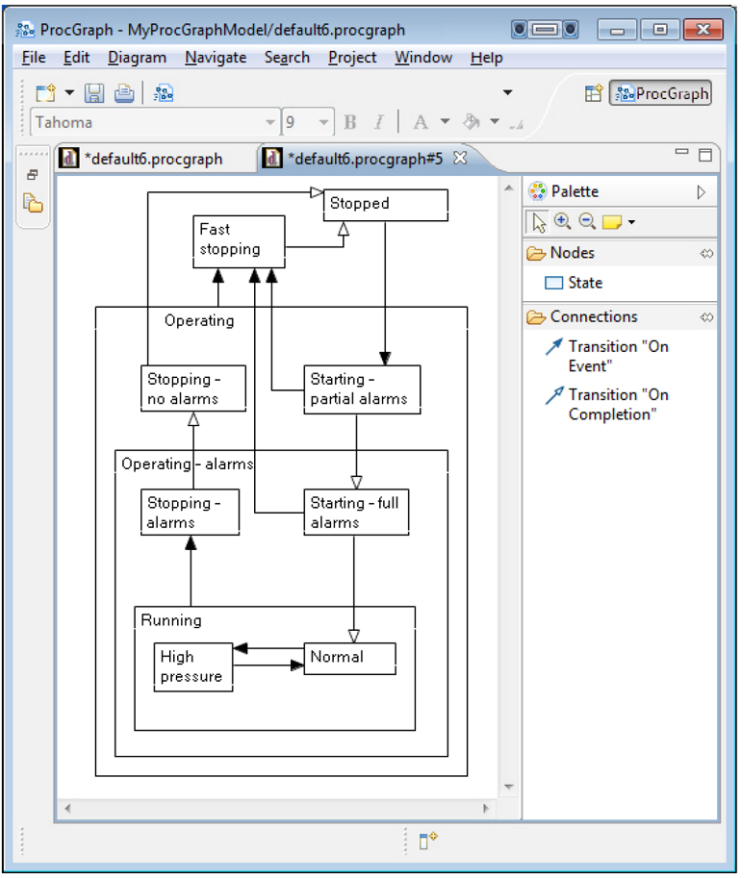


Fig. 12.20 The STD of the 'Pneumatic transport' operation

to be monitored in all states of the 'Grinding' operation, and 'Operating–alarms', which covers additional specific common alarms.

The operation 'End product transport' is decomposed into two activities, which are shown in the initial subED in Fig. 12.22. Because the 'End product transport–Core' main activity controls the 'DHC control' auxiliary activity, they are connected with a composite dependency. However, currently it is not yet clear which states of the main activity will influence which transitions of the auxiliary activity, because the behaviour of the activities is not yet defined. Therefore, the composite dependency has an "undefined shape" (the thick grey connecting line in Fig. 12.22).

The final STD of 'End product transport–Core' is depicted in Fig. 12.23. After adding the typical states and transitions, an investigation of the specific auto-stopping causes reveals the need for a transition from 'Starting' to 'Fast stopping'. 'Starting–DHC off' and 'Starting–DHC on' states and two mirror stopping states are created for the purpose of issuing commands (through propagation dependencies)

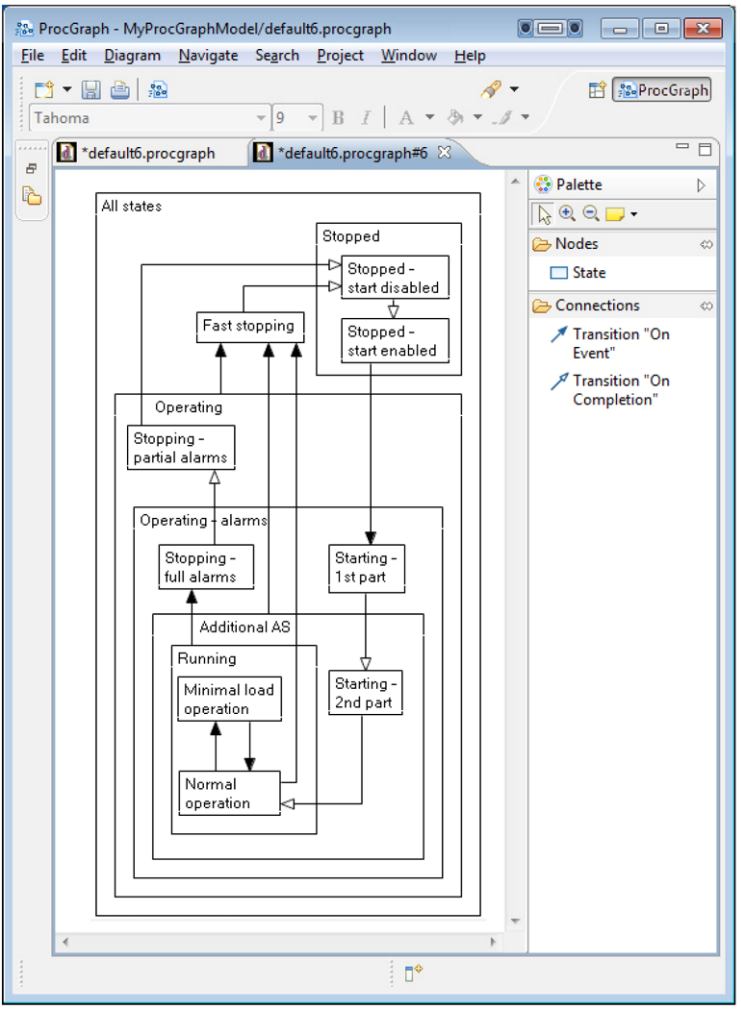


Fig. 12.21 The final STD of the 'Grinding' operation

to the 'DHC control' auxiliary activity. After looking if states have specific alarms, the 'Stopping–DHC on–alarms' state and the 'Operating–alarms' super-state are created.

The STD of the 'DHC control' activity shown in Fig. 12.24 is simple, because it has only two states.

The next development step is the modelling of the interdependent behaviour (see Sect. 12.4.2.3). The SDD in Fig. 12.25 defines that the 'End product transport–Core' main activity turns the 'DHC control' on and off through three propagation dependencies. Two propagation dependencies ensure that the 'DHC control' is turned off when 'End product transport–Core' is being stopped.

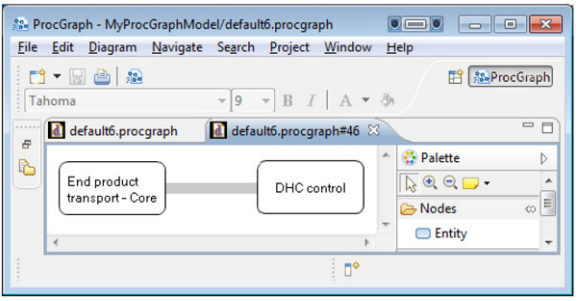


Fig. 12.22 The initial subED of the 'End product transport' operation

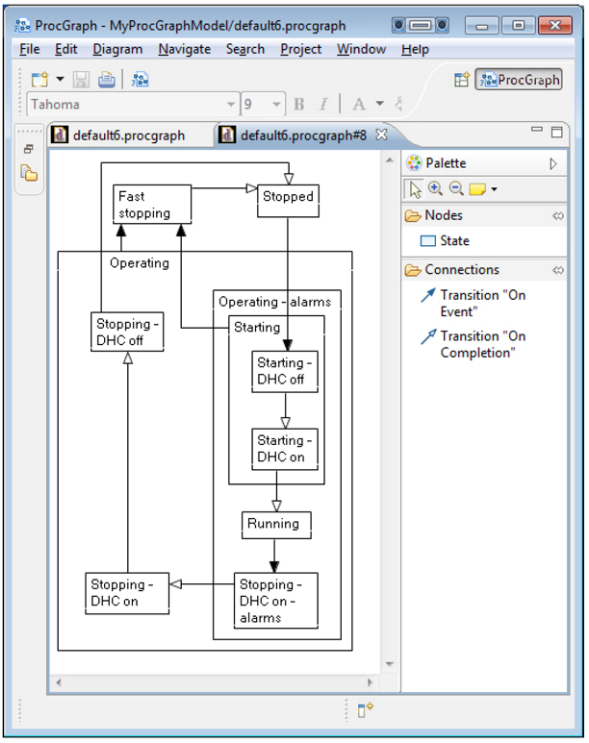


Fig. 12.23 The final STD of the 'End product transport–Core' activity

Because the emergent model and the requirements reveal that 'Grinding' and 'End product transport–Core' are interdependent, we add a composite dependency between them and define the SDD in Fig. 12.26. We can see that the 'End product transport–Core' has to be running before 'Grinding' can begin its transition to its first starting state. On the other hand, the 'End product transport–Core' can only go into its first stopping state when 'Grinding' is stopped, except when 'End product transport–Core' goes into fast stopping, which must then be followed by fast stopping of the 'Grinding' operation. In other words, this interdependency simply means

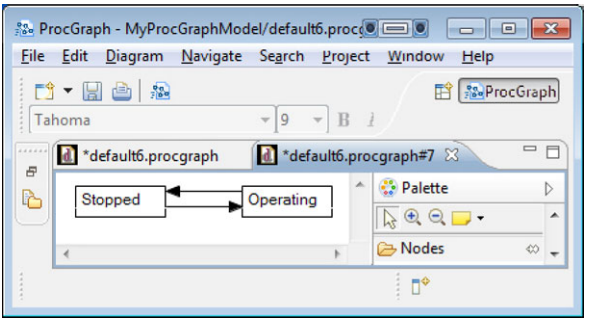


Fig. 12.24 The STD of the 'DHC control' activity

that 'Grinding' is only allowed to operate if 'End product transport–Core' is in the 'Running' state.

After all the SDDs are defined, the composite dependencies in the initial root ED (Fig. 12.9) and the initial subED of 'End product transport' (Fig. 12.22) change

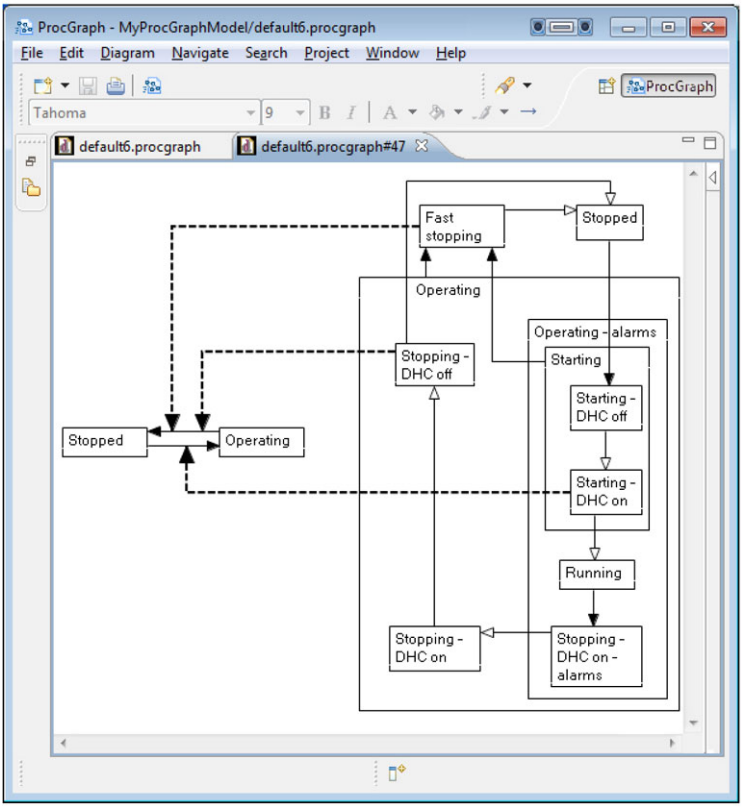


Fig. 12.25 The SDD showing the dependencies between the 'DHC control' and 'End product transport–Core' activities

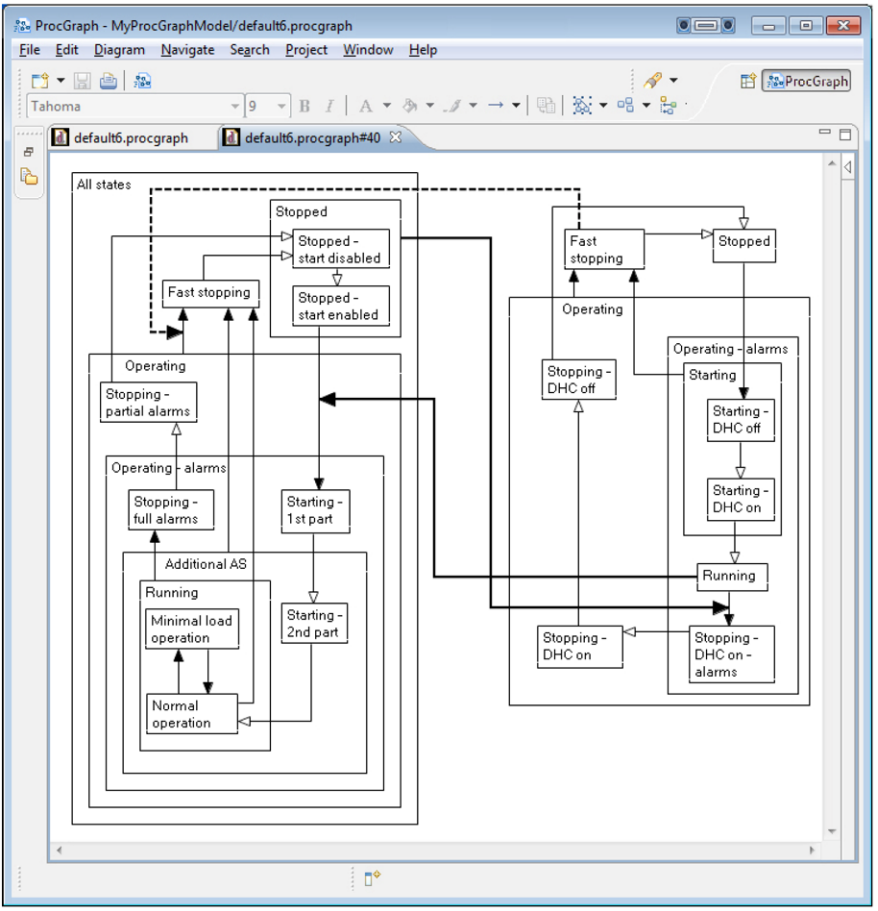


Fig. 12.26 The SDD showing the dependencies between the 'Grinding' and 'End product transport–Core' activities

their appearance to show the union of the elementary dependencies which are defined in the SDDs. We can see that 'Grinding' is now present in the subED (in Fig. 12.27) as a PCE proxy that references the original PCE, which is defined in the root ED (in Fig. 12.9). Consequently, the composite dependency between 'End product transport–Core' and 'Grinding', which actually spans across two hierarchy levels, also becomes visible in the root ED (in Fig. 12.28).

After the whole model is defined and validated to ensure its completeness and conformity to the constraints, the code can be automatically generated (according to the rules mentioned in Sect. 12.4.1.3). For this example, we selected the generator for Mitsubishi PLCs. Because there are no demands for critical response times in this example, it is appropriate that the generated software be executed only on one PLC and in only one task. Figure 12.29 contains a screenshot of the Mitsubishi IDE

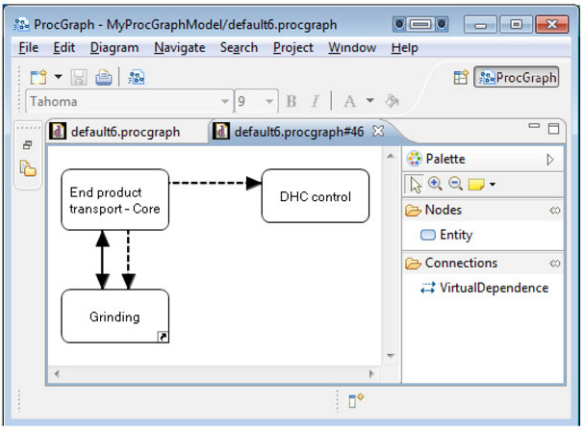


Fig. 12.27 The final subED of the 'End product transport' ED

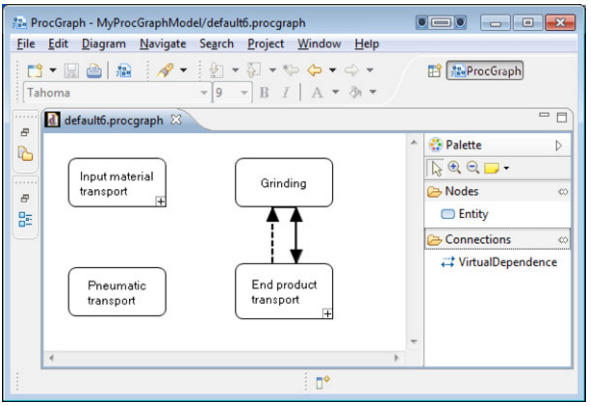


Fig. 12.28 The final root ED of the example process

showing the top-level code that is generated from the STD in Fig. 12.23. The mentioned STD is transformed into an FBD according to the transformation rules of the approach presented. In the FBD we see three networks, each with one FB instance. This corresponds to the number of top-level states of the source STD. It is visible that the 'EPTC_Stopped' and 'EPTC_FastStopping' FBs are active when the respective state from which they are transformed is the active state of the EPTC FBD (which is a transformation of the 'End product transport–Core' activity). The 'EPTC_Operating' FB instance is active when any of the elementary sub-states of the super-state from which it is transformed is the active state.

12.5.3 Evaluation of Results

The aim of the presented example was to illustrate the software development process based on the high level, domain-specific, process-oriented modelling language

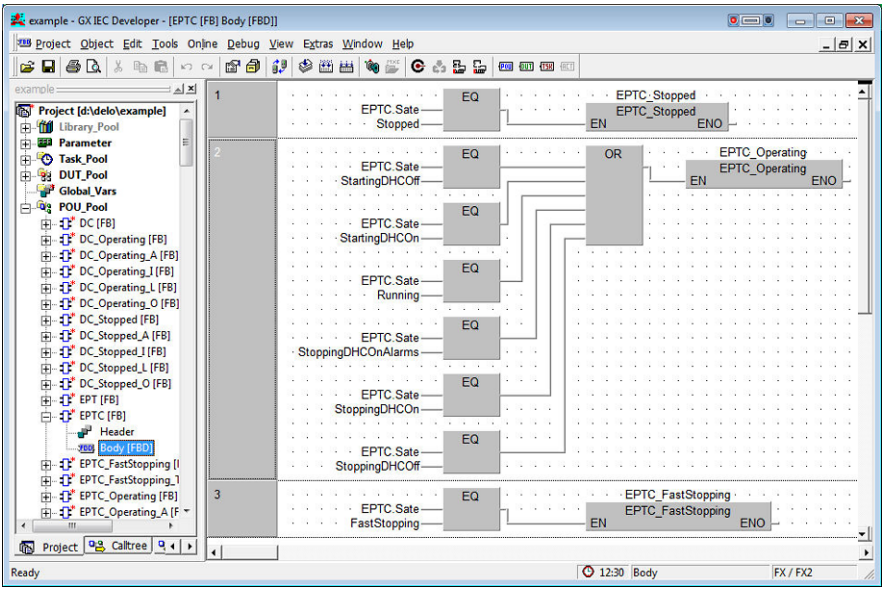


Fig. 12.29 A screenshot of the generated project showing an open EPTC FBD that is the transformation of the STD from Fig. 12.23

ProcGraph and the supporting IDE. The technological process under consideration is a mid-size and mid-complexity continuous process, and thus not very simple, however, we have shown that it can be mastered in a very transparent and easy way using elegant graphical specifications. The advantages of the approach could be better expressed if direct comparison with traditional PLC software development of the same process was possible where the productivity of the development and quality of the obtained SW would be compared in quantitative terms. This is unfortunately not the case. What we can do is to compare the different development paradigms of the traditional approach and the proposed new approach. This can be accomplished on the basis of our own experience in industrial process software development and experience gained during many years of cooperation with process software development SMEs.

Let us start with the traditional approach, which usually has the following phases: analysis-specification, coding, off-site testing, on-site testing and delivery. Typically the analysis-specification phase delivers an informal specification document, written in plain text, which is very difficult to understand, verify and transform correctly into program code. As a result, there is a large number of errors in the program code originating from both the analysis/specification and the coding phases, which are discovered only during the on-site testing phase, which is doubtless the most expensive method of producing correct software. Hence, this type of development paradigm could be characterised as a "*construct by correction*" process.

As the opposite of the process described above, the process based on the ProcGraph modelling language results in a specification that is formal and at the same

time transparent, understandable and elegant. This specification offers the required expressive power allowing efficient analysis and communication between the analyst and the process engineers. Furthermore, the specification can be verified (formally or informally, by reviews) and achieve a high degree of correctness. Finally, it can be routinely transformed into the program code, allowing the code to be verified (again formally or informally) and to reduce (according to our experience in some projects) between 90 % and 100 % of errors introduced by coding. Hence, this type of development paradigm could be characterised as a "*correct by construction*" process. The automatic transformation of the ProcGraph models into code which has been developed so far confirms that it is realistic to expect the elimination of the routine and error-prone manual transformation activity and consequently a reduction in the number of coding errors to zero.

12.6 Issues Related to Using the Approach in Real Projects

Based on the reports of the end-users and our previous experience in developing process control software, the presented approach provides a number of benefits, including improved software quality, improved productivity, improved communication and interaction between the development participants, platform independence, etc.

However, using the approach in practice has revealed some drawbacks and limitations which can also be seen as points where it can be improved. Let us mention some of them.

While the use of DSMLs results in the reuse of knowledge, it does not by itself help to raise the degree of the *reusability of components and the non-repetition* of information, which have significant potential to improve the presented approach. This improvement could be achieved by the introduction of object-oriented procedural control and libraries of reusable components. These libraries could be introduced in the area of procedural control (a library of reusable procedural control entities) as well as in the area of basic control (a library of reusable equipment entities). In the area of basic control, this would be rather straightforward (similar libraries already exist in commercial tools, e.g., PCS7 [27]), while in the area of procedural control this would be much more demanding, due to the great diversity of procedural control entities among different projects. However, reusing procedural control entities has a much greater economic potential than reusing basic control entities due to the much higher complexity of procedural control compared to basic control.

The presented approach in its current state does not include any automatic verification tool for various properties of the modelled software (e.g., safety properties, liveliness, or the absence of deadlocks). The development of such a tool could bring with it significant further improvement of the *quality* of the generated code, therefore this will be one of our main future research and development directions.

Since ProcGraph is a high-level, domain-specific and process-oriented modelling language, its elements have rather high *expressive power*. However, there are various points in the language definition where the expressive power could be raised.

Based on our experience, we identified some new language elements which would be useful, such as history states or additional real-time language constructs, e.g., the specification of deadlines. Furthermore, an important improvement would be to provide automatic support for developers in determining appropriate procedural control entities, based on requirements documentation, for example P&ID diagrams.

In the current version of the presented approach, the software-to-hardware mapping is performed manually, without any *distributed systems support*. Such support could be carried out by adopting an existing formalism or developing a new formalism which would enable the modelling of the software-to-hardware mapping.

Perhaps the most important challenge of using the presented approach in real projects lies in its *understanding and acceptance by developers*. As already mentioned, process control software developers are mainly electrical engineers, predominantly accustomed to using low-level constructs and languages, and often reluctant to use more abstract high-level constructs, models and modelling languages. Therefore, in our opinion, based on our experience with real projects and real people, it is of key importance to ensure that new developers receive proper training and mentoring, possibly at the very beginning of their professional careers. That would prevent them from acquiring improper development patterns, which are ever more difficult to eliminate once the acquisition of proper practices is delayed.

12.7 Conclusion

A new approach to process control software engineering was presented. It is based on the innovative domain-specific, process-oriented modelling language ProcGraph and an integrated development environment which includes automatic code generation for the PLC platform. The approach is aligned with the issues of the process control domain and aims to overcome several weaknesses of the state-of-the-art approaches.

The main benefits of the approach are improved software quality, improved productivity, improved communication and interaction between the members of the development team, and platform independence. The presented approach also has some drawbacks and open issues. These include the following: a lack of tool support for the reusability of the procedural control entities, no automatic verification is provided in the current version and the rather demanding determination of the right set of procedural control entities, which opens a new research direction with regard to the automatic generation of ProcGraph models based on users requirements documentation.

Acknowledgements The financial support of the Slovenian Research Agency (P2-0001) is gratefully acknowledged.

References

1. ANSI/ISA-88.00.01-1995 (1995) Batch control, Part 1: Models and terminology. ISA, Research Triangle Park

2. ANSI/ISA (2009) ANSI/ISA-5.1-2009: instrumentation symbols and identification. ISA, Research Triangle Park
3. Avila-Garcıa O, Garcıa AE (2008) Providing MOF-based domain-specific languages with UML notation. In: Proceedings of the 4th workshop on the development of model-driven software, San Sebastian, Spain, pp 11–20
4. Bitsch F, Gohner P, Gutbrodt F, Katzke U, Vogel-Heuser B (2005) Specification of hard real-time industrial automation systems with UML-PA. In: Proceedings of IEEE international conference on industrial informatics (INDIN'05), Perth, Australia, pp 339–344
5. Budinsky F, Steinberg D, Merks E, Ellersick R, Grose TJ (2003) Eclipse modeling framework. Addison-Wesley Professional, Reading
6. Chia-han Y, Vyatkin V (2010) Model transformation between MATLAB Simulink and function blocks. In: Proceedings of IEEE international conference on industrial informatics (INDIN'10). Osaka, Japan, pp 1130–1135
7. Colla M, Leidi T, Semo M (2009) Design and implementation of industrial automation control systems: a survey. In: Proceedings of IEEE international conference on industrial informatics (INDIN'09), Cardiff, UK, pp 570–575
8. Estevez E, Marcos M, Sarachaga I, Orive DA (2007) Methodology for multidisciplinary modeling of industrial control systems using UML. In: Proceedings of IEEE international conference on industrial informatics (INDIN'07), Vienna, Austria, pp 171–176
9. Estévez E, Marcos M, Orive D (2007) Automatic generation of PLC automation projects from component-based models. Int J Adv Manuf Technol 35(5):527–540
10. Friedrich D, Vogel-Heuser B (2007) Benefit of system modeling in automation and control education. In: Proceedings of American control conference (ACC'07), New York, NY, USA, pp 2497–2502
11. Godena G (1997) Conceptual model for process control software specification. Microprocess Microsyst 20(10):617–630
12. Godena G (2004) ProcGraph: a procedure-oriented graphical notation for process-control software specification. Control Eng Pract 12(1):99–111
13. Haase A, Volter M, Efftinge S, Kolb B (2007) Introduction to openArchitectureWare 4.1.2. In: Proceedings of model-driven development tool implementers forum (MDD-TIF'07) co-located with TOOLS 2007, Zurich, Switzerland
14. Hästbacka D, Vepsäläinen T, Kuikka S (2011) Model-driven development of industrial process control applications. J Syst Softw 84(7):1100–1113
15. Heck BS, Wills LM, Vachtsevanos GJ (2009) Software technology for implementing reusable, distributed control systems. IEEE Control Syst Mag 23(1):267–293
16. Hovsepyan A, Van Baelen S, Vanhooff B, Joosen W, Berbers Y (2006) Key research challenges for successfully applying MDD within real-time embedded software development. In: Proceedings of 6th international workshop on systems, architectures, modeling, and simulation (SAMOS'06), Samos, Greece, pp 49–58
17. John K-H, Tiegelkamp M (2001) IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids. Springer, Berlin
18. Lewis RW (1998) Programming industrial control systems using IEC 1131-3. The Institution of Engineering and Technology, London
19. Maurmaier M (2008) Leveraging model-driven development for automation systems development. In: Proceedings of IEEE international conference on emerging technologies and factory automation (ETFA'08), Hamburg, Germany, pp 733–737
20. Mitsubishi electric, MELSOFT-software—GX IEC developer. http://www.mitsubishi-automation.com/products/software_gx_iec_developer.htm
21. Noyrit F, Gérard S, Terrier F, Selic B (2010) Consistent modeling using multiple UML profiles. In: Proceedings of the 13th international conference on model driven engineering languages and systems (MODELS'10), Oslo, Norway, pp 392–406
22. Panjaitan S, Frey G (2007) Combination of UML modeling and the IEC 61499 function block concept for the development of distributed automation systems. In: Proceedings of the IEEE

international conference on emerging technologies and factory automation (ETFA'06), Prague, Czech Republic, pp 766–773
23. Peltola JP, Sierla SA, Stromman MP, Koskinen KO (2006) Process control with IEC 61499: designers' choices at different levels of the application hierarchy. In: Proceedings of IEEE international conference on industrial informatics (INDIN'06), Daejeon, Korea, pp 183–188
24. Rodriguez-Priego E, García-Izquierdo F, Rubio Á (2010) Modeling issues: a survival guide for a non-expert modeler. In: Proceedings of 13th international conference on model driven engineering languages and systems (MODELS'10), Oslo, Norway, pp 361–375
25. Schmidt DC (2006) Model-driven engineering. IEEE Comput 39(2):25–31
26. Shlaer S, Mellor SJ (1992) Object lifecycles: modeling the world in states. Yourdon Press, Englewood Cliffs
27. Siemens (1999) SIMATIC PCS7, technological blocks manual
28. Sprinkle J, Mernik M, Tolvanen J-P, Spinellis D (2009) What kinds of nails need a domain-specific hammer? IEEE Softw 26(4):15–18
29. Strasser T, Rooker M, Ebenhofer G, Hegny I, Wenger M, Sunder C, Martel A, Valentini A (2008) Multi-domain model-driven design of industrial automation and control systems. In: Proceedings of IEEE international conference on emerging technologies and factory automation (ETFA'08), Hamburg, Germany, pp 1067–1071
30. Streitferdt D, Wendt G, Nenninger P, Nyßen A, Horst L (2008) Model driven development challenges in the automation domain. In: Proceedings of the IEEE international computer software and applications conference (COMPSAC'08), Turku, Finland, pp 1372–1375
31. The Object Management Group (2009) UML superstructure specification version 2.2
32. Thramboulidis K (2005) IEC 61499 in factory automation. In: Proceedings of IEEE international conference on industrial electronics, technology and automation (IETA'05), Bridgeport, CT, USA, pp 115–124
33. Thramboulidis K, Tranoris C (2004) Developing a CASE tool for distributed control applications. Int J Adv Manuf Technol 24(1):24–31
34. Thramboulidis K, Perdikis D, Kantas S (2007) Model driven development of distributed control applications. Int J Adv Manuf Technol 33(3):233–242
35. Tranoris C, Thramboulidis K (2006) A tool supported engineering process for developing control applications. Comput Ind 57(5):462–472
36. Vogel-Heuser B, Wannegat A (2009) Modulares Engineering und Wiederverwendung mit CoDeSys V3. Oldenbourg Industrieverlag, München
37. Vogel-Heuser B, Witsch D, Katzke U (2005) Automatic code generation from a UML model to IEC 61131-3 and system configuration tools. In: Proceedings of the IEEE international conference on control and automation (ICCA'05), Budapest, Hungary, pp 1034–1039
38. Ward PT, Mellor SJ (1986) Structured development for real-time systems, vol II: Essential modeling techniques. Prentice Hall, Englewood Cliffs
39. Weisemöller I, Schürr A (2008) A comparison of standard compliant ways to define domain specific languages. In: Giese H (ed) Proceedings of the 11th international conference on model driven engineering languages and systems (MODELS'08), Nashville, TN, USA, pp 47–58