Journal of Software Engineering and Applications, 2023, 16, 315-347
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

DSML ProcGraph: Overview and a Mid-Size Industrial Application Example

Giovanni Godena, Miha Glavan

Jožef Stefan Institute, Ljubljana, Slovenia
Email: giovanni.godena@ijs.si, miha.glavan@ijs.si

How to cite this paper: Godena, G. and Glavan, M. (2023) DSML ProcGraph: Overview and a Mid-Size Industrial Application Example. *Journal of Software Engineering and Applications*, **16**, 315-347.
https://doi.org/10.4236/jsea.2023.168017

Received: June 12, 2023
Accepted: August 21, 2023
Published: August 24, 2023

Abstract

This paper presents model-based approach to process-control software development. The presented approach enables modelling of control software in a straightforward manner and, at the same time, on a high level of abstraction. The essence of the presented approach is a high-level, domain-specific modelling language ProcGraph, which is based on three types of diagrams that describe the modelled system using a domain-oriented hierarchical structure of interdependent procedural control entities and state-transition diagrams describing the behaviour of the procedural control entities. The presented concept is demonstrated by means of higher-level model segments of a real process-control application that deals with the micronisation process in the production of titanium dioxide. The presented industrial case shows that the application of ProcGraph provides adequate expressive power for an elegant preparation of graphic specifications in a transparent and easy way.

Keywords

Model-Driven Software Engineering, Domain-Specific Modelling Languages, Process-Control Software, State Machines, Titanium Dioxide

1. Introduction

Industrial process control systems are aimed at controlling and supervising the behaviour of technological processes in order to achieve their specific goals. Software is the central and most complex part of process control systems because it implements various demanding real-time control functions and procedures [1]. In our experience, the complexity of the development, operation and maintenance of process-control software is not so much associated with classic control functions, *i.e.* achieving and maintaining the desired state of process variables (hereinafter referred to as basic control), but more with the software im-

plementing the discrete behavioural control [2], which performs a sequence of activities that ensure that the goals of the system or process are achieved [3] (hereinafter referred to as procedural control). In addition to greater complexity, procedural control is also characterized by a significantly higher degree of variability between the individual applications than is the case in basic control.

For all above mentioned reasons, it is very important how this procedural control software is developed and what its quality attributes are. Unfortunately, software engineering state-of-the-practice in the domain of industrial process control mainly has a low maturity and is failing to address its demands, primarily those regarding software quality [4] [5] [6]. The causes for this are in the following deficiencies of the software development process:

- a focus on coding and testing phases with little activity in the earlier software lifecycle phases,
- reliance on the individual programming skills instead of advanced software engineering concepts,
- the use of inadequate device-centric, instead of process-oriented or goal-oriented abstractions,
- a low degree of reuse of artefacts and knowledge, in particular those belonging to early lifecycle phases,
- a time-consuming and error-prone development process due to the low expressive power of the programming languages used in the process-control domain, and,
- a low degree of development process automation.

In order to successfully overcome the issues mentioned above, better concepts and methodologies for the engineering of process control software are needed. One of these concepts is Model Driven Engineering (MDE) paradigm, which raises the abstraction level of software/system development from low-level artefacts to a higher-level of models as central artefacts in the software engineering process [7]. MDE bridges the gap between problem identification and software implementation phases [8] [9] by combining domain-specific modelling abstractions with transformation engines and generators that allow generating of various artefacts [7] [10]. Another concept is Domain-Specific Modelling (DSM) and Domain-Specific Modelling Languages (DSML), which enables the modelling of software and systems using highly expressive and reusable domain specific abstractions [11] and ontologies that provide conceptual models and the expressivity to capture requirements sufficiently [12]. Domain-specific modelling and Domain-specific modelling languages have recently been used in an increasing number of different domains, for example the domain of smart buildings [13] or railway control systems [14].

The paper is structured in the following manner. Section 2 gives an overview of the domain specific modelling language ProcGraph. In Section 3, an example application implementing the control system for the micronisation of the coarse granulated titanium dioxide is presented. Finally, Section 4 draws the conclu-

sions of the paper.

2. Domain-Specific Modelling Language ProcGraph

2.1. Language Requirements

As already mentioned above, in procedural control there is much more complexity and variability between the individual applications than is the case in basic control. Consequently, the central part of a ProcGraph model should be information about the procedural control, and not the information about the underlying basic control.

When defining the syntax and semantics of the domain specific modelling language ProcGraph, the following main requirements were emphasized:

1) The language must include elements that are suitable for describing the dynamic behaviour of reactive systems, since process-control systems are reactive systems in their very nature.

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, both in the extent of coupling and in the number of possible different types of coupling; furthermore, coupling must be explicitly visible at a high level in the model.

Given the above-listed main requirements, the following lower level language requirements were determined. The modelling language has to be closely related to the domain of process control, especially its procedural control entities, which 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 kind of an extended finite-state machine. Another important segment of the modelling language 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. As mentioned above, the modelling language should as much as possible limit the number of possible types of coupling, *i.e.* the number of different types of dependence relations between the procedural control entities. These dependency relations must also be part of the graphical notation, and it should appear explicitly and at a very high level in the model. This should be the best way to maintain good control of the number of these dependencies and, consequently, also to minimise the coupling. It should be noted, that there is a certain similarity between the concept of procedural control entities described above and agents in multi-agent systems [15], where in our opinion procedural control entities are somewhat more domain-oriented, while agents in multi-agent systems are based more on the method of solution and decomposition of

the problem than on the problem domain itself.

2.2. Language Elements and Diagram Types

Based on the requirements stated in the previous section, the language elements were defined. The example excerpts of a ProcGraph model shown in Figure 1 will be used as an aid to explain the elements of the language.

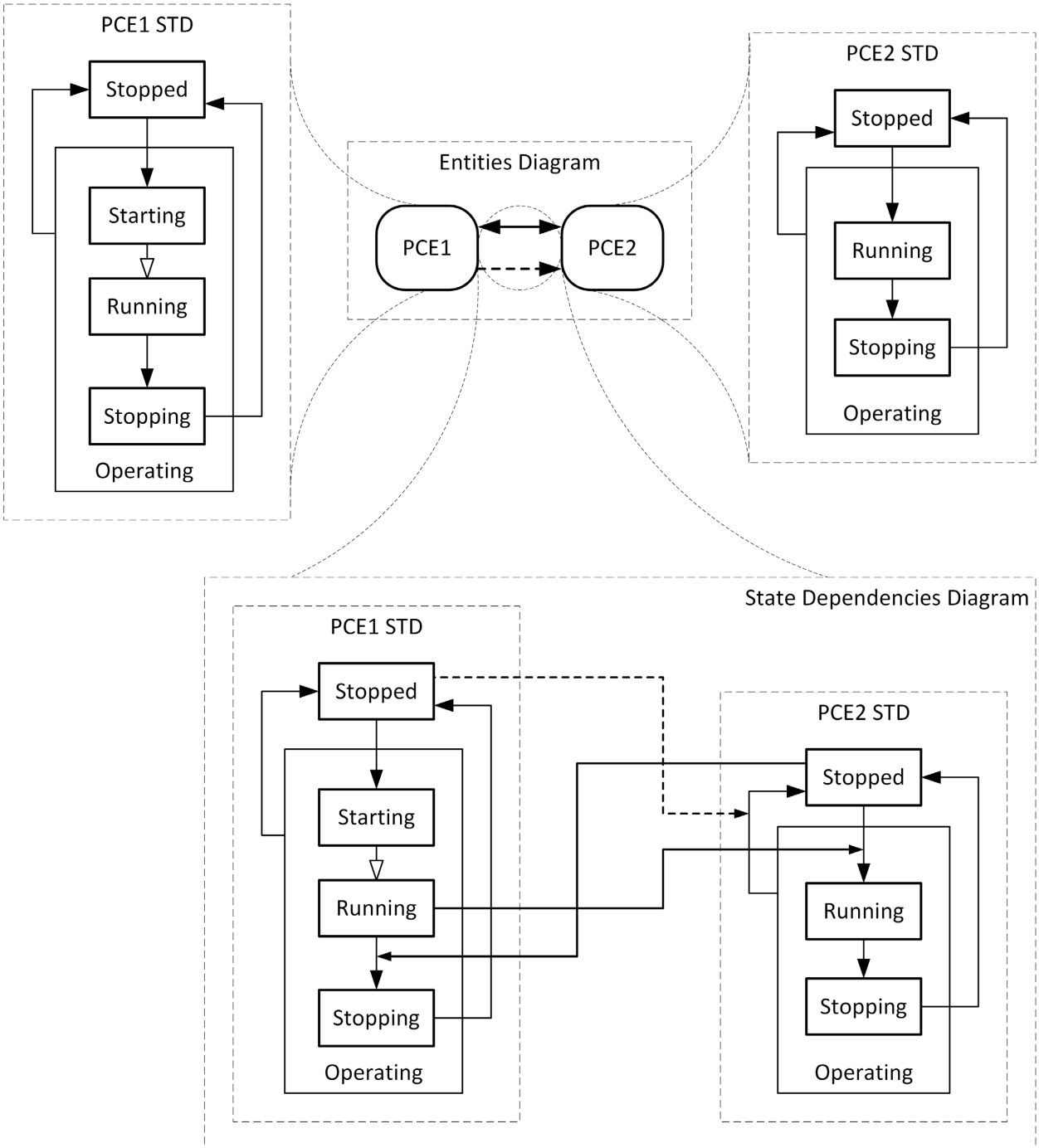

Figure 1. Schematic structure of a ProcGraph model shown on an example of an entity diagram containing two elementary entities (PCE1, PCE2), their state transition diagrams (STD), and their state dependencies diagram.

The main elements of the ProcGraph language are its central processing entities and its three different diagram types. The central processing entity in a ProcGraph model is the procedural control entity (PCE). A PCE can be either elementary or a super PCE, which is composed of lower-level PCEs. The highest-level PCEs are the operations, which may be decomposed to various levels of their sub-activities.

The three diagram types are entities diagram, state transition diagram, and state dependencies diagram. These diagram types are described in the following subsections.

2.2.1. Entities Diagram

The first diagram type is the procedural control entities diagram (hereinafter referred to as entities diagram, ED). The entities diagram shows the existence of procedural control entities of a certain hierarchy level (elementary PCE or super PCE) and the composite relationships (union of all relationships) between them. An elementary PCEs is denoted by a rounded rectangle drawn with a thin line, and a super PCE is denoted by a rounded rectangle drawn with a thick line.

There are only two possible types of relationships between PCEs: the transition condition relationship and the transition propagation relationship. The first is denoted by a solid arrowed line connecting two PCEs and the second by a dashed arrowed line. The example entities diagram in **Figure 1** shows that there are two PCEs, PCE1 and PCE2, and that they both are elementary entities. It also shows that there is one or more propagational dependencies from PCE1 to PCE2, meaning that the transitions to certain states of PCE1 cause (are propagated to) certain state transitions of PCE2. Finally, the example entities diagram shows that there is one or more conditional dependencies in both directions, from PCE1 to PCE2 (meaning that some state transitions of PCE2 are possible only if PCE1 is in a certain state), and from PCE2 to PCE1 (meaning that some state transitions of PCE1 are possible only if PCE2 is in a certain state).

2.2.2. State Transition Diagram

The second type of diagram is the state-transition diagram (STD), which defines the behaviour of a particular elementary PCE. ProcGraph state transition diagrams are particular in a number of ways, since they are process-control domain-specific [16].

2.2.3. State

Syntactically, a state is a graph node denoted by a rectangle with the state name written inside. Examples of states are Stopped, Running, and Stopping states of the procedural control entity PCE2 in **Figure 1**.

A state transition diagram can contain different types of states, which are divided according to two criteria. According to the criterion of the processing, the states are divided in the following way:

- Quiescent states are states without any processing.
- Active states are states that contain certain processing.

- According to the duration criterion, the states are divided in the following way:
- Transient states are those states that contain only one sequence, and when it is executed, a transition to another state occurs. Every active transient state contains one Transient sequence.
- Durative states are those states in which a procedural control entity normally remains for a longer time. The processing of active durative states is divided into several sequences, which can be of different types.

A more detailed discussion on the processing of states (and transitions) is given in a separate section below.

2.2.4. Superstate

Syntactically, a superstate is a graph node denoted by a rectangle with the superstate name written inside. The inner area of the superstate rectangle may contain other superstates, states, and other state transition diagram elements. An example of superstate is the Operating superstate of the procedural control entity PCE2 in Figure 1.

State diagrams are composed of a hierarchy of nested states, with superstates and substates, and with elementary states at the lowest level of the hierarchy. The purpose of nested states may be to incorporate conceptually related entities or, as the most important purpose, to avoid the repetition of information by closing into a superstate the actions and/or transitions and/or dependence relations common to a number of states. Note that a superstate is, in fact, concurrent with its active substates at all nesting levels (there may be as many concurrently active states as the number of nesting levels).

In the nested states in ProcGraph DSML, not only pure tree structures are possible, but also overlapping superstates. In other words, a state may be contained in one of the two superstates, or in both of them.

The state machine model of the ProcGraph DSML does not include the notion of the initial substate. In fact, all TO transitions are drawn explicitly to elementary states, while superstates may only have FROM transitions.

2.2.5. Transition "on Completion"

Syntactically, the transition on completion is a directed line connecting an ordered pair of nodes (source and sink), denoted by a solid line ending with an empty arrowhead. An example of transition on completion is the transition from the state Starting to the state Running of the procedural control entity PCE1 in Figure 1.

Transition on completion is a transition from one state (Source) to another state (Sink), which has no particular cause event, but occurs after the completion of the source state processing. According to the criterion of activity (processing), transitions on completion can be divided into active, containing activity sequences, and inactive, which have no activity sequence. A transition from a superstate cannot be "on completion". Each elementary active transient state must

have exactly one FROM transition of the type "on completion".

2.2.6. Transition "on Event"

Syntactically, the transition on event is a directed line connecting an ordered pair of nodes (source and sink), denoted by a solid line ending with a filled arrowhead. An example of transition on event is the transition from the state Stopped to the state Starting of the procedural control entity PCE1 in **Figure 1**.

Transition on event is a transition from one state (Source) to another state (Sink), which is executed on the occurrence of a particular event. At the beginning of such a transition, first any active processing of the source state is immediately terminated. A transition from a superstate can only be of the type "on event". A transition of the type "on event" from a state S with the causing event defined by the expression State(S) = complete is equivalent to the transition "on completion" from state S.

According to the criterion of activity (processing), transitions on event, similar to those "on completion", can be divided into active, containing activity sequences, and inactive, which have no activity sequence.

2.2.7. ProcGraph State Machine Processing Sequences

The ProcGraph state machine behaviour model is very finely granulated. In other words, we can say that the modelled control entities have a very finely granulated processing.

A very important feature of the processing in the behaviour model is that all processing is composed of sequences with a duration (*i.e.* with non-instantaneous execution), unlike, for example, the various variants of the Statecharts model, where only the Loop processing has duration, while the Entry, the Exit, and the processing of the transitions is instantaneous [17] [18]. The disadvantage of the latter model is in its separation of the states from the processing (actions only serve to trigger the activities, which are separated from the state model); this separation is very likely to bring difficulties with the synchronization of the activities.

As the processing of the transitions in the ProcGraph state machine model has duration, and since a modelled control entity must always be in a known state, the processing of the transitions is considered as a part of the target state and is called *Specific entry processing* (as it executes on the entry and is specific with respect to the source state).

The processing in the ProcGraph state machine model consists of the following elements:

1) The processing of states, for each state up to one sequence of each sequence type, defined as

- ENTRY sequence, which is executed only once on entry to a given active durative state,
- LOOP sequence, which is executed cyclically all the time while a procedural control entity is in a given active durative state (for elementary states also the

opposite is true—the completion of the Loop sequence implies the completion of the state, hence the two expressions are logically equivalent: *state completion* ≡ *Loop sequence completion*),

- EXIT sequence, which is executed only once at the exit from a given active durative state,
- ALWAYS sequence, which is executed concurrently with all activities of an active durative state, including the sequences of the transitions into that state (its specific ENTRY sequence),
- Transient sequence, which is executed only once at active transient states; and

2) Sequences of the transitions into a state, which are considered as its specific Entry sequences, as stated above.

As a particular detail, let us mention at this point that the ENTRY and LOOP sequences are not executed if, at the entry to a state, the condition for its termination is satisfied; however, the EXIT sequence is executed in every case.

2.3. ProcGraph State Machine Execution

2.3.1. Transitions Firing Susceptibility

The firing susceptibility of transition T1 begins when the processing of transition T2 traverses the source state of transition T1. In terms of processing sequences, the transition becomes susceptible after enabling the Always processing and before starting the transition's source state Entry sequence (if source state is a superstate, **Figure 2(a)**), or before starting the transition's source state Specific entry sequence, that is the sequence of the transition T2 (if source state is an elementary state, **Figure 2(b)**).

The firing susceptibility of transition T1 ends when the processing of transition T2 traverses the source state of transition T1, or just before starting transition's source state Exit sequence (**Figure 3(a)**). The firing susceptibility of transition T1 also ends on the firing of transition T2 from the same state (**Figure 3(b)**).

2.3.2. Transitions Priorities

If the transition conditions for more susceptible transitions become true at a given moment, the transition with the highest priority position will be activated according to the following rules:

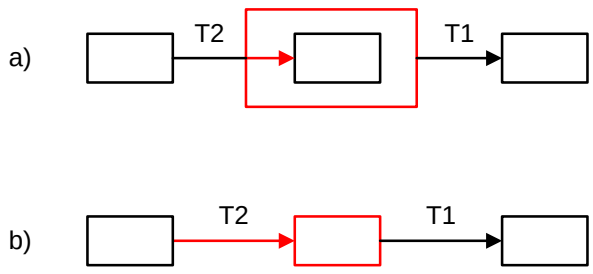

Figure 2. Begin of firing susceptibility.

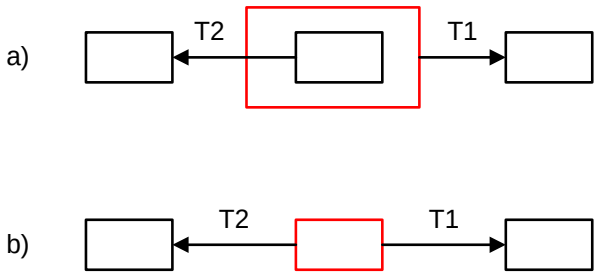

Figure 3. End of firing susceptibility.

1) Transitions caused by propagations have higher priority than ordinary transitions.

2) Among transitions at different levels in the hierarchy of states, higher priority is assigned to transitions with the source in higher-level (outer) superstates.

3) Among transitions with the sources at the same level in the hierarchy of states, higher priority is assigned to transitions with the sink in higher-level (outer) superstates.

4) If the transition conditions for more susceptible transitions with both sources and sinks at the same level become true at a given moment, the choice of a transition to be fired is non-deterministic.

2.3.3. The Sequence of Processing on State Transition

Before giving the definition of the sequence of processing on state transition, we shall define the term of the root state. The root state is the elementary state, active at the time of transition firing, that is the source state or a substate of the source state, in case that the source state is a superstate. The sequence of processing on state transition from a source state to a target state is as follows:

1) The processing of all substates of the source state is terminated. This includes all Entry, Loop, Exit, and Always processing and the processing of transitions into the state (specific Entry processing) of all substates.

2) On the path from the source to the sink state, first for the source state, and then successively, following the increasing state hierarchy for each superstate of the root state that is not a superstate of the target state, the following successive steps are performed:

a) termination of the Entry, Loop, Transient and Specific entry processing,

b) execution (starting and waiting for completion) of the Exit processing,

c) termination of the Always processing.

3) The state variable value changes from the source to the sink state.

4) On the path from the source to the sink state, following the decreasing state hierarchy for each superstate of the sink state that is not a superstate of the root state, the following successive steps are performed:

a) enabling the Always processing,

b) execution (starting and waiting for completion) of the Entry processing,

c) enabling the Loop processing.

5) For the sink state, the following successive steps are performed:

a) enabling the Always processing,

b) execution (starting and waiting for completion) of the Specific entry processing (the processing of the transition into the sink state),

c) execution (starting and waiting for completion) of the Entry processing,

d) enabling the Loop processing.

If the sink state is a transient state, then instead of c) and d), there is an execution (starting and waiting for completion) of its Transient processing.

The ProcGraph state machine behaviour model allows overlapping of superstates. For an illustration of the implications of this fact, let us consider a few examples of state transitions in state machines including superstates, which are shown in Figure 4. In the example, the firing of a transition has the following implications on the activation or deactivation of individual superstates.

1) Transition T1 implies activity of elementary state S3 and superstates SS2 and SS3. At the transition, the superstate SS2 has to be activated, while the possible need of the activation of SS3 depends on which elementary state was active at the time of transition firing, as follows:

- if elementary state S1 was active ⇒ superstate SS3 is activated, since it has to be active concurrently with state S3, which is the next active elementary state;
- if elementary state S2 was active ⇒ superstate SS3 is not activated, since it already was active, being a superstate of S2.

2) Transition T2 implies activity of elementary state S4 and superstate SS2. At the transition, the possible need of the deactivation of SS3 is dependent on which elementary state was active at the time of transition firing, as follows:

- if elementary state S1 was active ⇒ superstate SS3 is not deactivated, since it already was non-active, not being a superstate of S1;
- if elementary state S2 was active ⇒ superstate SS3 is deactivated, since it was active concurrently with S2, being its superstate, and it should not be active concurrently with S4, not being its superstate.

2.3.4. State Dependencies Diagram

The third type of diagram is the procedural control entities state-transition dependencies diagram (hereinafter referred to as a state dependencies diagram, SDD).

The SDD is an explosion of a composite dependency in an entity diagram, which exactly defines the mutual behaviour dependencies between the two PCEs

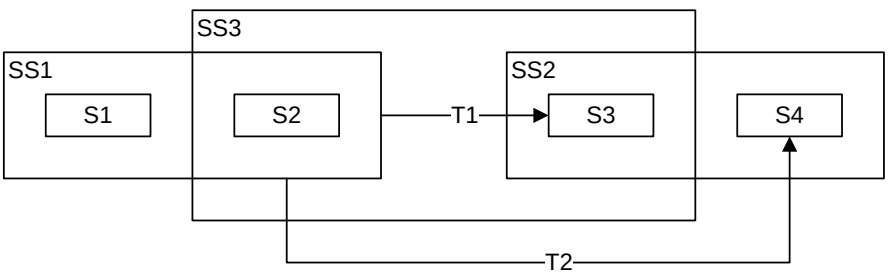

Figure 4. An illustration of state transitions including overlapping superstates.

it connects. An SDD consists of the STDs of two interdependent PCEs and a set of elementary dependencies. The lower part of Figure 1 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 Figure 1 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 Figure 1 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.

There are two additional features of the SDD. The first feature are delayed propagations, which take effect only after the propagation cause is present for a given time (defined by a parameter of the delayed propagation). The delayed propagations are denoted by the symbol Δ.

The second feature are conditioned dependency relations, both conditional and propagational, which have effect only if an associated logical expression has TRUE value. The conditioned dependency relations are denoted by the letter C. It should be mentioned at this point, that the conditioned dependency relations have a potential to increase the coupling and consequently the complexity of the system, therefore they should be used only exceptionally and carefully.

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 Figure 1 the composite dependency in the entities diagram shows that between PCE1 and PCE2 there are one 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.

2.3.5. Definition of Low-Level Processing in ProcGraph

The three above mentioned diagram types represent a high-level behavioural structure of the ProcGraph model, which has to be filled with specific, finely granulated processing definitions, written in a dedicated programming language of the implementation platform (e.g. IEC 61131-3 Structured Text language for the PLC platform). This processing performs the low-level core of 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 to provide the high-level, domain-specific structural framework that supports an appropriate modularisation of the software to procedural control entities, modelling the high-level behaviour of these entities, and defining the dependencies among them.

3. Example Application

3.1. Introduction

Titanium dioxide (TiO_2) is a white pigment that is widely used in paint and enamel production. The production of the TiO_2 pigment in the *Cinkarna* chemical works consists of a succession of several processes, both continuous and batch. One of the processes in the last stage of the production (the so-called final processing stage) is the continuous process of micronisation (micro-grinding) of the dried TiO_2 coarse granulate. A simplified technological scheme of the TiO_2 micronisation process is shown in Figure 5. The physical model of the system contains 10 equipment groups, 157 equipment devices, and 224 I/O points.

Coarsely crushed material with a granulation of up to 0.5 cm enters the micronisation process. The material is transported to the intermediate storage silo (A), from which it is taken and dosed from the weighing silo (B) to the jet mill (C), where it is ground to the desired fineness. Grinding is performed with compressed air (9.5 bar) provided by the compressor (D). Additives that improve certain properties of the pigment, which are stored in the additives storage silo (E), are added to the mill from the additives dosing vessel (F), through a

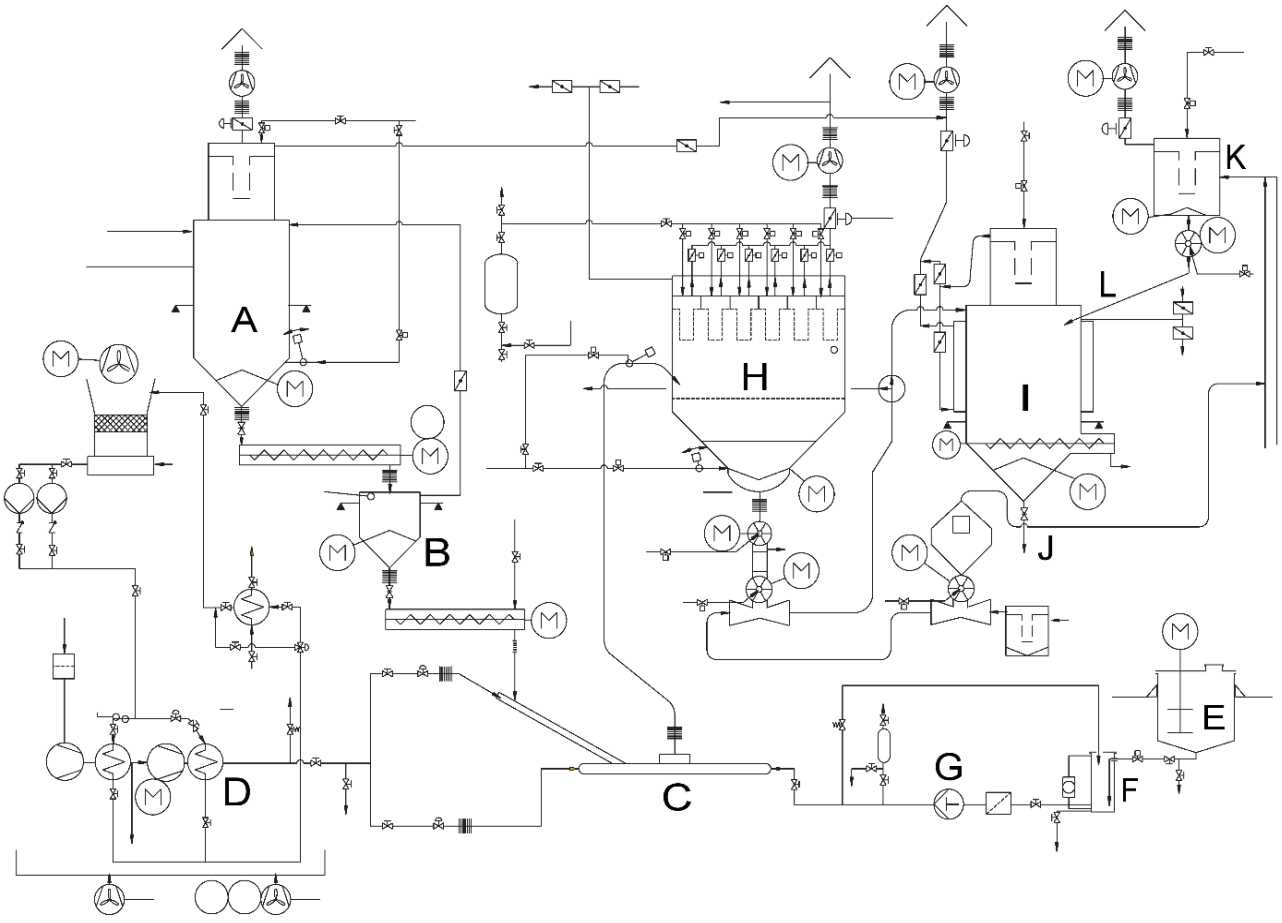

Figure 5. Micronisation process simplified technological scheme.

special dosing system (G). From the jet mill, the material is pneumatically transported first to the separating silo (H) and from there to the packing silo (I) and further to the packing room (J) of the final product. Dust removing (K) is carried out from the packing room, whereby the collected material is returned (L) to the packing silo.

3.2. ProcGraph Model of the Micronisation Process

Due to the limited 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 low-level processing. Omitting the low-level details should not hinder a general understanding of the presented approach and the system considered.

3.2.1. Micronisation Root Diagram

During the requirements analysis process, the procedural control of the micronisation process was decomposed into six operations (highest level procedural control entities), which constitute the highest level entity diagram, called root diagram: Dosing, Jet grinding, Compressed air supply, Pneumatic transport to the separating silo, Pneumatic transport to the packaging silo, and Packaging room dust removing. The root diagram is shown in Figure 6.

From the root diagram we can see that there are six top-level PCEs (operations) and that three of them (Dosing, Pneumatic transport to the separating silo, and Pneumatic transport to the packaging silo) are not elementary, but have further decomposition to sub-activities (not-elementary operations being denoted by rounded rectangles drawn with thick lines). Their lower-level activities are placed in each operation's subED.

The other three operations (Jet grinding, Compressed air supply, Packaging room dust removing) are elementary (denoted by rounded rectangles drawn with thin lines), so those operations have no further decomposition to lower-level activities (in other words, they are elementary PCEs); therefore for

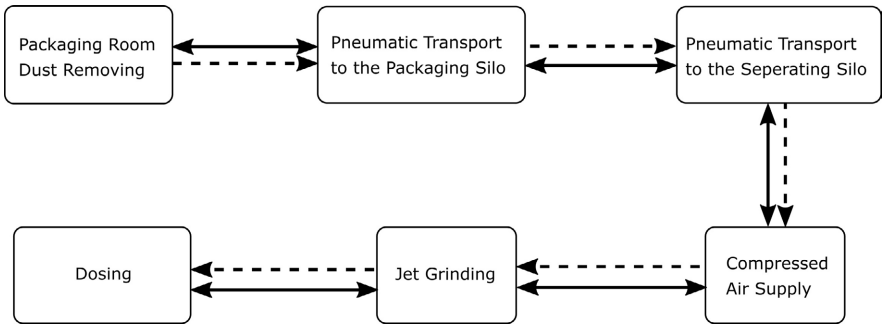

Figure 6. Root diagram of the micronisation process.

those operations the next step is to define their behaviour through respective STDs.

From the root diagram we can also see that there are conditional and propagational dependencies between pair of operations, which form a kind of conditional/propagational chain of operations. So, for example, we can see that between the Jet Grinding and Compressed Air Supply operations there are conditional dependencies in both directions (in each operation there is a state transition that is dependent from other operation being in a certain state). We can also see, that there is a propagational dependency directed from Compressed Air Supply to Jet Grinding operation, which means that a certain state of Compressed Air Supply operation causes (is propagated to) a certain transition of Jet Grinding operation.

3.2.2. Operation Dosing

The operation Dosing is non-elementary, hence it is decomposed into a new ED containing its sub-activities. The entity diagram of the Dosing operation is shown in Figure 7.

The Dosing operation entity diagram in Figure 7 shows that this operation is composed of four subactivities, namely:

1) Dosing.Core, which performs the main functional part of the Dosing operation:

- dosing coarse granulate, by means of (negative ramp) weight control in the weighing silo,
- dosing additives, by means of flow control from the additives dosing vessel;

2) Coarse Granulate Weighing Silo Pre-Charging (CGWSPC);

3) Coarse Granulate Weighing Silo Hammering (CGWSH);

4) Additives Dosing Vessel Pre-Charging (ADVPC).

In Figure 7 aproxy symbol of the Jet Grinding operation appears. The root ED in Figure 6 showed the existence of conditional and propagational dependencies between the operations Jet Grinding and Dosing. Since the Dosing operation is non-elementary, *i.e.* it is composed of four sub-activities, as stated above, it has to be defined which of its four sub-activities is/are interdependent with the Jet Grinding operation. The conditional/propagational dependence connections in Figure 7 show that Dosing.Core is the sub-activity of Dosing that is interdependent with the Jet Grinding operation.

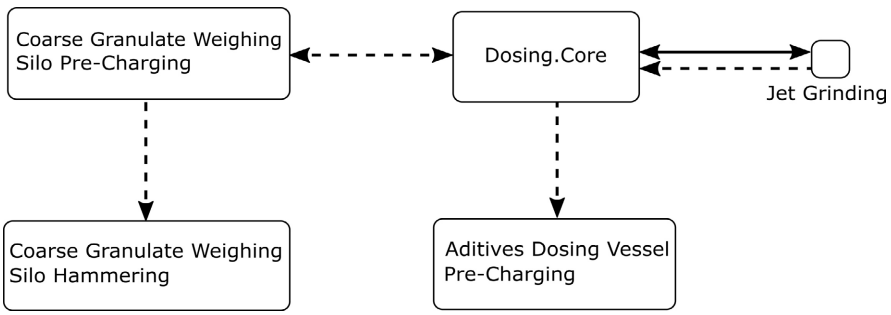

Figure 7. Entity diagram of the operation dosing.

Figure 7 also shows the dependencies between the main sub-activity Dosing.Core and other sub-activities, as follows:

- Additives Dosing Vessel Pre-Charging is propagationally dependent on Dosing.Core,
- Between the Coarse Granulate Weighing Silo Pre-Charging activity and the Dosing.Core activity there is a propagational dependence in both directions,
- Coarse Granulate Weighing Silo Hammering is propagationally dependent on Coarse Granulate Weighing Silo Pre-Charging.

In the following we present the state transition diagrams of the individual elementary sub-activities of the Dosing operation, as well as the state dependencies diagrams showing concrete details regarding the above mentioned propagational dependencies between individual sub-activities of the Dosing operation.

The state transition diagram of the Dosing.Core subactivity is shown in Figure 8.

From the STD in Figure 8 it can be seen that the behaviour of this activity is rather simple and self-explanatory. The starting part is divided into two states, according to the points of the starting sequence at which the two auxiliary suboperations (CGWSPC and GCWSH) have to be started. The Running state is divided into two substates: Weight control, which is active when the coarse granulate weighing silo is not charging, and Weight PV (process value) tracking, which is active when the coarse granulate weighing silo is charging.

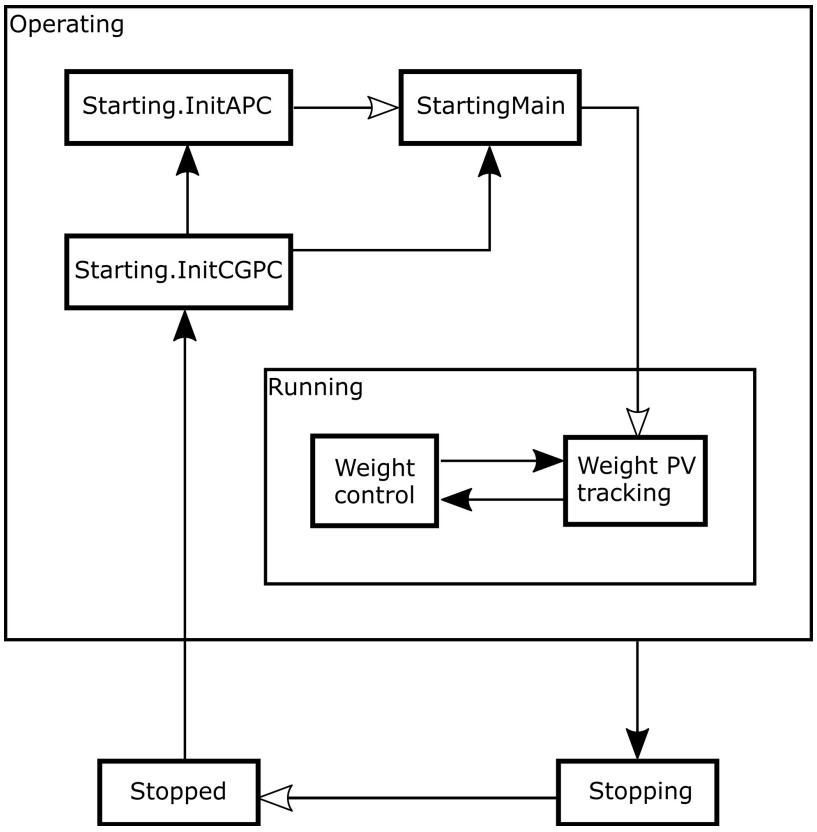

Figure 8. STD of the Dosing sub-activity Dosing.Core.

Figure 9 shows the STD of the activity Dosing.CGWSPC, showing its two substates of the state Running, namely ChargingOFF (charging of the weighing silo is not active, hence the granulate dosing from the weighing silo can be performed through weight control) and ChargingON (charging of the weighing silo is active, hence the weight control is in a PV (process value) tracking mode.

Figure 10 shows the bi-directional propagational dependency between Dosing.Core and Dosing.CGWSPC activities. On the one hand, Dosing.CGWSPC is subordinated to Dosing.Core, *i.e.* it is started and stopped via propagational dependencies from Dosing.Core. On the other hand, Dosing.Core is propagationally dependent by Dosing.CGWSPC, since it switches between Weight control and Weight process value tracking states, based on the charging status of Dosing.CGWSPC.

Figure 11 shows the simple Stopped-Running STD of the weighing silo hammering (Dosing.CGWSH) sub-activity, and the propagational dependencies in Figure 12 show that weighing silo hammering sub-activity is active (Running) if and only if the weighing silo pre-charging sub-activity (Dosing.CGWSPC) is in its ChargingON state.

Figure 13 shows the states of the Additives Dosing Vessel Pre-Charging (Dosing.ADVPC) activity, and the SDD in Figure 14 shows that Dosing.ADVPC is subordinated to Dosing.Core, *i.e.* it is started and stopped via propagational dependencies from Dosing.Core.

3.2.3. Operation Jet Grinding

Figure 15 shows the STD of the operation Jet Grinding, while the Figure 16

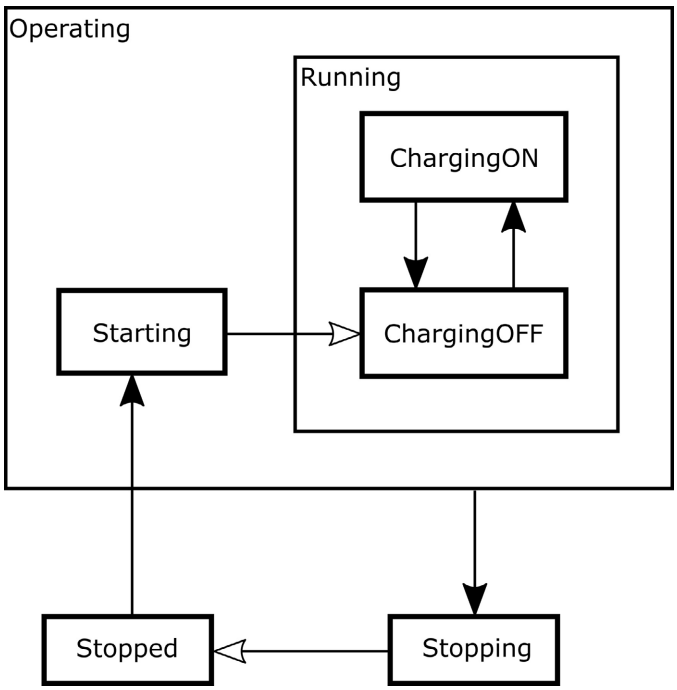

Figure 9. STD of the dosing sub-activity coarse granulate weighing silo pre-charging.

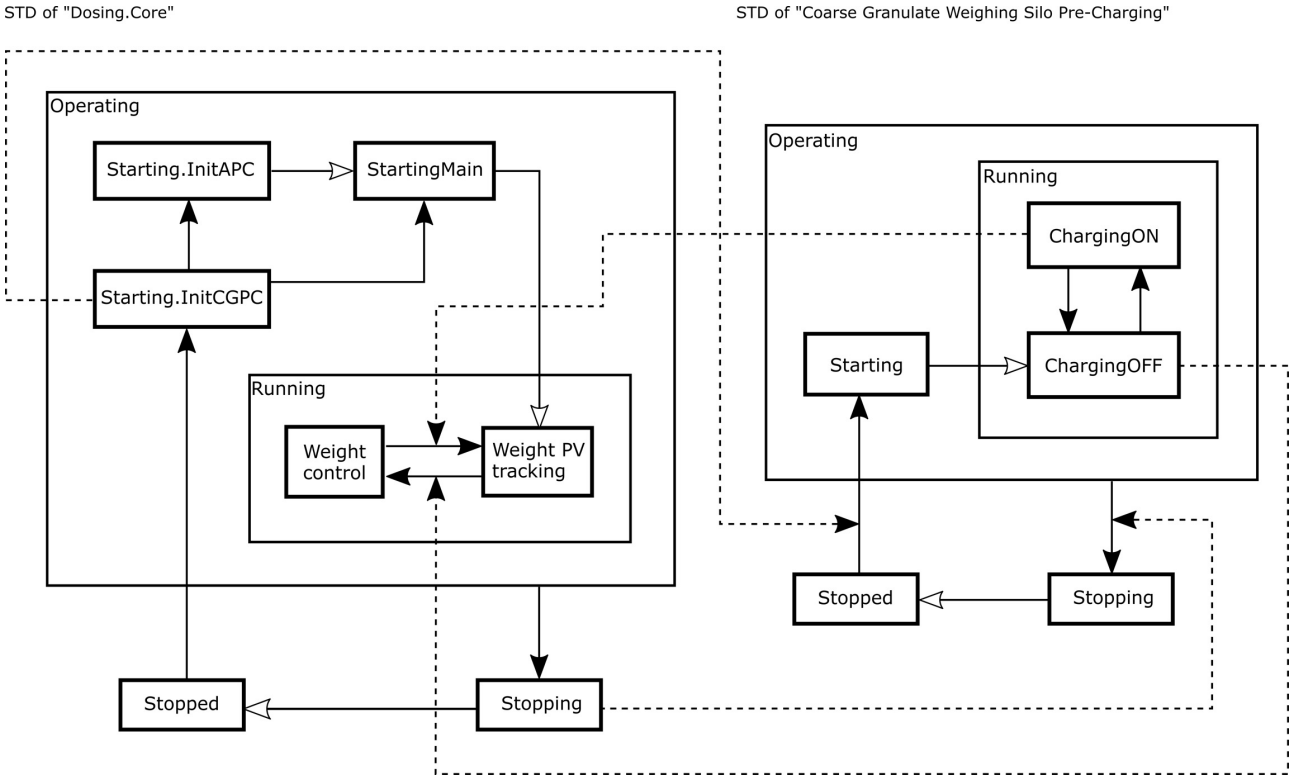

Figure 10. SDD between Dosing.Core and Dosing.CGWSPC.

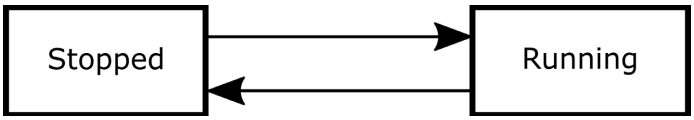

Figure 11. STD of the dosing sub-activity coarse granulate weighing silo hammering.

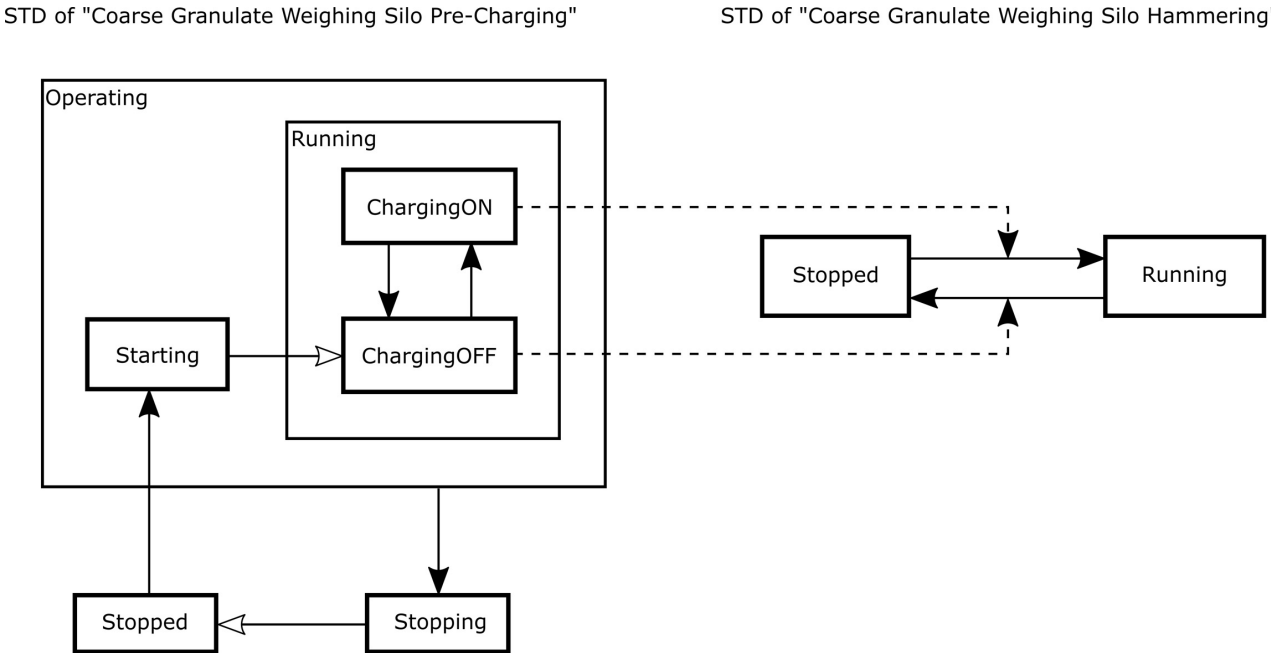

Figure 12. SDD between Dosing.CGWSPC and Dosing.CGWSH.

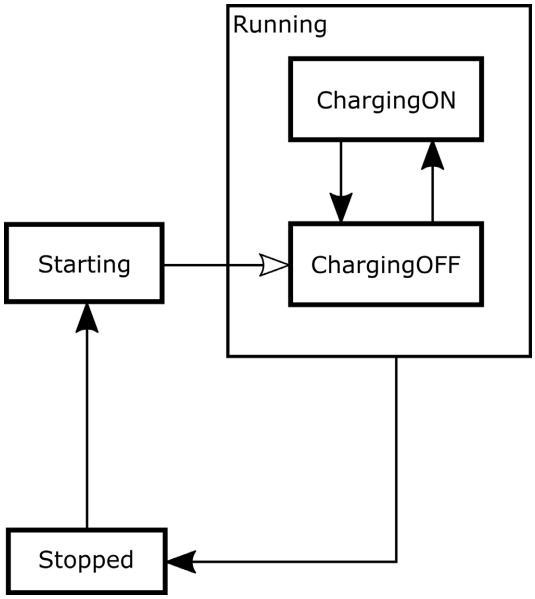

Figure 13. STD of the dosing sub-activity additives dosing vessel pre-charging.

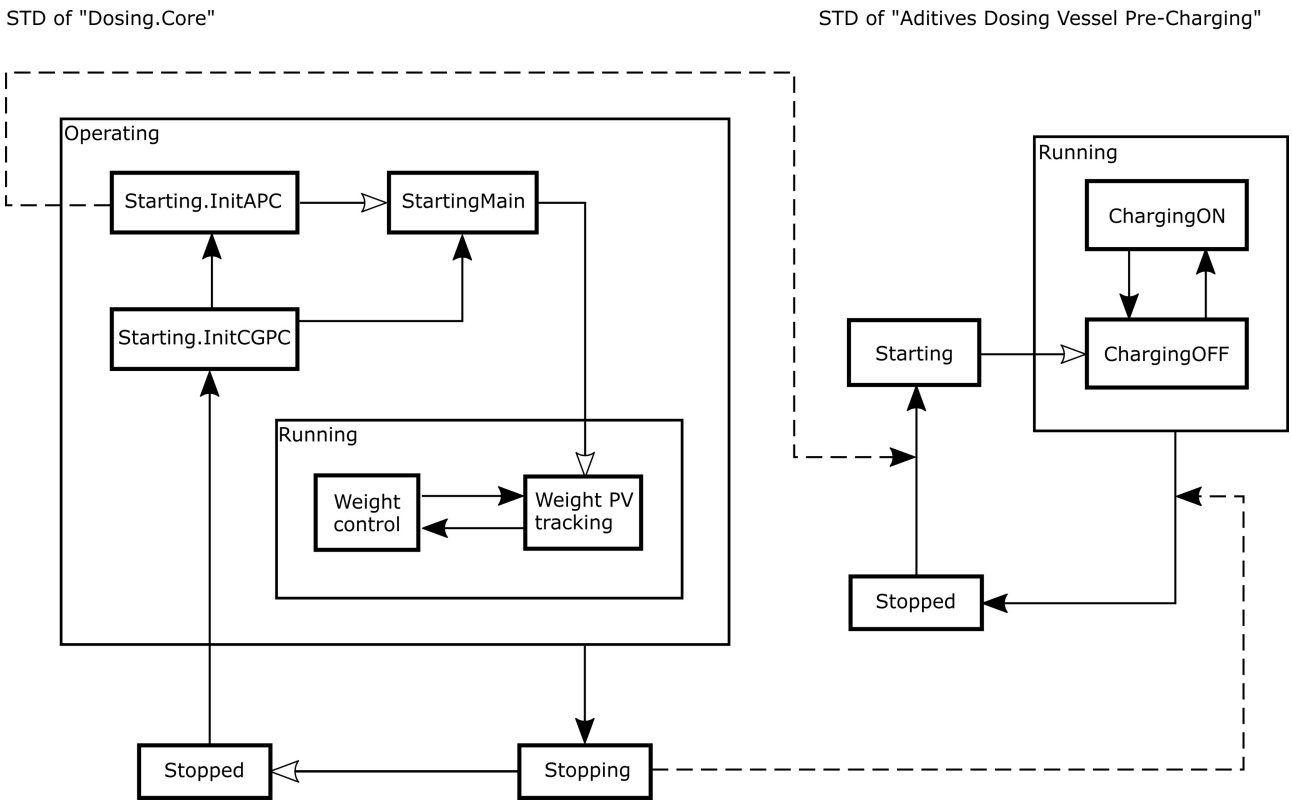

Figure 14. SDD between Dosing.Core and Dosing.ADVPC.

shows the dependencies between the Jet Grinding and the Dosing operation (more specifically its sub-activity Dosing.Core).

Simply stated, the dependency relation is that Dosing isn't allowed to operate if Jet Grinding is not running. This relation implies three dependencies (two

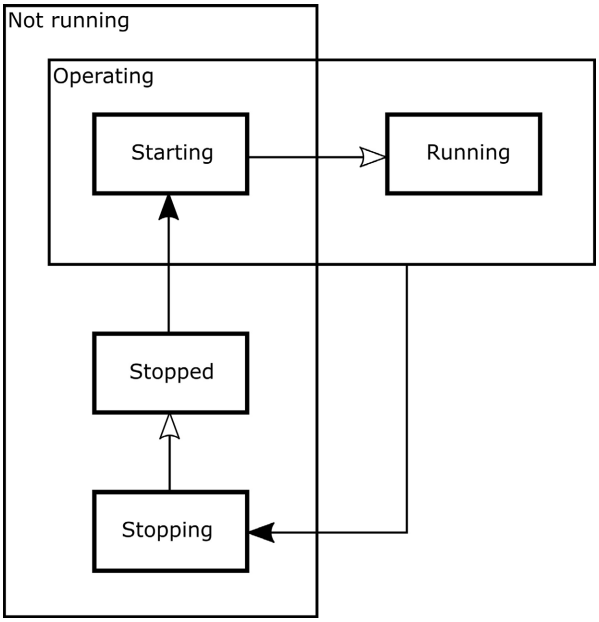

Figure 15. STD of the operation Jet Grinding.

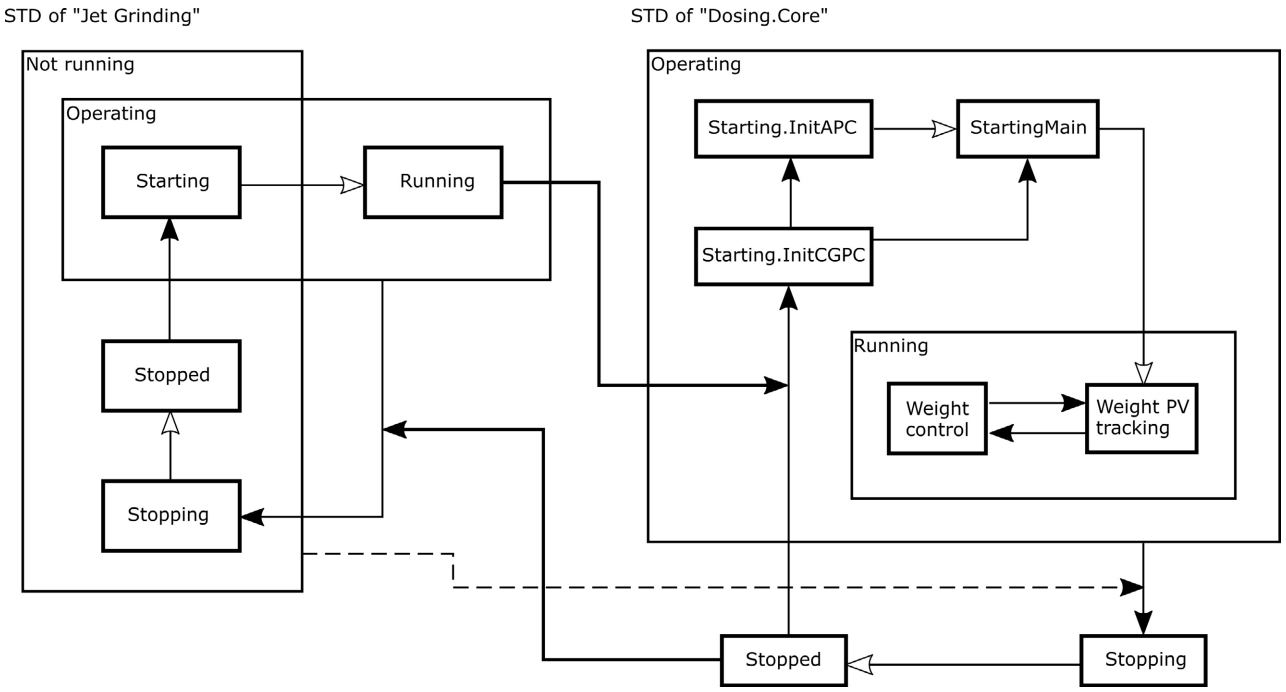

Figure 16. SDD between Dosing.Core and Jet Grinding.

conditional and one propagational) in the SDD between Jet Grinding and Dosing.Core, namely:

1) The condition for the starting of Dosing.Core is that the state of Jet Grinding is Running.

2) The condition for the stopping of Jet Grinding is that the state of Dosing.Core is Stopped.

3) If Dosing.Core is in one of the sub-states of the Operating superstate, and,

at the same time, Jet Grinding isn't in Running state (*i.e.* it is in Not running superstate), the transition of Dosing.Core activity to its Stopping state occurs; in other words, the Not running (super)state of Jet Grinding is propagated to the transition of Dosing.Core from the Operating (super)state to the Stopping state.

3.2.4. Operation Compressed Air Supply

Figure 17 shows the STD of the operation Compressed Air Supply, while the Figure 18 shows the dependencies between the Compressed Air Supply and the

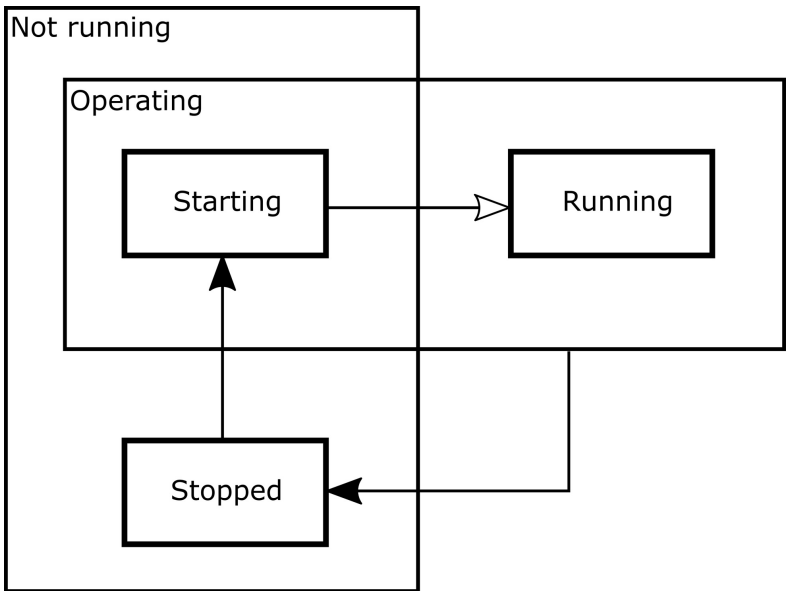

Figure 17. STD of the operation compressed air supply.

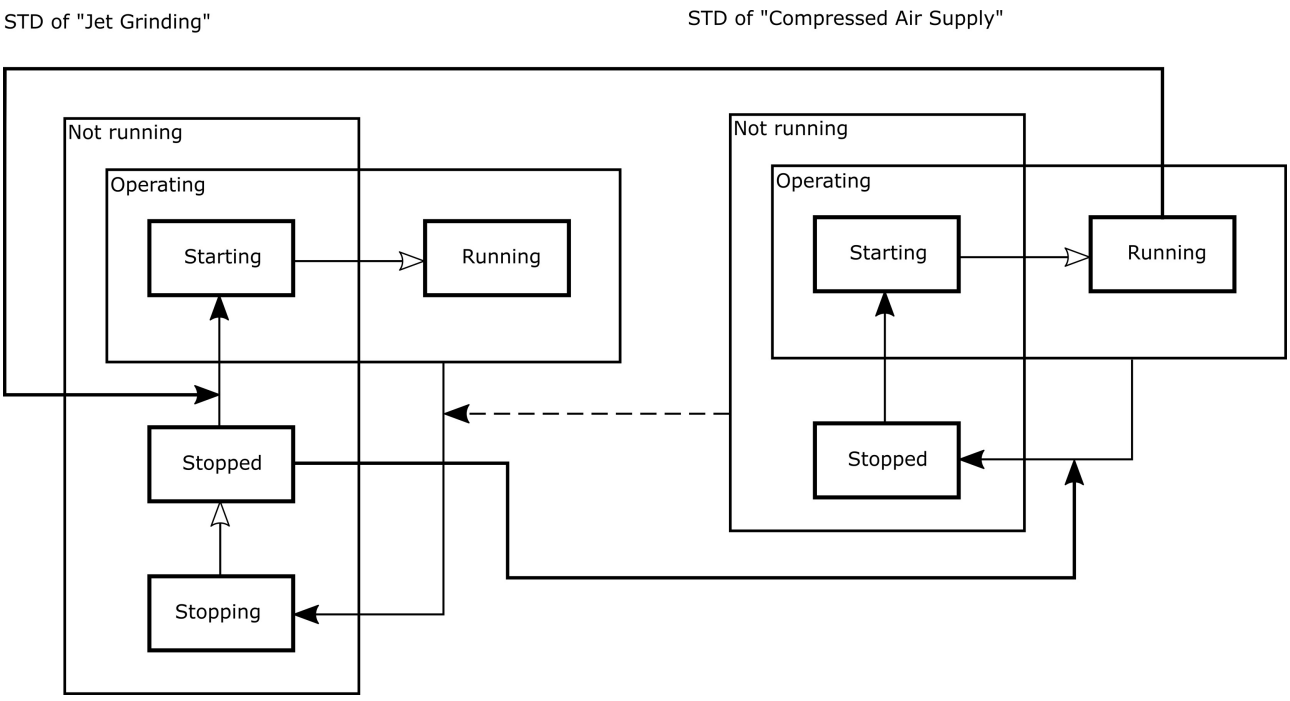

Figure 18. SDD between jet grinding and compressed air supply.

Jet Grinding operation. Simply stated, the dependency relation is that Jet Grinding isn't allowed to operate if Compressed Air Supply is not running. This relation implies three dependencies (two conditional and one propagational) in the SDD between Compressed Air Supply and Jet Grinding, namely:

1) The condition for the starting of Jet Grinding is that the state of Compressed Air Supply is Running.

2) The condition for the stopping of Compressed Ait Supply is that the state of Jet Grinding is Stopped.

3) If Jet Grinding is in one of the sub-states of the Operating superstate, and, at the same time, Compressed Air Supply isn't in Running state (*i.e.* it is in Not running superstate), the transition of Jet Grinding operation to its Stopping state occurs; in other words, the Not running (super)state of Compressed Air Supply is propagated to the transition of Jet Grinding from the Operating (super)state to the Stopping state.

3.2.5. Operation Pneumatic Transport to the Separating Silo

The operation Pneumating Transport to the Separating Silo (PTSS) is non-elementary, hence it is decomposed into a new ED containing its sub-activities. The entity diagram of the PTSS operation is shown in **Figure 19**.

The PTSS operation entity diagram in **Figure 19** shows that this operation is composed of three subactivities, namely:

1) PTSS.Core, which performs the main functional part of the PTSS operation, *i.e.* the pneumatic transport to the separating silo itself;

2) Separating Silo Filter Bags Pneumatic Shaking (SSFBPS);

3) Separating Silo Entry Pneumatic Hammering (SSEPH).

In **Figure 19** there appear two proxy symbols of other operations, which are interdependent with the PTSS operation, namely the Compressed Air Supply operation, and the PTPS operation (more specifically, its PTPS.Core sub-activity).

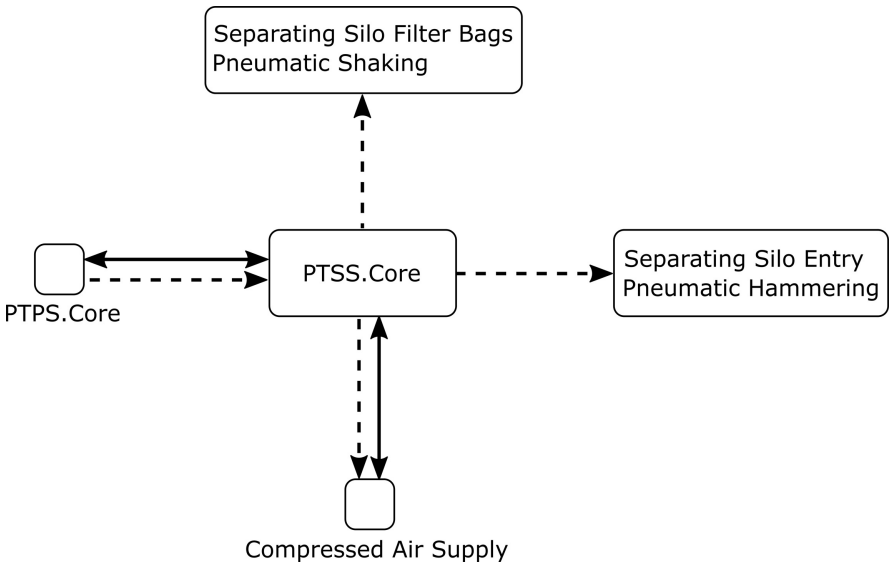

Figure 19. ED of the operation PTSS.

The root ED in Figure 6 showed the existence of conditional and propagational dependencies between the non-elementary operation PTSS and the operations Compressed Air Supply (elementary) and PTPS (non-elementary). Since the PTSS operation is non-elementary, *i.e.* it is composed of three sub-activities, as stated above, it has to be defined which of its three sub-activities is/are interdependent with the Compressed Air Supply operation, and with the PTPS operation. The conditional/propagational dependence connections in Figure 19 show that PTSS.Core is the sub-activity of PTSS that is interdependent with both Compressed Air Supply and PTPS operations.

Figure 19 also shows the dependencies between the main sub-activity PTSS.Core and other sub-activities, as follows:

- Separating Silo Filter Bags Pneumatic Shaking is propagationally dependent on PTSS.Core;
- Separating Silo Entry Pneumatic Hammering is propagationally dependent on PTSS.Core.

In the following we present the state transition diagrams of the individual elementary sub-activities of the PTSS operation, as well as the state dependencies diagrams showing concrete details regarding the above mentioned propagational dependencies between individual sub-activities of the PTSS operation.

The state transition diagram of the PTSS.Core subactivity is shown in Figure 20. From the STD it can be seen that the behaviour of this activity is rather simple and self-explanatory. The starting part is divided into two states, according to the point of the starting sequence at which the two auxiliary sub-activities (SSFBPS and SSEPH) have to be started. On the other hand, the stopping part is divided into four states, according to the points of the stopping sequence at which the two auxiliary sub-activities (SSFBPS and SSEPH) have to be stopped.

Figure 21 shows the simple Stopped-Running STD of the PTSS.SSFBPS sub-activity, with the Running (super)state divided into two states, namely Regular (pneumatic shaking operating on the regular basis, based on the pressure difference between the two sides of the filter bags) and Unconditional (pneumatic shaking operating unconditionally, regardless the pressure difference between the two sides of the filter bags).

The propagational dependencies in Figure 22 show that the PTSS.SSFBPS sub-activity is started on PTSS.Core entering the Starting.End state, forced to unconditional shaking on PTSS.Core entering the Stopping.3 state, and that it is stopped on PTSS.Core entering the Stopping.4 state.

Figure 23 shows the simple Stopped-Running STD of the PTSS.SSEPH sub-activity. The propagational dependencies in Figure 24 show that the PTSS.SSEPH sub-activity is started on PTSS.Core entering the Starting.End state, and that it is stopped on PTSS.Core entering the Stopping.2 state.

The ED of the operation PTSS in Figure 19 showed the conditional-propagational dependencies of the operation PTSS (its sub-activity PTSS.Core) with two other operations, namely PTPS (described in the next section), and Compressed Air Supply, described in the following. Figure 25 shows the dependencies

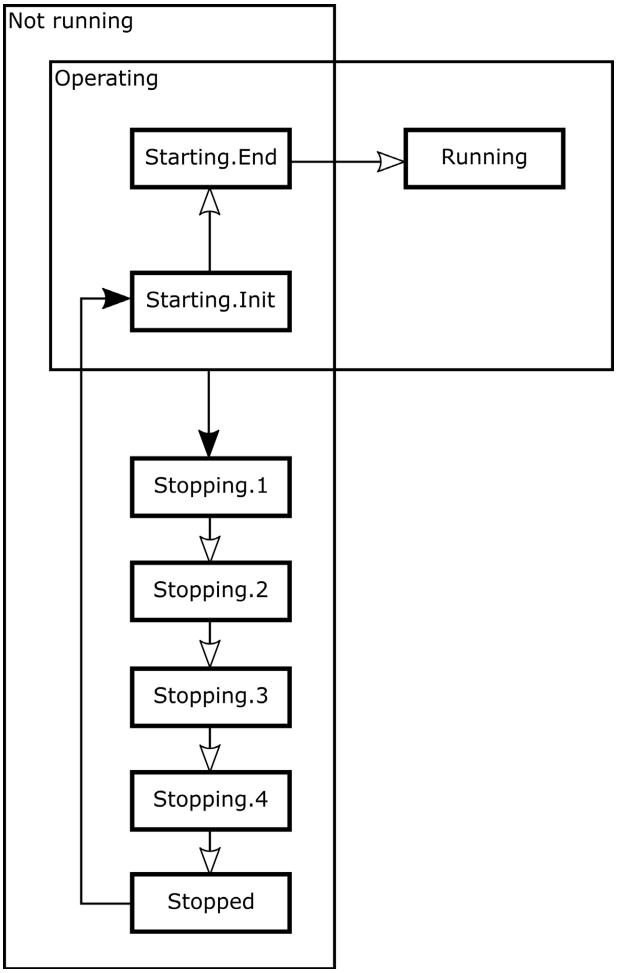

Figure 20. STD of the sub-activity PTSS.Core.

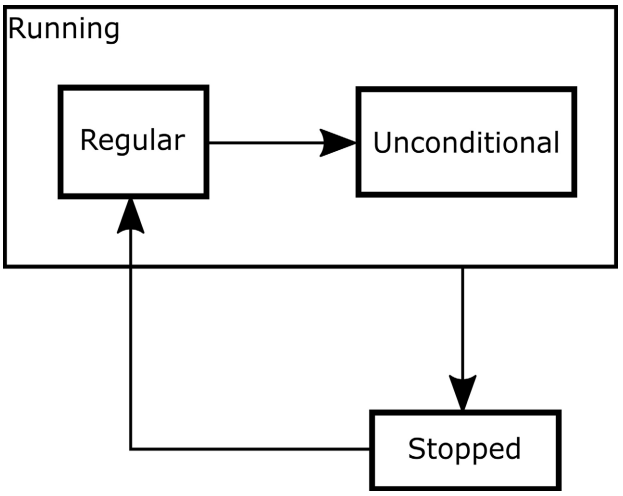

Figure 21. STD of the sub-activity PTSS.SSFBPS.

between the PTSS.Core sub-activity and the Compressed Air Supply operation. Simply stated, the dependency relation is that Compressed Air Supply isn't allowed to operate if PTSS.Core is not running. This relation implies three

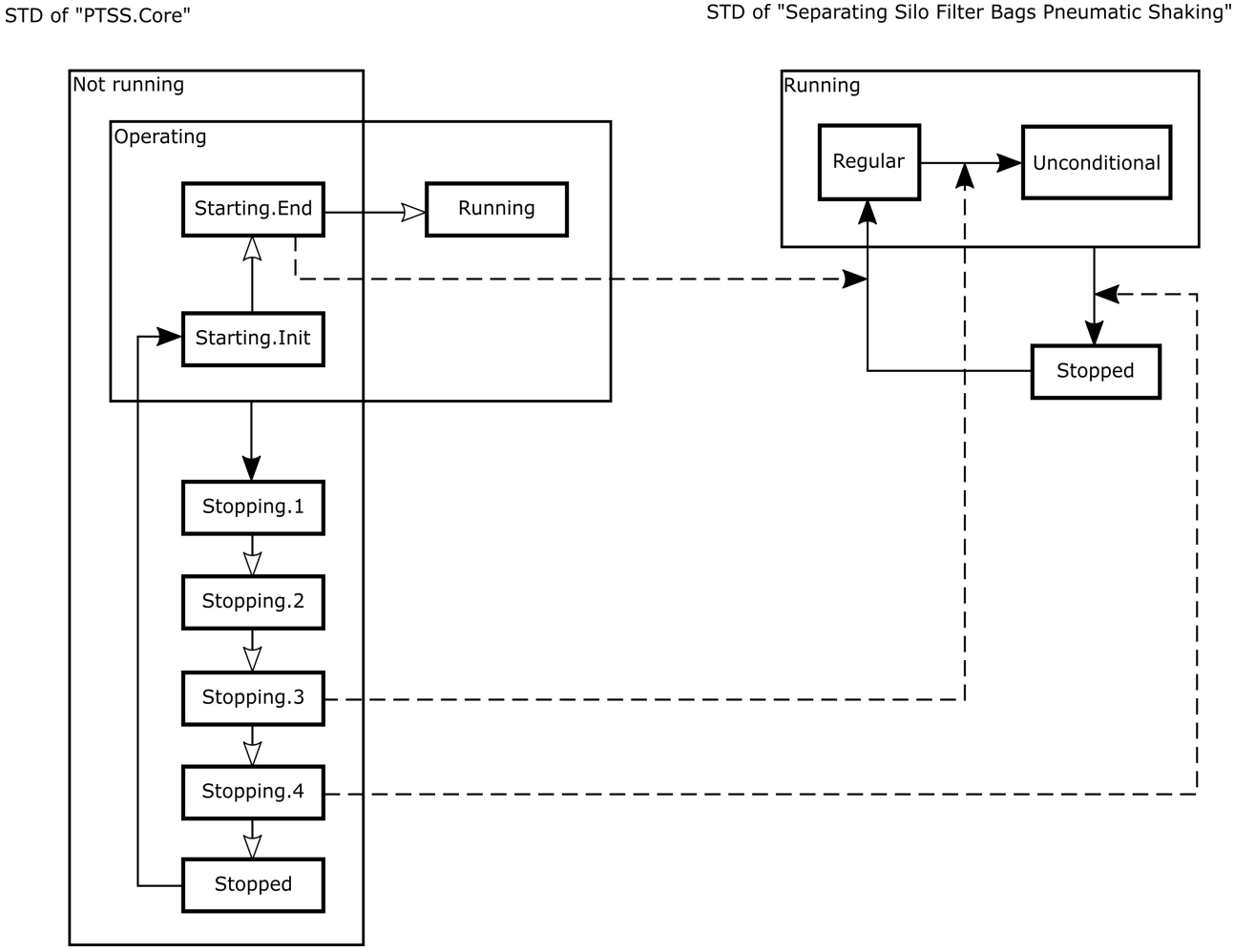

Figure 22. SDD between PTSS.Core and PTSS.SSFBPS.

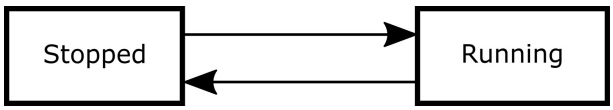

Figure 23. STD of the sub-activity PTSS.SSEPH.

dependencies (two conditional and one propagational) in the SDD between Compressed Air Supply and PTSS.Core, namely:

1) The condition for the starting of Compressed Air Supply is that the state of PTSS.Core is Running.

2) The condition for the stopping of PTSS.Core is that the state of Compressed Air Supply is Stopped.

3) If Compressed Air Supply is in one of the sub-states of the Operating superstate, and, at the same time, PTSS.Core isn't in Running state (*i.e.* it is in Not running superstate), the transition of Compressed Air Supply operation to its Stopping state occurs; in other words, the Not running (super)state of PTSS.Core is propagated to the transition of Compressed Air Supply from the Operating (super)state to the Stopped state.

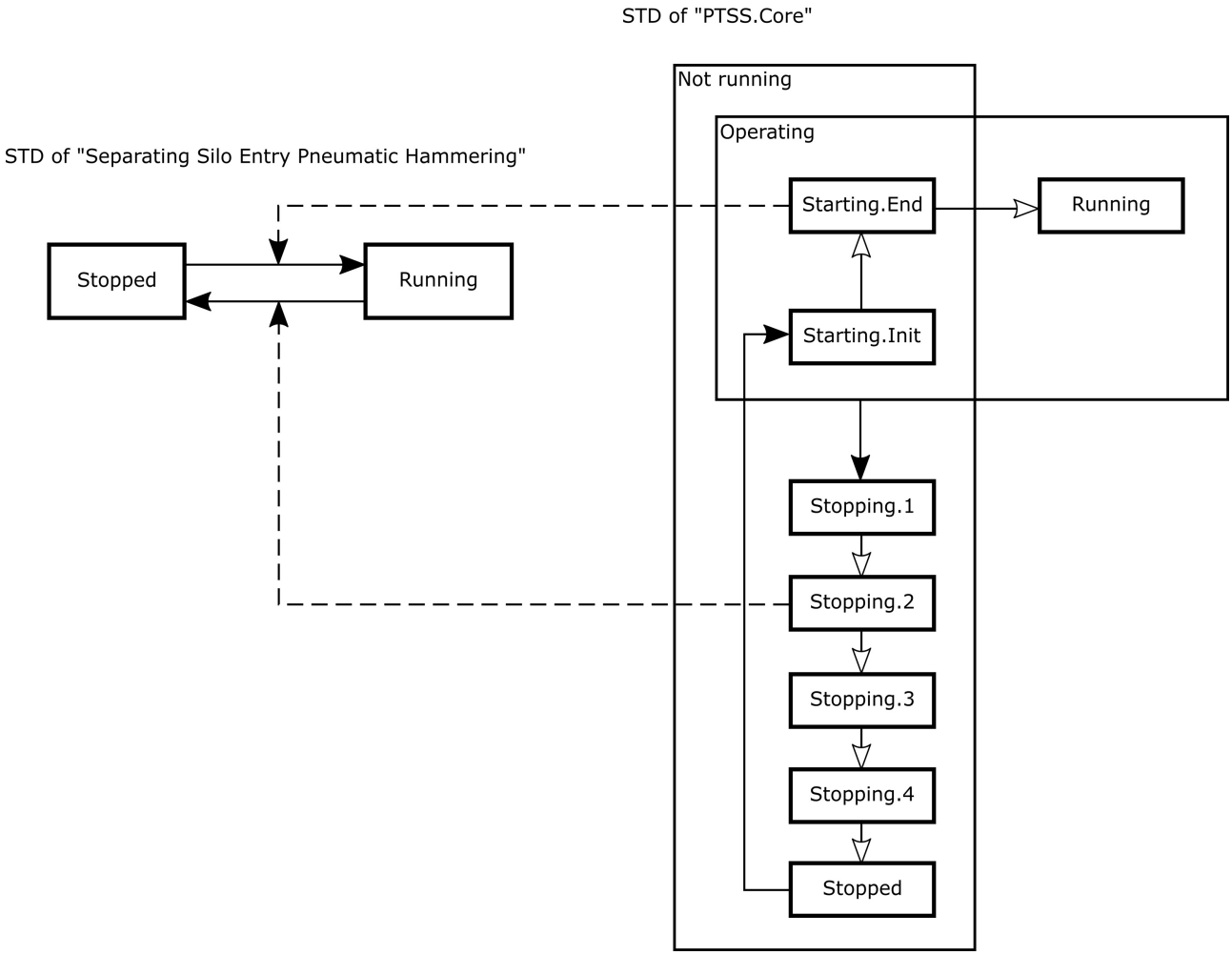

Figure 24. SDD between PTSS.Core and PTSS.SSEPH.

3.2.6. Operation Pneumatic Transport to the Packaging Silo

The operation Pneumating Transport to the Packaging Silo (PTPS) is non-elementary, hence it is decomposed into a new ED containing its sub-activities. The entity diagram of the PTPS operation is shown in **Figure 26**.

The PTPS operation entity diagram in **Figure 26** shows that this operation is composed of two subactivities, namely:

1) PTPS.Core, which performs the main functional part of the PTPS operation, *i.e.* the pneumatic transport to the packaging silo itself;

2) Separating Silo Bottom Pneumatic Hammering (SSBPH).

In **Figure 26** there appear two proxy symbols of other operations, which are interdependent with the PTPS operation, namely the PTSS operation (more specifically, its PTSS.Core sub-activity), and the Packaging Room Dust Removing operation. The root ED in **Figure 6** showed the existence of conditional and propagational dependencies between the non-elementary operation PTPS and the operations Packaging Room Dust Removing (elementary) and PTSS (non-elementary). Since the PTPS operation is non-elementary, *i.e.* it is composed of two sub-activities, as stated above, it has to be defined which of its two

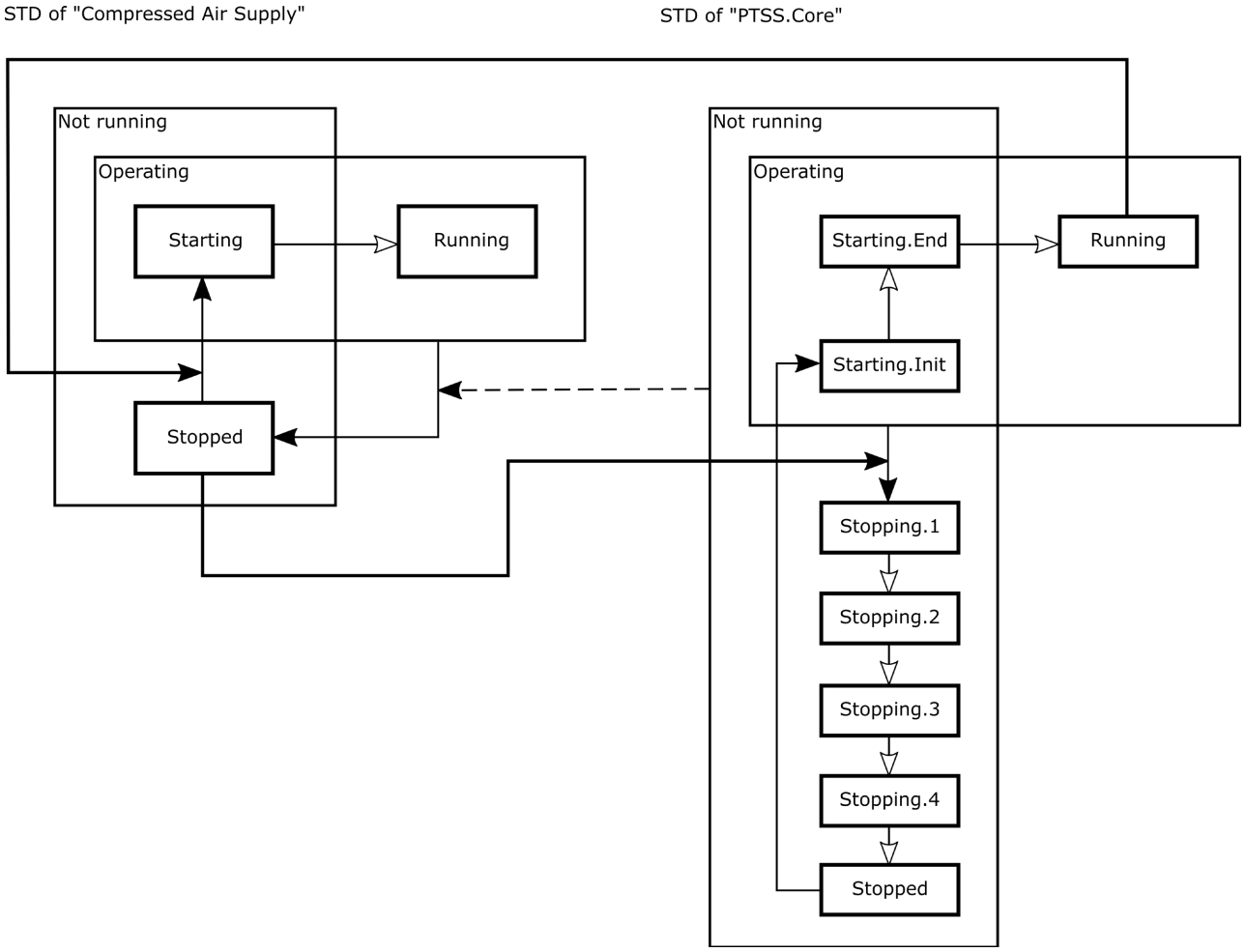

Figure 25. SDD between CAS and PTSS.Core.

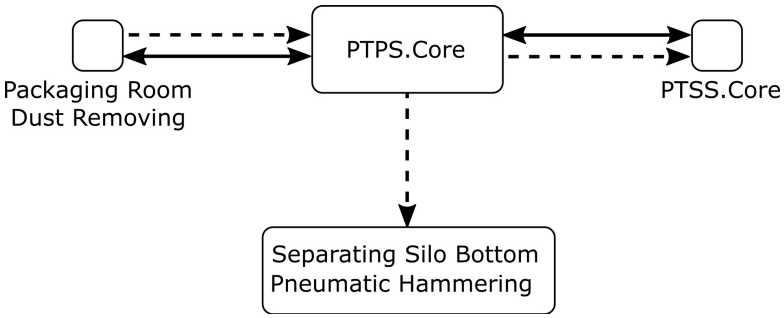

Figure 26. ED of the operation PTPS.

sub-activities is/are interdependent with the PTSS operation, and with the Packaging Room Dust Removing operation. The conditional/propagational dependence connections in Figure 26 show that PTPS.Core is the sub-activity of PTPS that is interdependent with both PTSS and Packaging Room Dust Removing operations.

Figure 26 also shows the dependencies between the sub-activity Separating Silo Bottom Pneumatic Hammering is propagationally dependent on PTPS.Core.

In the following we present the state transition diagram of the PTPS sub-activity Separating Silo Bottom Pneumatic Hammering, as well as the state dependencies diagrams showing concrete details regarding the above mentioned propagational dependencies between the two sub-activities of the PTPS operation.

The state transition diagram of the PTPS.Core subactivity is shown in Figure 27. From the STD it can be seen that the behaviour of this activity is rather simple and self-explanatory. The starting part is divided into two states, according to the point of the starting sequence at which the auxiliary sub-activity (SSBPH) has to be started. On the other hand, the stopping part is also divided into two states, according to the points of the stopping sequence at which the auxilliary sub-activity has to be stopped. The (super)state Running has two states, namely Running with or without feedback from the packaging room, which is defined by a technological parameter; the value of that parameter can be changed at any time, also when the operation PTPS is running.

Figure 28 shows the simple Stopped-Running STD of the PTPS.SSBPH sub-activity. The propagational dependencies in Figure 29 show that the PTPS.SSBPH sub-activity is started on PTPS.Core entering the Starting.End state, and that it is stopped on PTPS.Core entering the Stopping.End state.

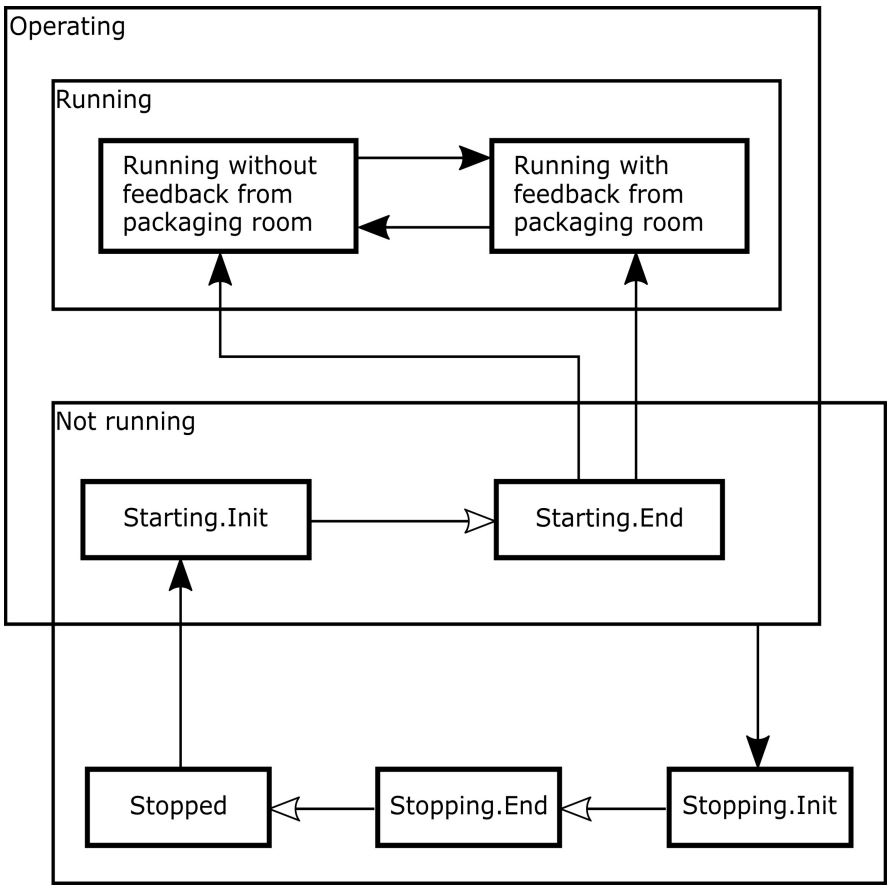

Figure 27. STD of the sub-activity PTPS.Core.

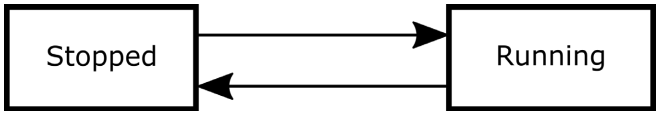

Figure 28. STD of the sub-activity PTPS.SSBPH.

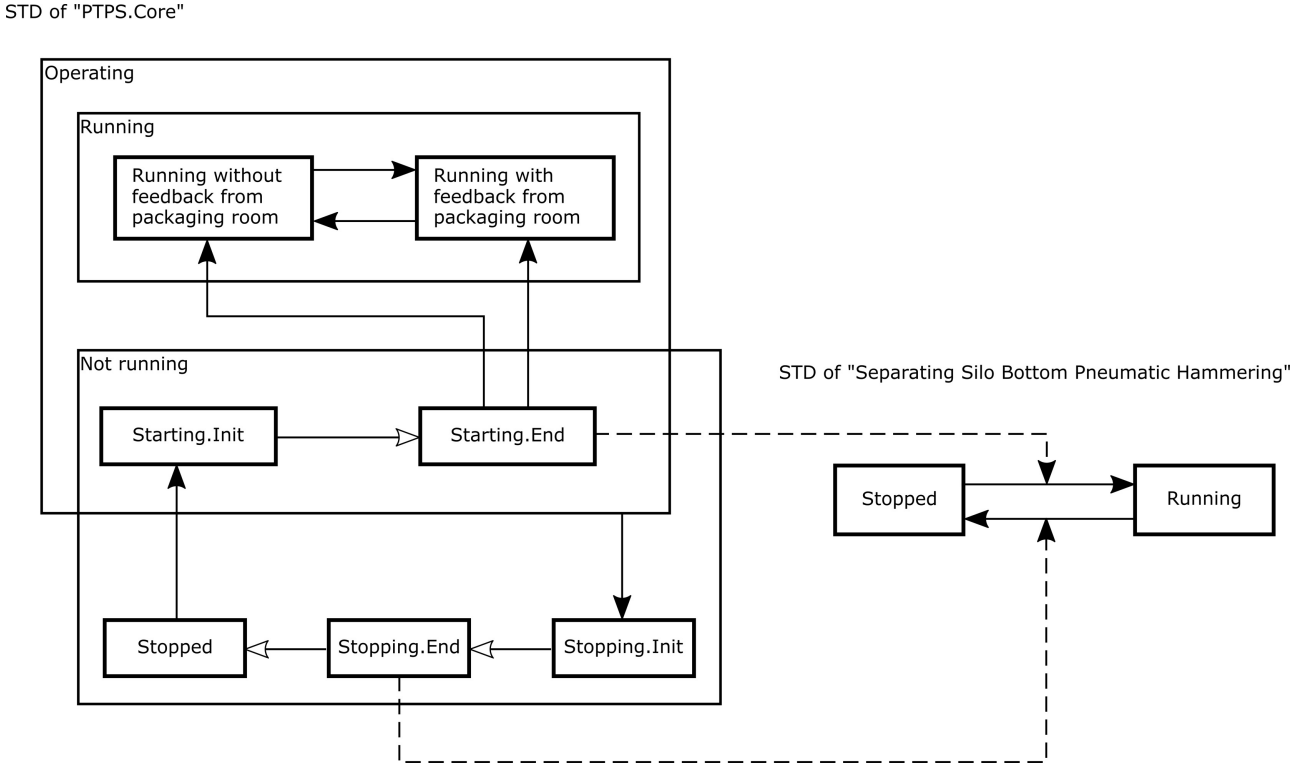

Figure 29. SDD between PTPS.Core and PTPS.SSBPH.

The ED of the operation PTPS in Figure 26 showed the conditional-propagational dependencies of the operation PTPS (its sub-activity PTPS.Core) with two other operations, namely Packaging Room Dust Removing (described in the next section), and PTSS (its sub-activity PTSS.Core), described in the following. Figure 30 shows the dependencies between the PTPS.Core PTSS.Core. Simply stated, the dependency relation is that PTSS.Core isn't allowed to operate if PTPS.Core is not running. This relation implies three dependencies (two conditional and one propagational) in the SDD between PTSS.Core and PTPS.Core, namely:

1) The condition for the starting of PTSS.Core is that the state of PTPS.Core is Running.

2) The condition for the stopping of PTPS.Core is that the state of PTSS.Core is Stopped.

3) If PTSS.Core is in one of the sub-states of the Operating superstate, and, at the same time, PTPS.Core isn't in Running state (*i.e.* it is in Not running superstate), the transition of PTSS.Core activity to its stopping sequence (Stopping.1 state) occurs; in other words, the Not running (super)state of PTPS.Core is propagated to the transition of PTSS.Core from the Operating (super)state to the

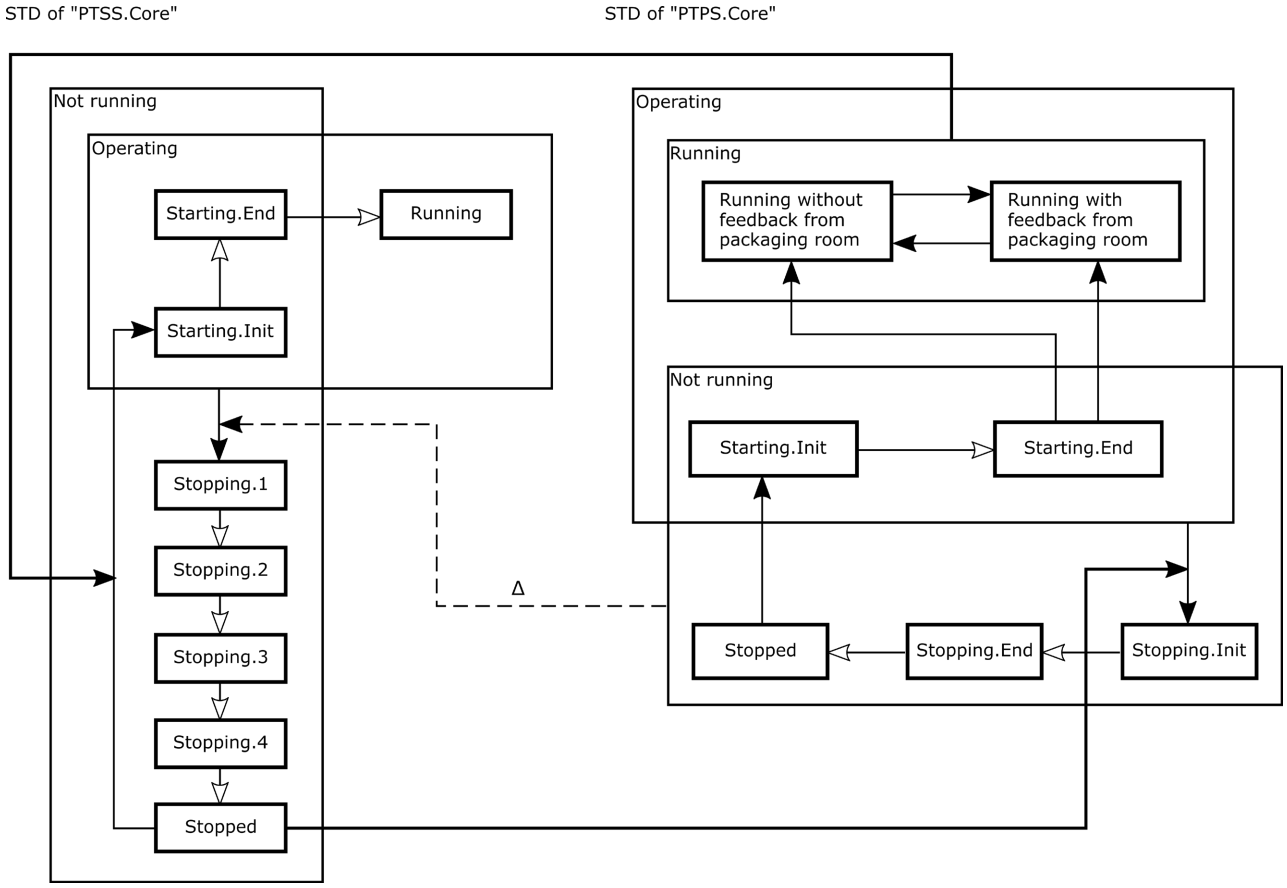

Figure 30. SDD between PTPS.Core and PTSS.Core.

Stopping.1 state. As a particular detail, let us mention that this propagation is delayed, *i.e.* it doesn't occur immediately on PTPS.Core entering its Not running state, but with a delay, defined by a technological parameter. In other words, it is allowed for PTSS.Core to operate when PTPS.Core is not running, however only for a limited time, defined by a parameter of the dependency.

3.2.7. Operation Packaging Room Dust Removing

Figure 31 shows the STD of the operation Packaging Room Dust Removing (PRDR), while the **Figure 32** shows the dependencies between the PRDR and the PTPS operation (more specifically its sub-activity PTPS.Core). Note that in this case the dependencies are conditional, meaning thar they take effect only if a condition C is TRUE, where C: = *Running with feedback from packaging room* = TRUE.

Simply stated, the dependency relation is that PTPS.Core isn't allowed to operate if the operation PTPS has feedback from packaging room turned on and the operation Packaging Room Dust Removing is not running. This relation implies three dependencies (two conditional and one propagational) in the SDD between PRDR and PTPS.Core, namely:

1) If *Running with feedback from packaging room* = TRUE, then the condition for the starting of PTPS.Core is that the state of PRDR is Running;

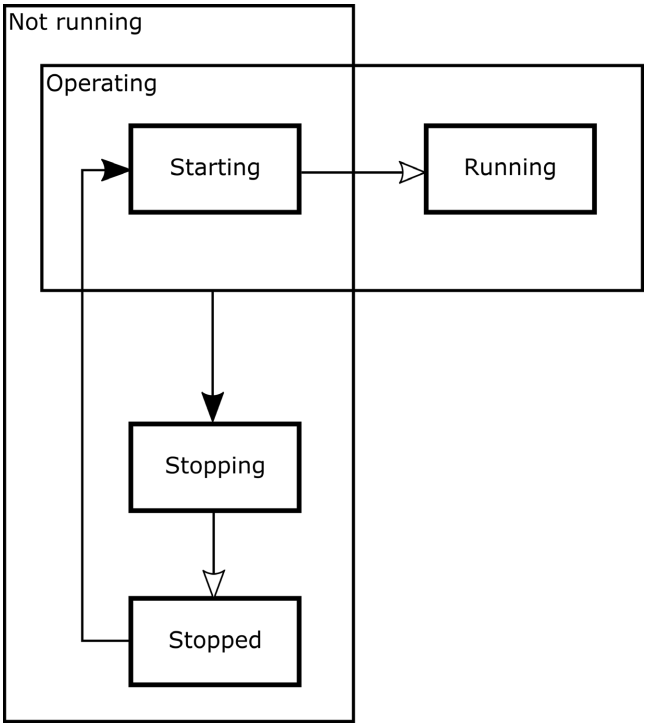

Figure 31. STD of the operation PRDR.

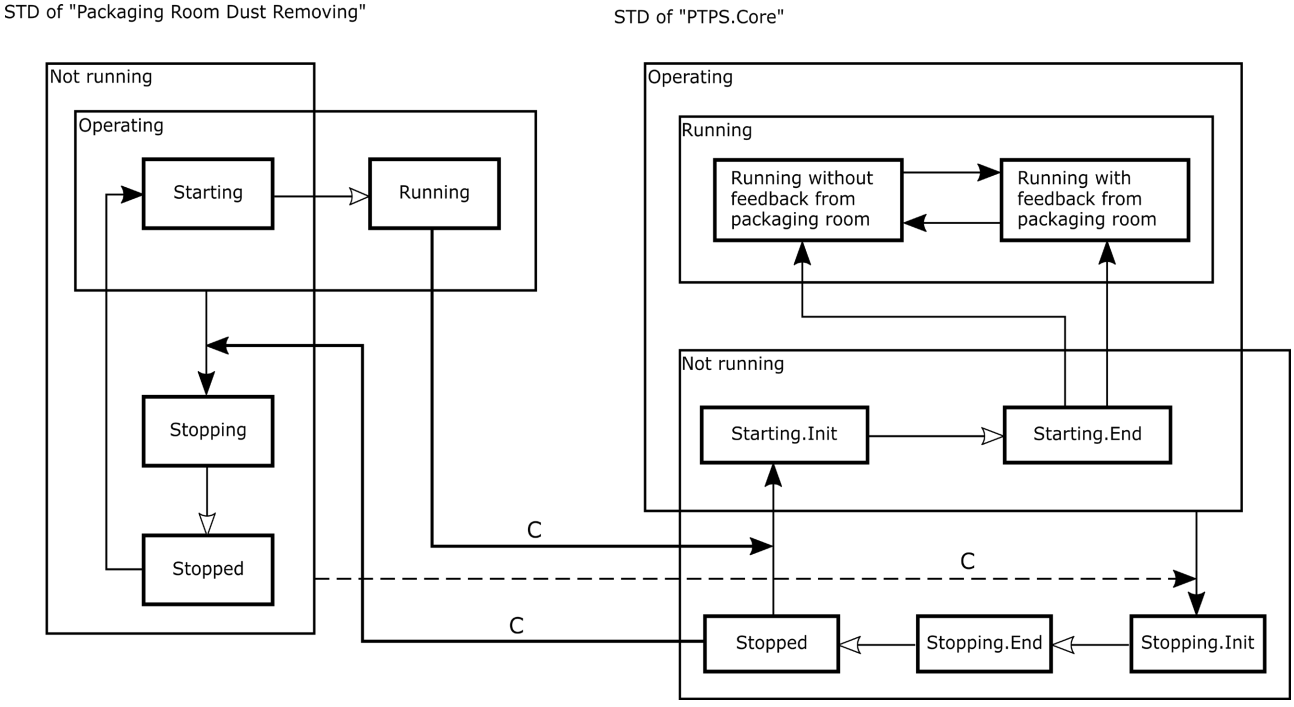

Figure 32. SDD between PRDR and PTPS.Core.

2) If *Running with feedback from packaging room* = TRUE, then the condition for the stopping of PRDR is that the state of PTPS.Core is Stopped;

3) If *Running with feedback from packaging room* = TRUE, and if PTPS.Core is in one of the sub-states of the Operating superstate, and, at the same time,

PRDR isn't in Running state (*i.e.* it is in Not running superstate), the transition of PTPS.Core activity to its stopping sequence (Stopping.Init state) occurs; in other words, if *Running with feedback from packaging room* = TRUE, then the Not running (super)state of PRDR is propagated to the transition of PTPS.Core from the Operating (super)state to the Stopping.Init state.

3.3. Final Remarks on the Example Application

The technological process under consideration is a mid-size and mid-complexity continuous process, and thus not very simple, however, the example shows that it can be mastered in a very transparent and easy way using elegant graphic specifications with the adequate expressive power, which allows efficient analysis and communication between the analyst and the process engineers during the early development phases, as well as between the analyst and the programmers in the implementation and deployment phases. The graphic specifications in the example have a rather fine granularity, as a result, sequential processing in the states and transitions is very straightforward and the size of individual sequences ranges from just a few lines to no more than a few tens of lines of code in IEC 61131-3 Structured Text language.

4. Conclusion

The paper presents a software engineering approach to analysis and design of process control software, based on an innovative high-level, domain-specific modelling language ProcGraph. The approach is process-oriented, unlike the equipment-oriented methods that are commonly used in process control software design, and hence it is closer to the process engineer's point of view, which facilitates the mapping of the process engineer's requirements into software analysis and design model, reducing software development time and improving the quality of the software. Furthermore, the specification can easily be verified and, consequently, has a great potential to achieve a high degree of correctness. Finally, it can be routinely transformed into the program code, allowing the code to be verified and to achieve a near-zero defect in the coding phase. Hence, the presented type of development paradigm could be characterised as an example of »correct by construction "process, in contrast to the nowadays common »construct by correction" process.

Acknowledgements

The authors gratefully acknowledge the support of the Slovenian Research Agency through the programme P2-0001.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Lukman, T., Godena, G., Gray, J., Heričko, M. and, Strmčnik, S. (2013) Model-Driven Engineering of Process Control Software—Beyond Device-Centric Abstractions. *Control Engineering Practice*, **21**, 1078-1096. https://doi.org/10.1016/j.conengprac.2013.03.013

[2] Heck, B.S., Wills, L.M. and Vachtsevanos, G.J. (2009) Software Technology for Implementing Reusable, Distributed Control Systems. In: Valavanis, K.P., Ed., *Applications of Intelligent Control to Engineering Systems*, Springer, Dordrecht, 267-293. https://doi.org/10.1007/978-90-481-3018-4_11

[3] Hayward, A., Daun, M., Petrovska, A., Böhm, W., Krajinski, L. and Fay, A. (2021) Function Modeling for Collaborative Embedded Systems. In: Böhm, W., Broy, M., Klein, C., Pohl, K., Rumpe, B. and Schröck, S., Eds., *Model-Based Engineering of Collaborative Embedded Systems*, Springer, Cham, 71-93. https://doi.org/10.1007/978-3-030-62136-0_4

[4] Zaytoon, J. and Riera, B. (2017) Synthesis and Implementation of Logic Controllers—A Review. *Annual Reviews in Control*, **43**, 152-168. https://doi.org/10.1016/j.arcontrol.2017.03.004

[5] Friedrich, D. and Vogel-Heuser, B. (2007) Benefit of a System Modeling in Automation and Control Education. 2007 *American Control Conference*, New York, 9-13 July 2007, 2497-2502. https://doi.org/10.1109/ACC.2007.4282926

[6] Fay, A. and Schumacher, F. (2014) Formal Representation of GRAFCET to Automatically Generate Control Code. *Control Engineering Practice*, **33**, 84-93. https://doi.org/10.1016/j.conengprac.2014.09.008

[7] Da Silva, A.R. (2015) Model-Driven Engineering: A Survey Supported by the Unified Conceptual Model. *Computer Languages*, *Systems & Structures*, **43**, 139-155. https://doi.org/10.1016/j.cl.2015.06.001

[8] Mohamed, M.A., Challenger, M. and Kardas, G. (2020) Applications of Model-Driven Engineering in Cyber-Physical Systems: A Systematic Mapping Study. *Journal of Computer Languages*, **59**, Article ID: 100972. https://doi.org/10.1016/j.cola.2020.100972

[9] Hästbacka, D., Vepsäläinen, T. and Kuikka, S. (2011) Model-Driven Development of Industrial Process Control Applications. *Journal of Systems and Software*, **84**, 1100-1113. https://doi.org/10.1016/j.jss.2011.01.063

[10] Alvarez, M.L., Sarachaga, I., Burgos, A., Estevez, E. and Marcos, M. (2016) A Methodological Approach to Model-Driven Design and Development of Automation Systems. *IEEE Transactions on Automation Science and Engineering*, **15**, 67-79. https://doi.org/10.1109/TASE.2016.2574644

[11] Sprinkle, J., Mernik, M., Tolvanen, J.P. and Spinellis, D. (2009) What Kinds of Nails Need a Domain-Specific Hammer? *IEEE Software*, **26**, 15-18. https://doi.org/10.1109/MS.2009.92

[12] Taye, M.M. and Ghoul, S. (2023) An Approach towards Goal-Oriented Requirements Ontology: Consistency and Completeness Based Requirements Analysis. *Journal of Software Engineering and Applications*, **16**, 31-49. https://doi.org/10.4236/jsea.2023.162003

[13] Albreshne, A. and Pasquier, J. (2015) A Domain Specific Language for High-Level Process Control Programming in Smart Buildings. *Procedia Computer Science*, **63**, 65-73. https://doi.org/10.1016/j.procs.2015.08.313

[14] Niang, M., Riera, B., Philippot, A., Zaytoon, J., Gellot, F. and Coupat, R. (2020) A

Methodology for Automatic Generation, Formal Verification and Implementation of Safe PLC Programs for Power Supply Equipment of the Electric Lines of Railway Control Systems. *Computers in Industry*, **123**, Article ID: 103328. https://doi.org/10.1016/j.compind.2020.103328

[15] Alaca, O.F., Tezel, B.T., Challenger, M., Goulão, M., Amaral, V. and Kardas, G. (2021) AgentDSM-Eval: A Framework for the Evaluation of Domain-Specific Modeling Languages for Multi-Agent Systems. *Computer Standards & Interfaces*, **76**, Article ID: 103513. https://doi.org/10.1016/j.csi.2021.103513

[16] Godena, G. and Strmčnik, S. (2018) A New State Machine Behaviour Model for Procedural Control Entities in Industrial Process Control Systems. *Journal of Information Technology and Control*, **47**, 419-430. https://doi.org/10.5755/j01.itc.47.3.19630

[17] Harel, D. (1987) Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, **8**, 231-274. https://doi.org/10.1016/0167-6423(87)90035-9

[18] Crane, M.L. and Dingel, J. (2007) UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. *Software and Systems Modeling*, **6**, 415-435. https://doi.org/10.1007/s10270-006-0042-8