Modelling a product based workflow system in CPN tools

Irene Vanderfeesten, Wil van der Aalst, Hajo A. Reijers

Technische Universiteit Eindhoven, Department of Technology Management,
PO Box 513, 5600 MB Eindhoven, The Netherlands
{i.t.p.vanderfeesten, w.m.p.v.d.aalst, h.a.reijers}@tm.tue.nl

Abstract. A new approach to workflow process (re)design is the use of Product Based Workflow Design (PBWD) theory. This theory takes the workflow product as the central concept (instead of the workflow process) and provides a product data model of the structure of information processing in the workflow process. In this paper we introduce a prototype for applying PBWD constructed using CPN Tools. The goal of this prototype is to provide a tool for 'experimentation' with the product data model. The CPN model supports the step-by-step calculation and processing of data in the workflow process based on several strategies and provides insight in the way PBWD works. Finally, the prototype is evaluated with on the one hand a state space analysis to assess the correctness of the model and on the other hand a simulation of a sample product data model.

1 Introduction

Workflow processes are administrative business processes in which much information is processed. Although the focus on the business process instead of a focus on a single department has become more and more popular during the past decade, the (re)design of models for these business processes is still more art than science.
In redesigning workflow processes generally two approaches can be distinguished [10]:

- *Evolutionary approaches*: the existing process is taken as a starting point, which is gradually refined or improved by using a set of best practices or rules to transform that process.
- *Revolutionary approaches*: a clean-sheet of paper is taken to design the complete process from scratch.

Product Based Workflow Design (PBWD, see [1,2,3,15,16,17]) is a revolutionary approach. It takes a different view on the workflow process in which the processing of data and the workflow end product are the central concepts, rather than the activities and the workflow process itself. In PBWD, the workflow product is represented by a product data model, i.e. a network structure of the construction of the product (cf. the Bill-of-Material (BOM) in Manufacturing, [11]). In manufacturing processes the BOM is used to structure the operations process [6,7,19,20]. It seems a promising step to adopt this view in administrative business processes too [12]. Based on the product representation the workflow process model can be derived [17].
Although the "PBWD view" on a workflow process may not be the most popular way people think about designing a workflow process, it has several advantages [15]:

- The clean sheet approach that is taken allows for maximal space to establish performance improvements (*radicalism*). Approaches that use the existing process will to some extent copy constructions from the current process, taking over errors or undesirable constructs.
- Moreover, PBWD is *objective*. In the first place because the product specification is taken as the basis for a workflow design, each recognized information element and each production rule can be justified and verified with this specification. As a consequence there are no unnecessary tasks in the resulting workflow. Secondly, the ordering of (tasks with) production rules themselves is completely driven by the performance targets of the design effort.
- The *analytical* approach of PBWD renders detailed deliverables suitable to use for systems development purposes.

– Finally, the *focus on the product* can help to create consensus between different stakeholders. It gives a clear and objective representation of the workflow product that can help in discussing the process. Because the product data model contains less information than a process model it is easier and less complex to design, understand, and maintain.

At this point in time, there is no tool support for PBWD yet. Because it is a time-consuming and error prone method to come from the product data model to a workflow process model, it is useful to develop tools that support this process. However, the first step towards such a tool is a better understanding of and experimenting with the product data model. The goal of this research is the development of a prototype to get more insight in the product data model. This prototype provides a way to qualitatively compare the sequential unfolding of the product tree based on several strategies for local selection of the next step. The prototype is built in CPN Tools [4,8]. Modelling and analysis with Colored Petri Nets is quite common in a manufacturing context [5,13,14]. There are several reasons for using Colored Petri Nets and CPN Tools:

– First, the use of Petri nets provides correctness analysis of the model by means of, for instance, a state space analysis.
– It is very easy to make small adaptations to the models and then compare the outcomes.
– Moreover, the integration of process and data is essential to this view on a workflow process. CPN Tools provides the possibility to integrate them in one model. The data is then captured in the data structures of the model, i.e. in the colours or types, and the process in the Petri net itself.
– Finally, CPN Tools contains a simulation environment, in which one can go step-by-step through the model, following or defining oneself a sequence of firing.

In this paper we describe the CPN prototype for PBWD. The remainder is organised as follows. First we introduce the concept of PBWD and the product data model. Next, the CPN model is explained. In section four we go into further detail of the selection strategies and explain their differences. This section is followed by an evaluation of the prototype based on a state space analysis and a simulation example. Finally, the paper concludes with a discussion and directions for future research.

2 Product Data Model

PBWD theory is based on a product data model. This product data model describes the most elementary parts of a workflow process. It contains the data and information that is processed in the workflow process. It can be compared to a Bill-of-Material from manufacturing [11]. For example, a car is assembled from an engine and a subassembly. This subassembly consists of a chassis and four wheels (see Figure 1).
Similarly, in an administrative context the balance of a company is determined by its assets and liabilities. Moreover, the assets are based on the fixed assets (such as goodwill, (in)tangible assets) and current assets, i.e. debtors, cash at bank and in hand, of the company and the liabilities can be calculated based on the long-term liabilities (like loans and mortgages) and the short-term liabilities (for example creditors).
The data pieces in the product data model are called *data elements*. On these data elements *operations* are defined (e.g. a calculation on numbers, decision making based on certain information). Operations can be executed automatically or manually or through a combination of automatic and manual execution. They denote the relationship between several data elements. Considering the example of the balance of a company as introduced above, three of the data elements are the balance, the assets and the liabilities. An operation defined on these data elements is then the calculation of the balance of the company, i.e. the output of the operation, out of the assets and liabilities, i.e. the input elements of the operation. For a better understanding of these product data structures we refer to [15,16,17].

The goal of the prototype described in this paper is to provide insight in the way the PBWD method works. With the help of this prototype it is possible to simulate the calculation of new information elements based on the elements that are already known.
For implementation of the CPN prototype the product data model needs to be formalized. This formalization is discussed below.

2.1 Formalization of product data model

Taking a mathematical view, a product data model is a set of data elements and 'directed' links or relations between them. The data elements of the product data model are denoted by an identifier (e.g. a). An operation is represented by a tuple $(a, [b, c, d])$. The first element of the tuple (a) is the output data element of the operation, while the second element is a list of input data elements $([b, c, d])$. Note that an operation can have several input elements, but only one output element. Next, this tuple is extended with some other information: $(\text{“}opID\text{”}, a, [b, c, d], attributes)$. “$opID$” is a unique identifier of an operation and *attributes* is a tuple containing a value for duration and costs of the operation $(attributes = (duration, cost))$. An operation can be executed when the values for all input information elements are available and when the values of these input elements satisfy certain conditions (if declared).

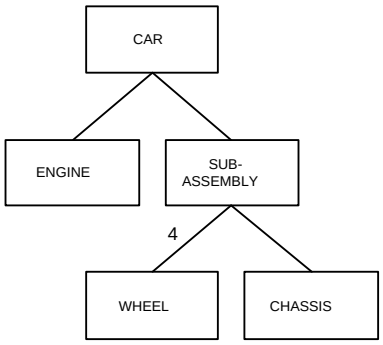

Fig. 1. The bill-of-material of a car

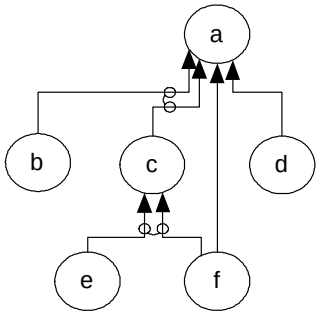

Fig. 2. The product data model of suitability to become a helicopter pilot

This formalization will be clarified with an example (from [15]). In Figure 2, the procedure to determine one's suitability to become a helicopter pilot is denoted in terms of a product data model. All nodes in the figure are data elements that can be used to decide whether an applicant is suitable to become a helicopter pilot or not. The meaning of these data elements (denoted by a, b, c, d, e, and f) is:

a = Suitability to become a helicopter pilot.
b = Psychological fitness.
c = Physical fitness.
d = Latest result of suitability test in the previous two years.
e = Quality of reflexes.
f = Quality of eye-sight.

The suitability to become a helicopter pilot can be determined in either of three ways:

1. Combine the results of the psychological test and physical test. (In Figure 2 this is: e, f combined to c and b, c combined to a)
2. Use the result of a previous suitability test. $(d \rightarrow a)$
3. Make a decision based on the candidate's quality of eye-sight. $(f \rightarrow a)$

These different ways to determine a candidate's suitability may be applicable under different conditions. It can be imagined that if a pilot's eye-sight is bad, then this directly gives a result that the candidate is not suitable. However, in a more common case, the eye-sight quality is one of the many aspects that are incorporated in a physical test, which should be combined with the outcome of the psychological test to determine the suitability result. Also, not for each candidate that applies, a previous test result is available. But if there is one of a recent date, it can be used directly, without knowing the values for the other information elements.

The description of this situation can be formalized in a product data model as follows:

("$Op1$", a, $[b, c]$, $(0, 0)$)
("$Op2$", a, $[d]$, $(0, 0)$)
("$Op3$", a, $[f]$, $(0, 0)$)
("$Op4$", c, $[e, f]$, $(0, 0)$)

Given this product data model and a set of available data element values, it is possible to determine which operations could be executed and thus which new data element values could be produced. For instance, looking at the example of the helicopter pilot, when the values for data element e (quality of reflexes) and f (quality of eye-sight) are available, the value for element c (physical fitness) can be determined, i.e. the execution of "$Op4$". Initially, the values of elements with no ingoing arcs are available (in our example that is b, d, e, f).
Moreover, some constraints for execution can be added to the operations. Consider for instance operation number three ("$Op3$") from the helicopter pilot example. This operation can only be executed when the value of data element f (quality of eye-sight) is 'bad'. It should not be executed when the value of f is otherwise. The constraints on the execution of an operation are in this prototype added as a function that checks whether an operation satisfies the condition when all of its input elements are available (see appendix A.3).

The formalization of the product data model as presented in this section will be used in the prototype. The CPN-model of this prototype is explained in detail in the next section.

3 CPN prototype

The CPN model we developed consists of two levels: the main level and the sublevel. In this section we first explain the main level and afterwards we elaborate on different variants modelling the sublevel.

3.1 Main level

The main level of the CPN model is depicted in Figure 3. It contains eight places and three transitions. The main stream in the model is indicated by thick lines. Initially, places *available data elements*, *not yet executable operations*, *ready for calculation*, *executable operations*, *total cost* and *total duration* contain a token.
The token in *ready for calculation* does not contain real information. It is only needed to ensure transition *calculate* only fires when it is allowed to, i.e. when the operation that was selected in the previous execution of the main stream has been executed.
However, the tokens in the other places do contain information in order to parameterize the model. The place *available data elements* contains a list of data elements that are available with their corresponding value. The elements of the list change over time. Initially, this place holds the data elements that initially were available and are not obtained through the execution of an operation in this process, i.e. the elements with no ingoing arcs. In many cases these data elements will be the elements that are provided by the customer that starts the case in the workflow process.
The place *not yet executable operations* holds a list of operations that can not be executed yet. Initially, this list contains all operations in the product data model. The format of an operation

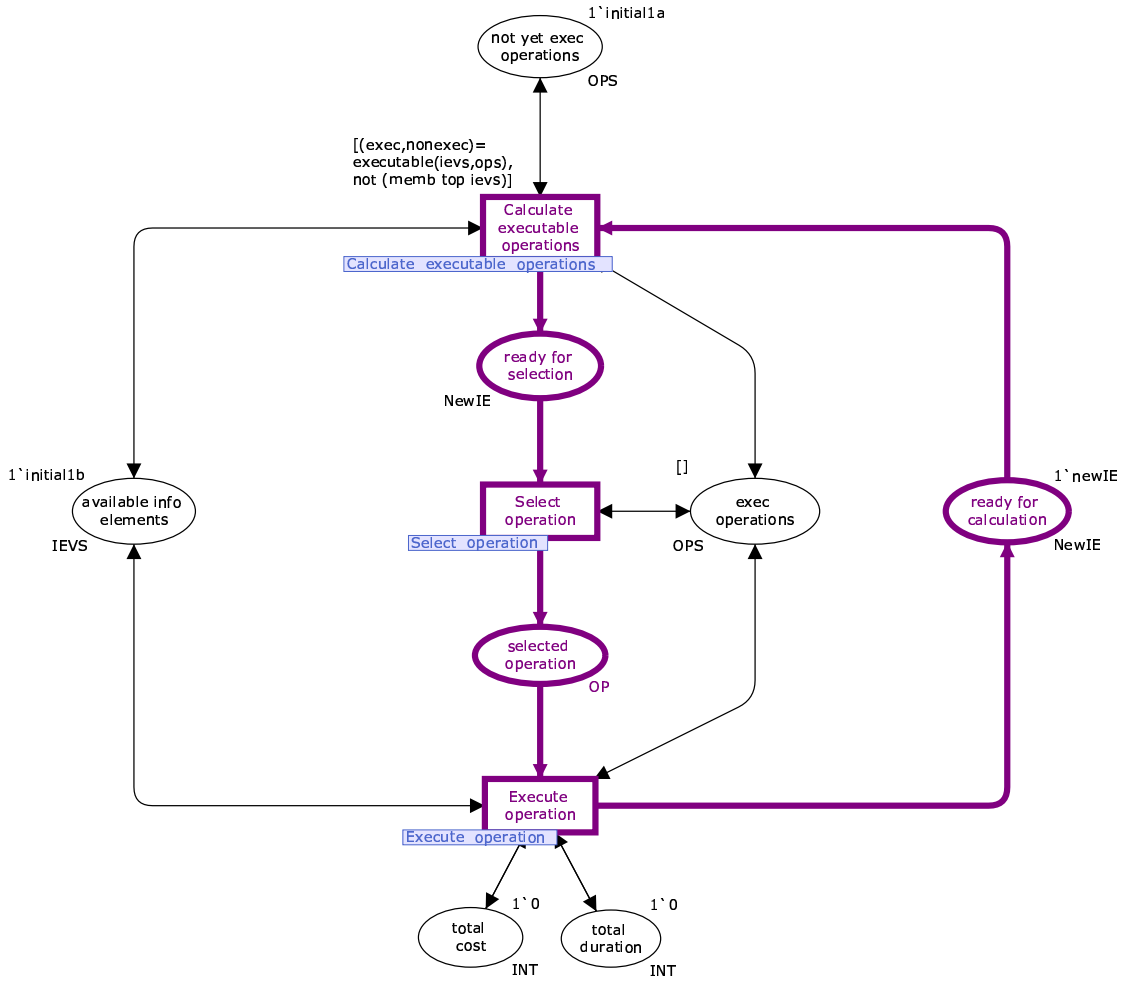

Fig. 3. The main level of the CPN model.

is as defined above in section 2.1. Similarly, *executable operations* contains the list of operations that are executable. Initially, this is an empty list, because the executable operations first have to be calculated.
Finally, *total cost* and *total duration* contain an integer value that denotes the total cost and total duration of the process. For every execution of an operation these values are updated and the cost and duration attributes of that operation are added. Clearly, the initial value of these variables is zero.
The CPN model also shows places that initially do not contain tokens: *ready for selection* and *selected operation*. The first one has a similar meaning of the *ready for calculation* place and makes sure that the *select operation* activity can only fire when there has been a recalculation of the executable operations. Place *selected operation* is used to store operations that are selected. Clearly, this place can not contain a token initially, because the executable operations first have to be calculated before one of them can be selected.

The first step in the execution of this prototype is the firing of transition *calculate executable operations*. When it fires, the operations that are executable, i.e. all of its input elements are in the list of available data elements, are determined from the list of *not yet executable operations*. The executable operations are stored in the place *executable operations*. Next, one of the executable operations is selected and finally this operation is executed, i.e. the output element is added to the list of available data elements and the value for this output element is determined. Then the flow starts again: the executable operations can be calculated based on new information, an operation can be selected, etcetera, until the end product of the process is reached. More specifically this means that the top data element is in the list of available data elements.

In the next sections the subpages for the transitions *calculate executable operations*, *select operation*, and *execute operation* are clarified.

3.2 Calculation of executable operations

As explained before, the *calculate executable operations* transition determines which of the operations from the list of *not yet executable operations* become executable based on the list of available data elements. This calculation turns out to be a bit more involved than one may expect.
As is shown in Figure 4, the *calculate* transition takes the list of available data elements and the list of not yet executable operations as inputs. For each operation in the list it determines to which output place it should be sent.
When the condition of an operation is not satisfied, i.e. if the value of one of the input elements is not satisfying a pre-defined condition, the operation should not become executable and is sent to place *condition not satisfied*. After that, the operation can never become executable again. We assume that conditions remain stable during the process.
If the output element of the operation is already in the list of available data elements then the operation is put in the place *data element already known* and again it stays there. There is no need to determine the value again. When one or more of the required input data elements are not available yet, i.e. a required item is not in the list of available data elements, the operation is put back in *not yet executable operations*.
Finally, the operations which satisfy the predefined condition on the input data element values, of which the output element is not already known, and of which all input elements occur in the list of available data elements, are sent to the *executable operations* place.
Note that *calculate* also recalculates the operations that where put in the place *executable operations* in one of the earlier iterations, because they can become superfluous (e.g. already determined or not satisfying the condition after the calculation of the value of one of the input elements).
Besides that, if the end product is determined, it makes no sense to still perform some operations that are available. Therefore, a guard is added to *calculate executable operations* which only allows this transition to fire when the end product ('top') is not yet in the list of available data elements.

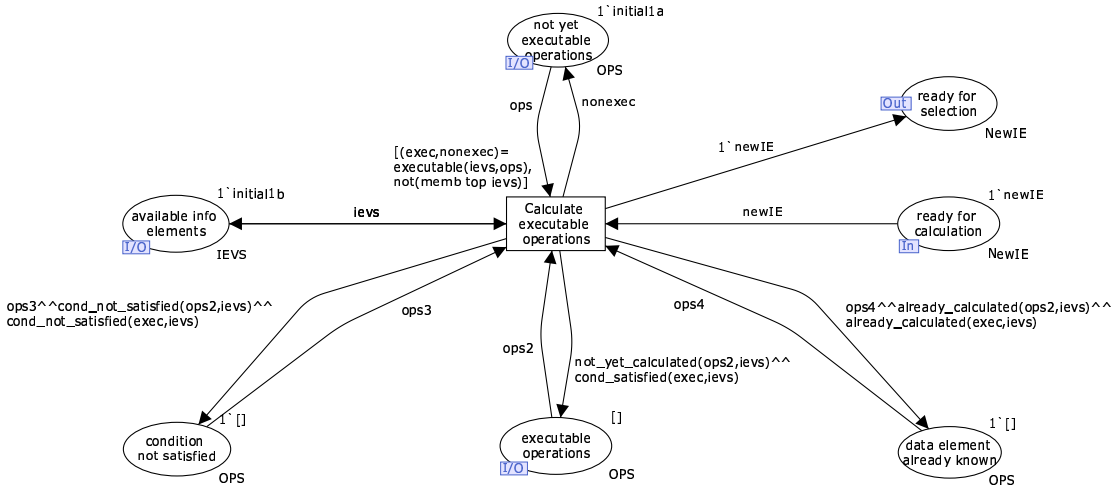

Fig. 4. The calculate executable operations subpage

3.3 Selection of operation

When the executable operations are determined by the *calculate* transition (there is a token available in *ready for selection* and *executable operations* contains a list of operations), one of these operations can be selected for execution. Several strategies can be used to determine which operation from the list of executable operations should be selected. The strategies we considered are: (a) the first operation of the list, (b) the operation with the lowest cost, (c) the operation with the shortest duration, (d) random selection, and (e) selection by a user.
The way in which the next operation to be executed is determined influences the performance of the process of making the workflow end product. We need performance measures to compare the selection strategies in actual situations. To assess the performance of a certain strategy we limit ourselves to total cost and total duration, although one could think of other performance indicators. The selection strategies are further explained in Section 4, and are applied to an example in Section 5.

3.4 Execution of operation

After selecting an operation for execution, this operation will be executed in two steps (see Figure 5). First the execution will be started and the total cost and duration are updated, while the execution of this operation takes some time (the time delay is determined by the attribute *duration*). Next, the execution is ended, either successfully or unsuccessfully. When the operation is performed successfully the output data element of the operation is put in the list of available data elements together with its newly determined value (the simple function *calculation* determines this new value, see Appendix A.3).
When the execution of the operation fails, the operation is put back at the end of the list of *executable operations*, from where it can be selected again some time. The *end execution* transition also puts back a token in *ready for calculation*, so that the loop can start again by calculating the executable operations for the next possible step.

4 Selection strategies

Once the executable operations have been determined, one of them can be selected for execution. The selection in a way influences the costs and duration as explained before. In this section we

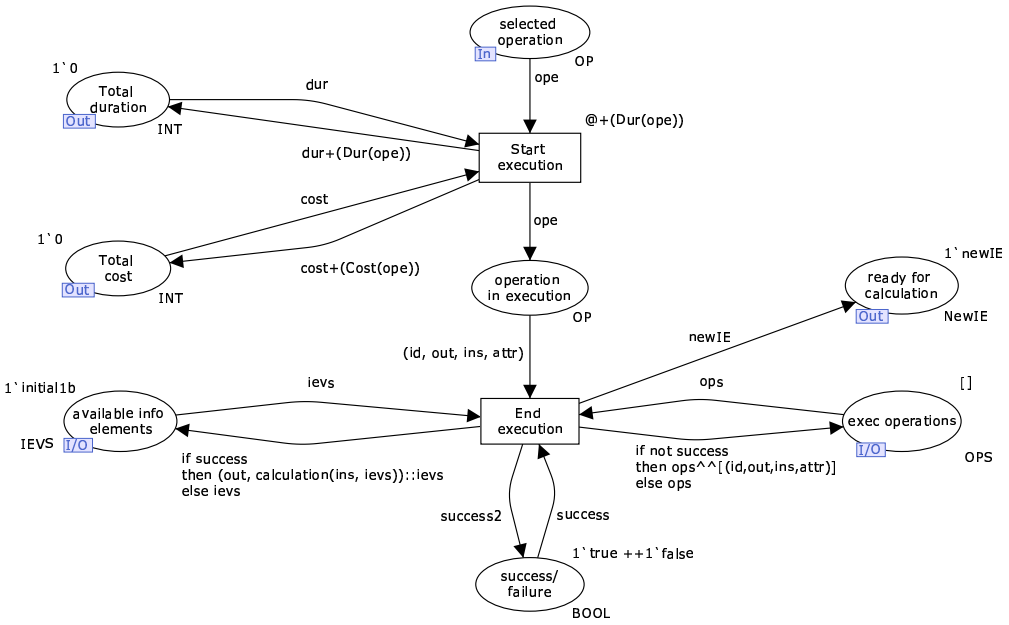

Fig. 5. The execute operation subpage

elaborate on several strategies to select the operation to be executed. We have identified five different strategies that are described separately. We start with a basic strategy: the selection of the first element from the list. Next, for every other strategy it is shown which adaptations to the first model have to be made in order to get a model that supports this strategy.

4.1 Select first element from list

To select the first element from the list with executable operations it is not necessary to write a complex function in CPN Tools[1]. As is shown in the subpage of *select operation* in Figure 6 this element can be specified by explicitly distinguishing the head from the rest of the list: *ope* :: *ops*, where *ope* is declared as an operation (type OP, see declarations in appendix A.1) and *ops* as a list of operations (type OPS). This first element is then taken from the list and passed on to the place *selected operation*, while the tail of the list of operations is put back in the place *executable operations*.
Clearly, the order of elements in the list determines which element is the first element of the list. The list can have an arbitrary order (like a lexicographical order or an order based on one of the attributes). The order of the list can be defined by the user by means of the declaration of the list containing all operations. In this way the selection strategy can be influenced. In our example we use a list order based on the ID of the operation, i.e. "$Op1$" is before "$Op2$", etcetera.

4.2 Select element with lowest cost

The selection of the element from the list with the lowest cost is slightly more complex. The basic structure of the select first element subpage can be maintained but the arc inscriptions have to be changed as shown in Figure 7. The whole list is taken as an input to the transition. Next, the element with the lowest value for a certain attribute is determined by a function called *SelMin*. To this function we add the attribute that we want to have a minimal value, that is *Cost*.

[1] Note that CPN Tools uses the functional programming language Standard ML [18] to manipulate token colors

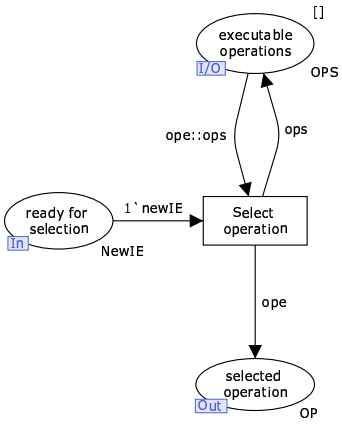

Fig. 6. The select first element from the list subpage

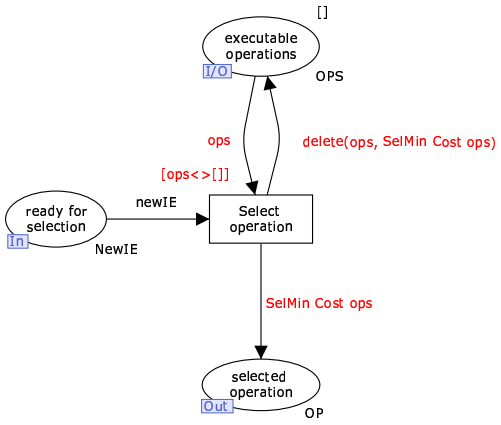

Fig. 7. The select element with lowest cost subpage

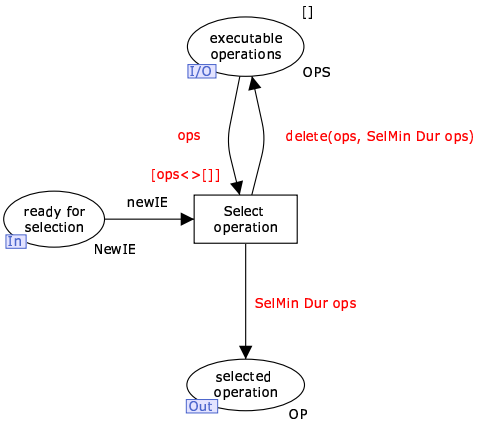

Fig. 8. The select element with minimal duration subpage

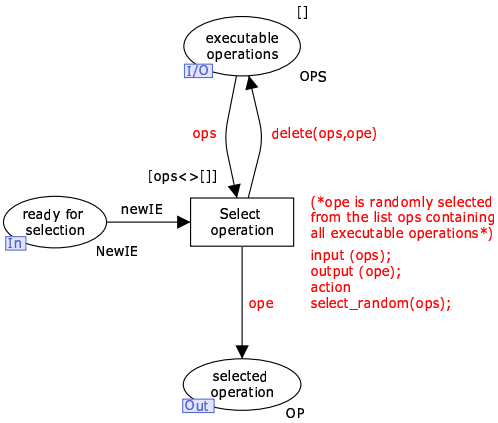

Fig. 9. The random selection subpage

The *SelMin Cost* function is applied to the list of executable operations *ops*. The operation with the lowest cost is sent to *selected operation* and is deleted from the list of executable operations. Of course this only works for a non-empty list in *executable operations*. Therefore a guard is added to the transition.

4.3 Select element with shortest duration

Similar to the selection of the element with the lowest cost we can select the element with the shortest duration (Figure 8). In this case the structure of the model is the same, only the *SelMin* function is applied to the list of executable operations considering the value of attribute *Dur* (Duration) instead of *Cost*.

4.4 Random selection

Another way to select an operation from the list of operations is through random selection. Basically, we can keep the structure of the model as it was in the previous three strategies. To select an

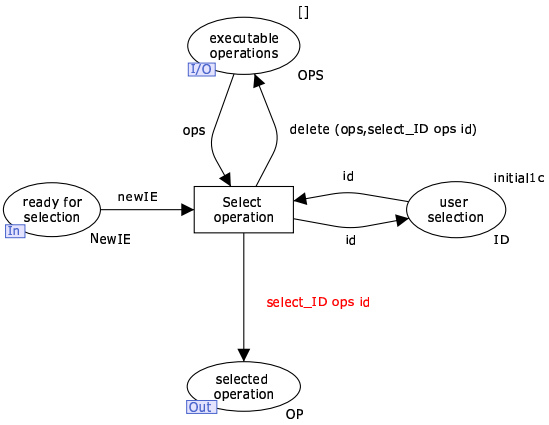

Fig. 10. The selection by user subpage

arbitrary element from the list, we need a function that transforms a real number from a random number generator into an integer as the number of an element from the list.
Because the random function returns a different value every time it is evaluated we can not use the function as we used the *SelMin* function before. Therefore, the random function is put in the action part of the transition. In this way the random number is bound to a variable *ope*. Through this variable the element can be selected and deleted from the list. This function is shown in Figure 9 and is explained in more detail in Appendix A.3.

4.5 User selection

The last strategy for selecting an operation is to let the user decide which operation is selected. In this way, of course, any strategy can be 'imitated' through the actions of a user (e.g. when the user selects the first element of the list every time, the strategy is equal to the select first element of list strategy).
Most of the structure of the model can be reused. An extra input place *user selection* with colour *ID* is added. By changing the current marking of that place a user can specify which element from the list, denoted by its ID, he or she wants to take. This ID is an input to the transition and is used on the outgoing arcs to specify the element from the list as shown in Figure 10.
Moreover, when CPN Tools version 1.4.0 is used it is possible to select a binding for the next step from all possible binding elements (see Figure 11). Initially, all *opID*-s are present in the place user selection, although not all operations may be present in the list of executable operations. Therefore, the user should pay attention to select an operation that really is executable because it is possible to bind an *opID* that does not exist in the list. When an operation is chosen that is not in the list, the 'Empty' exception is raised and the simulation is stopped. However, we feel this is a nicer solution than manually changing a marking. Obviously, this is not a very sophisticated solution from a GUI perspective, but it serves our purpose.

5 Evaluation

In the previous two sections, the prototype for experimenting with a product driven workflow system is explained. In this section the model is assessed. As explained in the introduction, CPN Tools provides several ways to evaluate the correctness of a model. The first one is by means of a state space analysis, which is elaborated on in Section 5.1. In the second part of this section, simulation experiments are conducted to compare several selection strategies with each other. Section 5.2 describes the simulation.

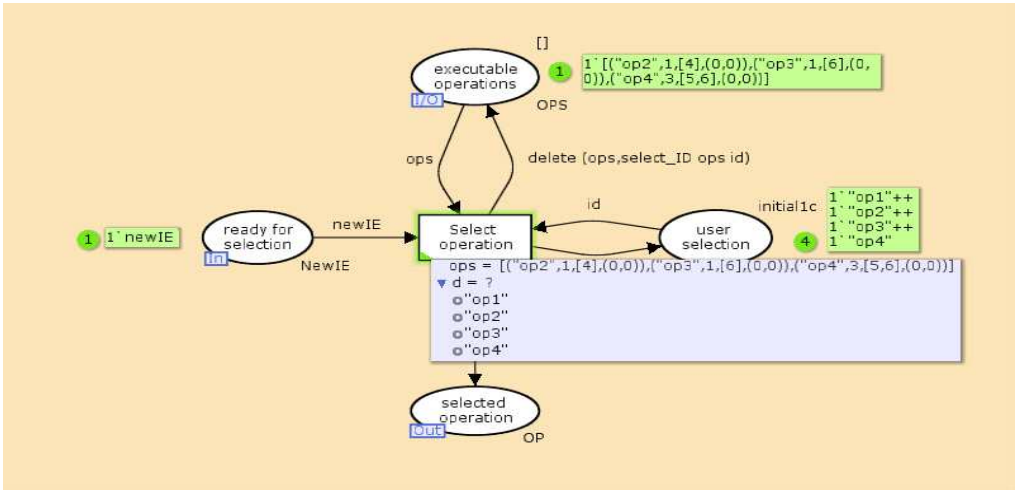

Fig. 11. Selecting a binding element in CPN Tools 1.4.0

5.1 State space analysis

A state space analysis is an analysis of all possible occurrence paths in the model. Based on the occurrence graph, containing all occurrence paths, a lot of information on the properties of the model (and thus on the correctness of the model) can be gained.
To limit the calculation time we have conducted a state space analysis of the model based on a small example product data model. The Helicopter Pilot example only contains six data elements and four operations, as was shown in Section 2.1. We have only reported on the state space for the random selection strategy[2] because it covers all the other strategies. CPN Tools calculated a full state space, containing 157 nodes and 241 arcs, in less than one second. Below, the properties of the net are discussed based on the generated state space report.

Boundedness properties - The first property that is derived from the state space analysis is the upper and lower bound for all places of the model. Upper and lower bounds indicate how many and how few token elements a place can contain, e.g., how many tokens we can have of a particular color on a particular place instance [8].
The integer upper and lower bounds give an integer number of tokens that are at most, respectively at least, present in the place. Multi-set upper and lower bounds in addition give information on the value of the token(s) in the place.
In Table 1 and Table 2, the integer upper and lower bounds and the multi-set upper and lower bounds of the model are given respectively. For the sake of clarity, the operations in the multi-set bounds are represented by an abbreviation: $(\text{“}op1\text{”}, a, [b, c], (0, 0))$ is denoted by “$op1$”.
To give an example of an upper multi-set bound we focus on the place *selected operation*. The upper bound of this place is the multi-set of all operations in our example, i.e. $1\text{`}(\text{“}op1\text{”}) + +1\text{`}(\text{“}op2\text{”}) + +1\text{`}(\text{“}op3\text{”}) + +1\text{`}(\text{“}op4\text{”})$. Because of the random selection strategy all operations can be selected from the list, i.e. for every operation there is a path in the occurrence graph in which the operation appears in the place selected operation. However, there can never be more than one operation in the place at the same time.
The tables show that all places have an upper and lower bound. Thus, the net is bounded. This makes sense because the Helicopter Pilot product data model is represented by a finite list of operations, from which the elements are taken one by one.

[2] To avoid problems with calculating the full state space, the ‘random’ function is replaced by a variable that denotes the range of the list of operations. Through this variable a random selection can still be made from the list.

Table 1. Integer Bounds Helicopter Pilot Model

Place	Upper Integer Bound	Lower Integer Bound
Available info elements	1	1
Executable operations	1	1
Not yet executable operations	1	1
Ready for calculation	1	0
Ready for selection	1	0
Selected operation	1	0
Total cost	1	1
Total duration	1	1
Condition not satisfied	1	1
Data element already known	1	1
Operation in execution	1	0
Success	2	2

Table 2. Multi-set Bounds Helicopter Pilot Model

Place	Upper Multi-set Bound	Lower Multi-set Bound
Available info elements	1‘[(1,0),(2,0),(4,0),(5,0),(6,0)]++ 1‘[(1,0),(3,0),(2,0),(4,0),(5,0),(6,0)]++ 1‘[(2,0),(4,0),(5,0),(6,0)]++ 1‘[(3,0),(2,0),(4,0),(5,0),(6,0)]	empty
Executable operations	1‘[]++1‘[(“op1”),(“op2”)]++ 1‘[(“op1”),(“op2”),(“op3”)]++ 1‘[(“op1”),(“op3”)]++ 1‘[(“op1”),(“op3”),(“op2”)]++ 1‘[(“op2”),(“op1”)]++ 1‘[(“op2”),(“op1”),(“op3”)]++ 1‘[(“op2”),(“op3”)]++ 1‘[(“op2”),(“op3”),(“op1”)]++ 1‘[(“op2”),(“op3”),(“op4”)]++ 1‘[(“op2”),(“op4”)]++ 1‘[(“op3”),(“op1”)]++ 1‘[(“op3”),(“op1”),(“op2”)]++ 1‘[(“op3”),(“op2”)]++ 1‘[(“op3”),(“op2”),(“op1”)]++ 1‘[(“op3”),(“op4”)]++ 1‘[(“op2”),(“op4”),(“op3”)]++ 1‘[(“op3”),(“op2”),(“op4”)]++ 1‘[(“op3”),(“op4”),(“op2”)]++ 1‘[(“op4”),(“op2”),(“op3”)]++ 1‘[(“op4”),(“op3”),(“op2”)]++ 1‘[(“op4”),(“op2”)]++ 1‘[(“op4”),(“op3”)]	empty
Not yet executable operations	1‘[]++1‘[(“op1”)]++ 1‘[(“op1”),(“op2”),(“op3”),(“op4”)]	empty
Ready for calculation	1‘newIE	empty
Ready for selection	1‘newIE	empty
Selected operation	1‘(“op1”)++1‘(“op2”)++ 1‘(“op3”)++1‘(“op4”)	empty
Total cost	1‘0	1‘0
Total duration	1‘0	1‘0
Condition not satisfied	1‘[]	1‘[]
Data element already known	1‘[]	1‘[]
Operation in execution	1‘(“op1”)++1‘(“op2”)++ 1‘(“op3”)++1‘(“op4”)	empty
Success	2‘false++2‘true	empty

Home properties - The existence of home markings in the net is the next property of colored Petri nets that we discuss. A home marking is a marking to which it is always possible to return to [8].
In our prototype we consider the fulfilment of the product tree. This is a step-by-step development of the end product, through the 'assembly' of subproducts. In this case, it is not desirable to return to a previous state, i.e. undo an assembly. Thus, the net should have no home marking(s) and the state space analysis shows that is indeed the case.

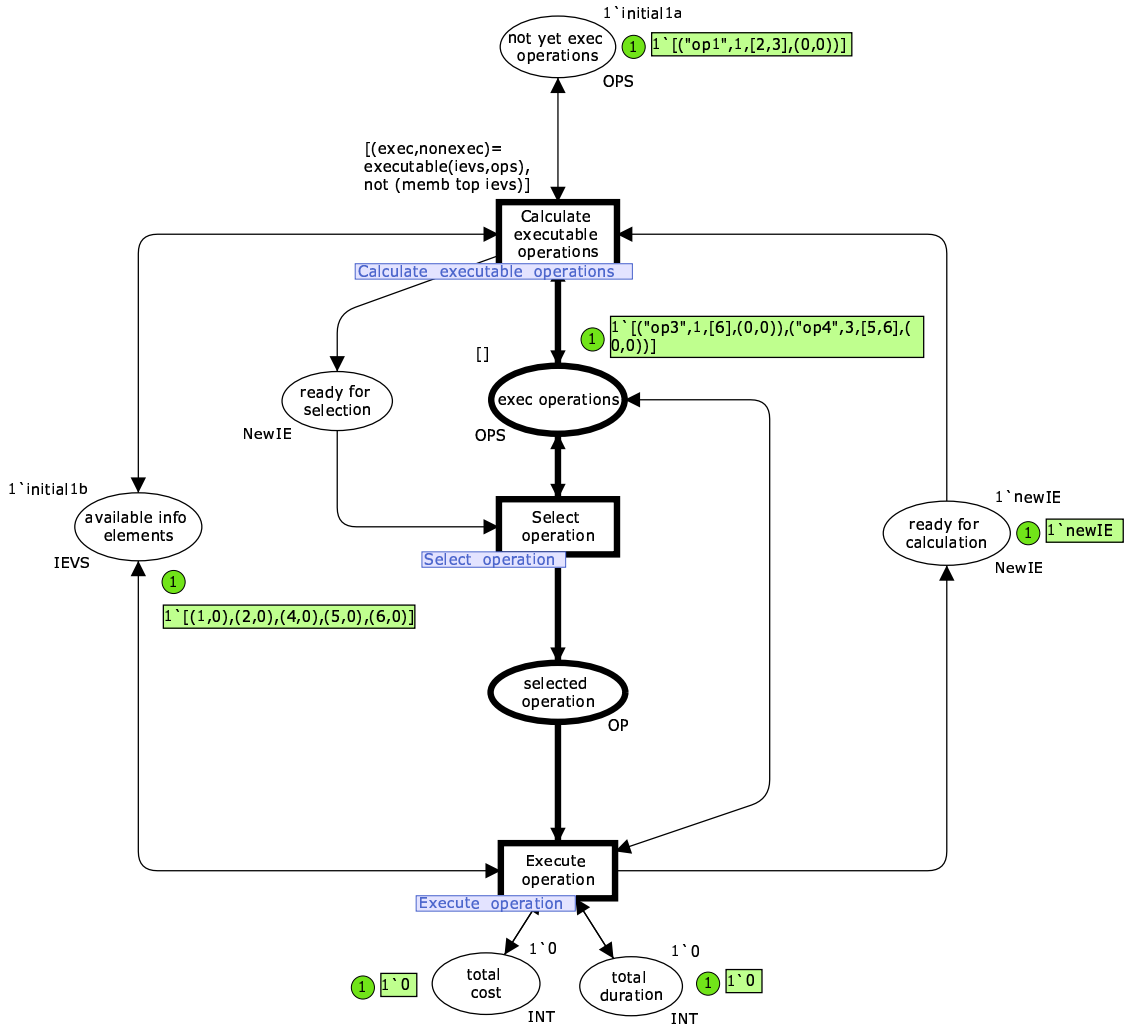

Fig. 12. State '9' from the state space is a dead marking.

Liveness properties - Liveness denotes that a set of binding elements remains active, i.e. every transition can become enabled by firing an arbitrary number of transitions [8]. In contrast, a dead marking is a state of the net in which no transition is enabled.
Our model contains thirty dead markings according to the state space report. These dead markings are caused by the guard for transition *calculate executable operations*. The guard *not (memb top ievs)* enforces that when the end product is produced, nothing else is allowed to happen. In this

case, the tokens in places *not yet executable operations* and *executable operations* can not be removed and form a dead marking.
The state space reports the states of the state space that are dead markings. For instance, state '9' from the state space is a dead marking. By using the 'SStoSim' tool in CPNTools we can visualize this state as is shown in Figure 12. This state indeed is a valid dead marking. The end product (data element 1) is determined by executing the operation with ID *"op2"* starting from the initial marking. Nothing else can happen in the net and there are still operations left in places *not yet executable operations* ("*op*1") and *executable operations* ("*op*3", "*op*4").
Furthermore, the net contains neither dead transition instances, i.e. a transition that never becomes enabled, nor live transition instances (i.e. a transition that will always get enabled again after firing).

Fairness properties - Finally, we look at the fairness of this net. Fairness indicates how often different binding elements occur [8]. The state space analysis report shows that all transitions in the model are impartial. An impartial transition will fire infinitely often given an arbitrary infinite occurrence sequence. This means that any infinite occurrence sequence in our model will contain all transitions infinitely often.
This behavior can be explained by the sequential order of the transitions and the possibility of failure of the execution of an operation. When the execution of an operation has failed, the loop of calculation, selection, and execution has to be started over again. An operation can fail infinitely often (although this is not likely). Therefore infinite occurrence sequences exist in the occurrence graph. Since all transitions have to occur in a predefined sequential order before the operation can be executed again all transitions in the model will occur infinitely often in an arbitrary infinite occurrence sequence.

5.2 Comparison of strategies

The correctness of the model has been shown using state space analysis. Now, we will illustrate the correct working and results of our model based on a real-life example. The example is taken from [15,16] and describes the process of a request for social benefits within a Dutch social security administration office ('GAK'). This agency implements the social security legislation in the Netherlands. On a daily basis, it handles large amounts of requests for unemployment benefits and occupational disability allowances. Social security laws as well as contracts with employer organizations impose restrictions on the way these requests are handled. To decide on unemployment benefits much information is processed. Figure 13 shows the product data model for this case. The exact meaning of all data elements can be found in the original study [15].
The product data structure of the unemployment benefits case consists of 42 data elements and 33 operations in total. Initially, 18 data elements are available. In [15], the costs for each operation are given. We have copied them and multiplied them by ten to avoid rounding problems in the CPN simulation. However, the duration of operations was not indicated in this case study. Therefore, we have added realistic values for the duration of every operation (see the declaration of the list of operations in Appendix A.2).

To be able to compare the different strategies of selecting executable operations in this case study, we have executed several simulation runs for each strategy. A run corresponds to one complete unfolding of the product tree until the end product is produced and thus gives us one sample of the total cost and total duration of the process. The data on cost and duration are stored for each sample. Per strategy we collected 60 samples ($n = 60$) and constructed 90%-confidence intervals, according to the formula for a t-distribution[3] [9].
In Figure 14 and 15 the confidence intervals are represented. These figures show how much the value for total cost and total duration may vary given a certain selection strategy. Because most

[3] We assume that the population has a normal distribution.

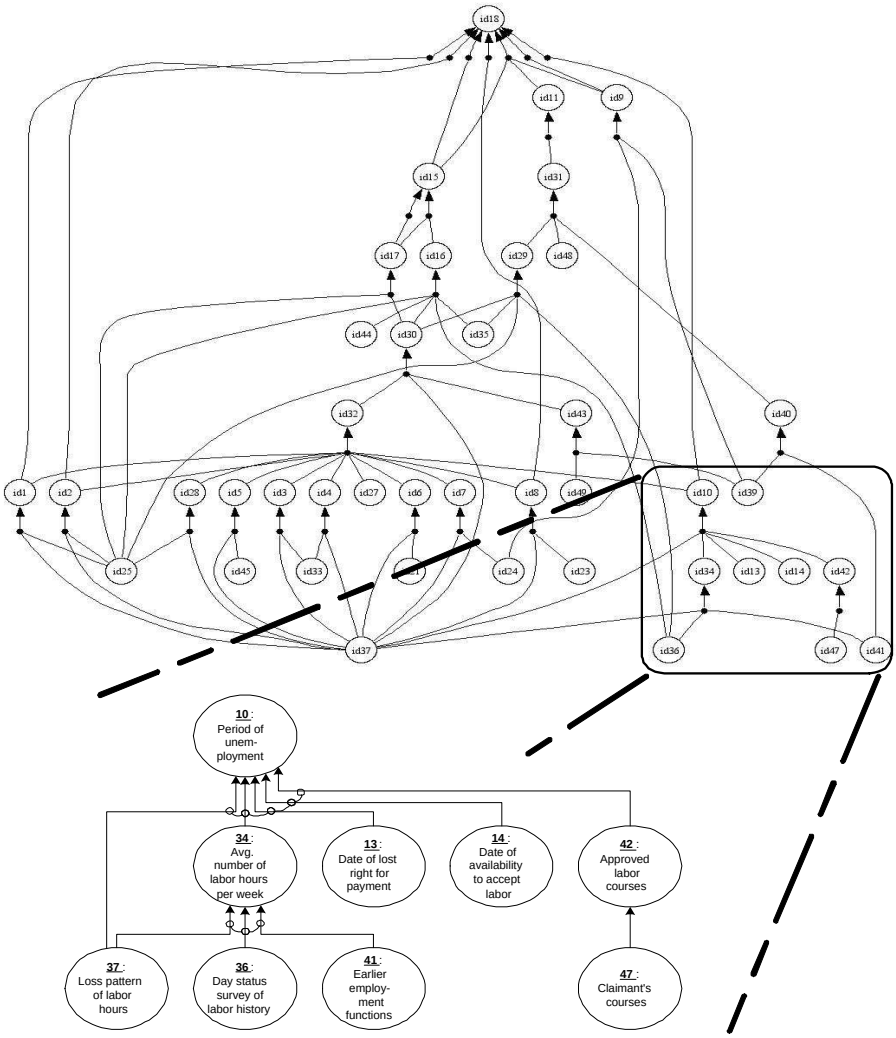

Fig. 13. Product data model for the social benefits case

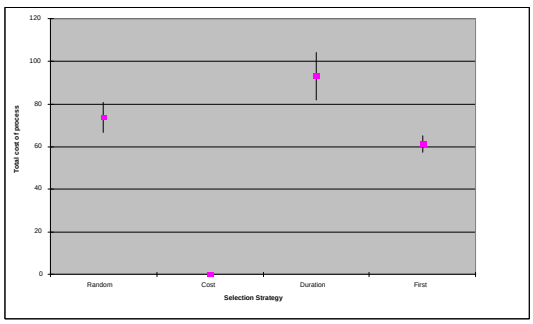

Fig. 14. Confidence intervals for total cost

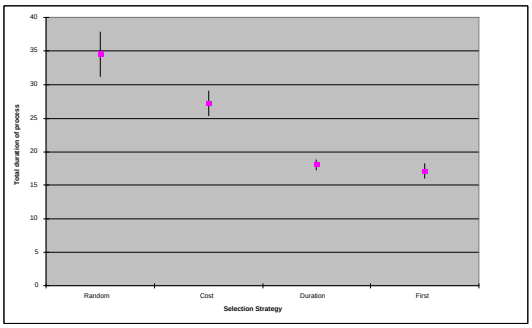

Fig. 15. Confidence intervals for total duration

of the confidence intervals do not overlap it is possible to draw reliable conclusions from the constructed confidence intervals. Some examples of such quantitative conclusions and some qualitative conclusions as well are:

- The total costs are, for example, minimal in case of a 'minimal cost' strategy, followed by a 'first element from list' strategy and a 'random selection' strategy. The 'minimal duration' strategy gives the highest total costs with high variation, i.e. a broad confidence interval, compared to the other strategies.
- For total duration under different strategies the confidence intervals are less explicit: The confidence intervals of the strategies of 'minimal duration' and 'first element from list' selection do overlap. The figures clearly show that a 'minimal cost' strategy leads to a shorter total duration than a random strategy does. Next, the selection of 'minimal duration' and 'first element from list' lead to a shorter total duration than in case of a 'minimal cost' strategy.
- The total cost confidence interval under a 'minimal cost' selection is only one point (see Figure 14). This can be explained by the nature of the example we are considering. The example contains many operations that can be executed automatically and which therefore have cost 0. It is possible to get to the end product by only selecting operations with zero costs. Therefore, by using a 'minimal cost' selection strategy only operations with no costs are selected and we can get to total costs of zero in each sample. This leads to a confidence interval of only one point.
- Confidence intervals for selection of the 'first element from the list' are small. This makes sense because in this case every sample starts with (almost) the same sequence of execution. The only factor that can influence this order is the failure of an execution of an operation which leads to the 'skip' of one operation execution and an extra selection in the end.

The figures show there is no optimal strategy under both optimization criteria. Which strategy is best therefore depends on the importance of each of the criteria. If costs are important, then the strategy of selecting operations with minimal costs is best in our example. If flow times are important, then a minimal duration or first element from list strategy would be best. However, this simulation example gives a clear idea about the information that can be retrieved using our prototype.

6 Conclusion

In this paper we have presented a prototype, built in CPN Tools, for modelling a product based workflow system. This prototype shows how a workflow process can be derived from a product data model (cf. Bill-of-Material). The core of the prototype is formed by the selection strategy. We have shown several strategies to select the next step that has to be performed in the process, from all possible next steps. Note that these strategies only focus on local optimization of the process. They only consider the operations with, for instance, the lowest cost or shortest duration that are executable at that point in time. In the end the selection of the local optimal operation can lead to a suboptimal strategy in terms of overall costs or duration. The prototype provides a way to qualitatively, and to some extend quantitatively, evaluate these different strategies as is shown by a real-life example.
The prototype we developed is a generic tool to play with product data models of different cases. To do so only the initial marking has to be changed by providing different declarations for *top*, *initial1a*, *initial1b*, *initial1c* and the function *check*.

The main goal of this research was to provide more insight in PBWD and the product data model. During the process of the prototype development we have made some design choices that gave us a better understanding of the concepts and issues in this area. The first choice we made is to represent the operations of the product data model by a list of operations instead of a number of separate tokens. Although the latter would perhaps be the most intuitive way, this raised some problems implementing the selection strategies. In a CPN model it is not possible to check all

tokens present in a place and then select the one satisfying a certain condition. The only solution to this problem was to put the operations in a list. This issue made us think about operations, their representation and their relations.
Another choice we made is the strict sequential execution of operations. This point can also be seen as a limitation to the prototype. It would be more realistic if another operation could already be calculated, selected and started while the previous operation is still in execution, especially when execution delays are long. To implement this, extra constructs are needed to make the model work. For instance, it should be possible to break off an already started execution to avoid duplicate determinations of one data element. Moreover, one has to make sure *calculate executable operations* is fired every time a new data element is added to the list of *available data elements*. For the sake of understandability and readability we have not incorporated this possibility in the prototype, because it adds much more complexity to the model. However, this issue made us think of how to deal with parallel execution and other concepts related to the execution of a product data model.

Furthermore, the user interface of the prototype could be improved by the automatic reading of a file containing declarations and other information, and a nicer way for user selection. Nevertheless, by this simple prototype we think we have reached our goal for a better understanding of the product data model and PBWD theory.

Acknowledgements

This research is supported by the Technology Foundation STW, applied science division of NWO and the technology programme of the Dutch Ministry of Economic Affairs.

References

1. W.M.P. van der Aalst. Designing workflows based on product structures. In: K. Li, S. Olariu, Y. Pan, I. Stojmenovic (eds.), Proceedings of the ninth IASTED International Conference on Parallel and Distributed Computing Systems, IASTED/Acta press, Anaheim, pp. 337-342, 1997.
2. W.M.P. van der Aalst. On the automatic generation of workflow processes based on product structures. Computers in Industry, 39, pp. 97-111, 1999.
3. W.M.P. van der Aalst, H.A. Reijers, S.Limam. Product-based workflow design. In: W. Shen, Z. Lin, J.P. Barthes, M. Kamel (eds.). Proceedings of the sixth international conference on CSCW in design. Ottawa: Research Press, pp. 397-402, 2001.
4. CPN Tools home page and manual at http://wiki.daimi.au.dk/cpntools/cpntools.wiki. University of Aarhus, Denmark, 2005.
5. A.A. Desrochers, R.Y. Al-Jaar. Applications of Petri Nets in Manufacturing Systems: Modeling, Control, and Performance Analysis. IEEE Press, 1995.
6. F. Erens, A. MacKay, R. Sulonen. Product modelling using multiple levels of abstraction - instances and types. Computers in Industry, 24 (1), pp. 17-28, 1994.
7. H.M.H. Hegge. Intelligent product family descriptions for business applications. PhD Thesis, Eindhoven University of Technology, Eindhoven, 1995.
8. K. Jensen. Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use, Volume 1, 2 and 3. Springer-Verlag, 1992.
9. D.C. Montgomery, G.C. Runger. Applied Statistics and Probability for Engineers. John Wiley & sons, 2nd edition, 1999.
10. M. Netjes, I.T.P. Vanderfeesten, H.A. Reijers. "Intelligent" software tools for workflow process redesign: a research agenda. Proceedings of the first international workshop on Business Process Design (BPD'05), Nancy, September 2005 (*accepted*).
11. A. Orlicky. Structuring the Bill of Materials for MRP. Production and Inventory Management, december, 1972, pp. 19-42.
12. E.A.H. Platier. A Logistical View on Business Processes: Concepts for Business Process Redesign and Workflow Management. PhD thesis, Eindhoven University of Technology, Eindhoven, 1996. (in Dutch)
13. J.M. Proth, X. Xie. Petri Nets, A Tool for Design and Management of Manufacturing Systems. Wiley, 1996.

14. L. Recalde, M. Silva, J. Ezpeleta, E. Teruel. Petri Nets in Manufacturing Systems: An Examples-Driven Tour. In: J. Desel, W. Reisig, and G. Rozenberg (eds.), ACPN 2003, LNCS 3098, pp. 742-788. Springer-Verlag, Berlin, 2004.
15. H.A. Reijers. Design and Control of Workflow Processes: Business Process Management for the Service Industry. Lecture Notes in Computer Science 2617. Springer-Verlag, Berlin, 2003.
16. H.A. Reijers, S. Limam, W.M.P. van der Aalst. Product-Based Workflow Design. Journal of Management Information Systems, 20(1): pp. 229-262, 2003.
17. H.A. Reijers, I.T.P. Vanderfeesten. Cohesion and Coupling Metrics for Workflow Process Design. Proceedings of the 2nd International Conference on Business Process Management, pp. 290-305, Potsdam, 2004.
18. J.D. Ullman. Elements of ML Programming. Prentice-Hall, 1993.
19. E.A. van Veen. Modelling Product Structures by Generic Bills-of-Material. PhD Thesis, Eindhoven University of Technology, 1990.
20. E.A. van Veen, J.C. Wortmann. Generative bill of material processing systems. Production Planning and Control 3 (3), pp. 314-326.
21. M. Zhou, K. Venkatesh. Modeling, Simulation, and Control of Flexible Manufacturing Systems: A Petri Net Approach. World Scientific, Londen, 1999.

A Declarations

In this appendix the color, variable and function declarations of the CPN model are listed in ML language [18]. It contains values that belong to the 'GAK'-case, which we used for simulation.

A.1 color declarations

```
color IE = INT;                          color IEVS = list IEV;
color ID = STRING;                       color DURATION = INT;
color NewIE = with newIE;                color COST = INT;
color VAL = INT;                         color ATTR = product DURATION*COST;
color IEV = product IE*VAL;              color OP = product ID*IE*IES*ATTR timed;
color IES = list IE;                     color OPS = list OP;
```

A.2 variable declarations

```
var out: IE;                             var exec, nonexec:OPS;
var ins : IES;                           var id:ID;
var attr:ATTR;                           var dur: DURATION;
var cost: COST;                          var ope:OP;
var ops,ops2,ops3,ops4: OPS;             var ievs:IEVS;
var success,success2: BOOL;              var number:INT;

(* 'top' is the end product of the workflow process*)
val top=18;
(*value initial1a contains all operations from the product data model, including
their ID, output element, input elements, and duration and cost attributes*)
val initial1a =[("op1", 1,[25,37],(1,0)),("op2", 2,[25,37],(1,0)),
               ("op3", 3,[33,37],(1,0)),("op4", 4,[33,37],(1,0)),
               ("op5", 5,[37,45],(2,0)),("op6", 6,[21,37],(1,0)),
               ("op7", 7,[24,37],(1,0)),("op8", 8,[23,37],(1,0)),
               ("op9", 9,[24,39],(5,0)),("op10", 10,[13,14,34,37,42],(5,0)),
               ("op11", 11,[31],(1,6)),("op12", 15,[16],(1,0)),
               ("op13", 15,[17],(1,0)),("op14", 15,[16,17],(2,0)),
               ("op15", 16,[25,30,35,36,44],(8,56)),
               ("op16", 17,[25,30],(1,0)),("op17", 18,[1],(0,0)),
```

```
               ("op18", 18,[2],(0,0)),("op19", 18,[8],(0,0)),
               ("op20", 18,[9],(0,0)),("op21", 18,[10],(0,0)),
               ("op22", 18,[11],(0,0)),("op23", 18,[15],(0,0)),
               ("op24", 18,[9,11,15],(0,0)),("op25", 28,[25,37],(1,0)),
               ("op26", 29,[25,30,35,36],(9,0)),
               ("op27", 30,[32,37,43],(4,0)),("op28", 31,[29,40,48],(7,0)),
               ("op29", 32,[1,2,3,4,5,6,7,8,10,27,28],(20,0)),
               ("op30", 34,[36,37,41],(1,42)),("op31", 40,[39,41],(0,3)),
               ("op32", 42,[47],(1,3)),("op33", 43,[39,49],(3,6))]
(*value initial1b contains all initially available data elements with their
corresponding value*)
val initial1b =[(13,0),(14,1),(21,1),(23,0),(24,3),(25,2),(27,0),(33,2),(35,1),
               (36,3),(37,1),(39,2),(41,0),(44,3),(45,2),(47,1),(48,0),(49,2)]
(*value initial1c contains a token with the operation ID for every operation
from the product data model*)
val initial1c = 1`"op1" ++ 1`"op2" ++ 1`"op3" ++ 1`"op4" ++ 1`"op5" ++ 1`"op6"
                ++ 1`"op7" ++ 1`"op8" ++ 1`"op9" ++ 1`"op10" ++ 1`"op11"
                ++ 1`"op12" ++ 1`"op13" ++ 1`"op14" ++ 1`"op15" ++ 1`"op16"
                ++ 1`"op17" ++ 1`"op18" ++ 1`"op19" ++ 1`"op20" ++ 1`"op21"
                ++ 1`"op22" ++ 1`"op23" ++ 1`"op24" ++ 1`"op25" ++ 1`"op26"
                ++ 1`"op27" ++ 1`"op28" ++ 1`"op29" ++ 1`"op30" ++ 1`"op31"
                ++ 1`"op32" ++ 1`"op33";
```

A.3 function declarations

```
(*memb checks whether x is an element of the list l*)
fun memb x l = List.exists (fn (y,_)=>(y=x)) l;

(*membs checks whether all elements of the first list are elements of the
second list*)
fun membs l1 l2 = List.all (fn x => memb x l2) l1;

(*divides the list ops into those operations that are ready for execution and
those that are not; it returns a pair of lists*)
fun executable (ievs:IEVS, ops:OPS) =
List.partition (fn (_,_,ies',_) => (membs ies' ievs)) ops;

(*returns a list containing the elements of list ops of which the output element
and its value already are a member of list ievs*)
fun already_calculated(ops,ievs) =
List.filter (fn (_,ie,_,_) =>  memb ie ievs) ops;

(*returns a list containing the elements of list ops of which the output element
and its value is not a member of list ievs*)
fun not_yet_calculated(ops,ievs) =
List.filter(fn (_,ie,_,_) => not(memb ie ievs)) ops;

(*returns the value of data element ie*)
fun get_value (ie:IE, (ie2,v)::ievs) =
if ie=ie2 then v else get_value(ie,ievs)
| get_value(ie:IE, []) = 0;

(*checks for operation with ID id, whether condition for execution is satisfied*)
```

```
fun check(id:ID, ievs:IEVS) =
case id of
"op12" => ((get_value(16,ievs))>8)
|"op13" => ((get_value(17,ievs))<23)
|"op17" => false
|"op19" => false
|"op20" => ((get_value(9,ievs))<0)
|"op22" => ((get_value(11,ievs))<0)
|"op23" => ((get_value(15,ievs))<0)
| _ => true;

(*returns a list containing those elements of ops that don't satisfy the
condition for execution*)
fun cond_not_satisfied(ops,ievs) =
List.filter (fn (id,_,_,_)=> not(check(id,ievs))) ops;

(*returns a list containing those elements of ops that satisfy the condition
of execution*)
fun cond_satisfied(ops,ievs) =
List.filter (fn (id,_,_,_) => (check(id,ievs))) ops;

(*calculates the value for the newly produced data element by adding the values
of the input elements when the input set contains two or more elements and by
changing the sign in front of the value when the input set contains only a
single element*)
fun calculation(one::two::ins, ievs:IEVS) =
get_value(one, ievs) + get_value(two,ievs) + calculation(ins, ievs)
| calculation([one], ievs) = ~(get_value(one,ievs))
| calculation([], ievs) = 0;

(*deletes element op1 from list ops*)
fun delete(ops, op1) = List.filter (fn ope => (ope<>op1)) ops

(* returns the cost of an operation *)
fun Cost((id,out,ins,(dur,cost)):OP) = cost;

(* returns the duration of an operation *)
fun Dur((id,out,ins,(dur,cost)):OP) = dur;

(* returns the operation ope for which Val(ope) is minimal,fails if it is
called with an empty list *)
fun SelMin Val (op1::op2::tail) = if Val(op1) < Val(op2)
           then (SelMin Val (op1::tail)) else (SelMin Val (op2::tail))
| SelMin Val (op1::[]) = op1
| SelMin Val ([]) = raise Match;

(* Selects an arbitrary element from the list l *)
fun select_random(l)  = List.nth (l,discrete(0,(List.length l)-1))

(* returns ID of an operation *)
fun ID((id,out,ins,(dur,cost)):OP)= id;

(* returns element with ID id from list ops *)
fun select_ID ops id = hd(List.filter (fn ope => (ID(ope) = id)) ops);
```