Supporting Collaborative Work through Flexible Process Execution

Stefan Jablonski
Chair for Applied Computer Science IV
University of Bayreuth
Bayreuth, Germany
Stefan.Jablonski@uni-bayreuth.de

Michael Igler
Chair for Applied Computer Science IV
University of Bayreuth
Bayreuth, Germany
Michael.Igler@uni-bayreuth.de

Christoph Günther
Chair for Applied Computer Science IV
University of Bayreuth
Bayreuth, Germany
Christoph.Guenther@uni-bayreuth.de

***Abstract*— In this paper we present how a combination of declarative and imperative process modeling constructs facilitate compact presentation of complex process based applications. We show how to effectively implement such constructs in Prolog. Since our concept implicates a new way of interaction between process management system and user we also present our new concept of a worklist. It guides process executors through the execution of complex processes. Through this flexible way of executing processes collaborative work is much better supported than in traditional process management systems.**

Keywords: Flexible Process Execution, Declarative Modeling, Logic in Conceptual Modeling

I. INTRODUCTION

Process management has been accepted as adequate method to describe complex business applications and to support their enactment. Deliberately we focus on complex applications since there the benefits of a process based approach are of particular importance. Process models illustrate nicely how complex applications are structured and describe what has to be done by what persons using which tools. However, we believe that process management approaches still do not cope with complexity well. In order to substantiate this proposition we want to analyze the causes of complexity.

We focus the discussion of complexity on two situations. A process based application is complex if it consists of a huge number of different process steps (step complexity). It is not so easy to reduce this kind of complexity. Such an application can be structured by creating sub-processes through de-composition. Then process models are at least easier to comprehend. However, it is hard to eliminate process steps such that the application gets "smaller". Step complexity is a kind of an inherited feature. There is a chance that domain experts recognize that some process steps are not necessary; then this complexity can be reduced partially.

A second sort of complexity arises when a huge number of execution paths exists (path complexity). In this case the number of process steps might even be moderate. However, through the flexibility of many different execution paths complexity escalates. For example, consider three process steps A, B, and C

- which must be executed “as fast as possible”
- which all have to be executed exactly once, and
- whose executions must not overlap.

In Figure 1. a solution to this scenario is depicted. We regard this process model as complex: although only three different process steps are involved, the process model consists of 15 process steps (repetitions of the three basic steps A, B, and C), 29 arcs, and 8 flow constructs (XOR) for splitting and joining control flow. In this context it is not so relevant how to count steps and arcs; the message is that there are a lot of modeling elements although the application is rather small. The most severe drawback of this process model is that its pragmatics (what it means from an application point of view) is totally camouflaged, i.e. users do not comprehend the meaning and purpose of the process. We state that path complexity is partially avoidable when powerful process modeling constructs are applied.

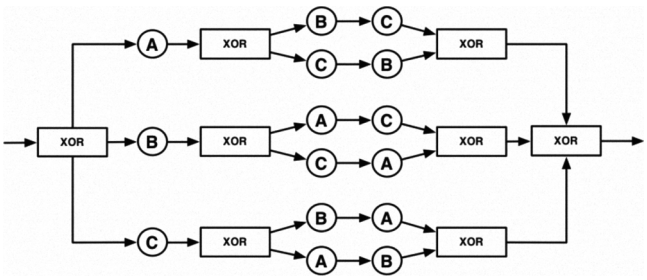

Figure 1. Example process model

There are two reasons for path complexity. One of it is a pragmatic one, the other a technical one. The pragmatic one refers to the need of collaboration. The process management approach mainly fosters coordination. However, in many (process based) applications collaboration is a main requirement. How does collaboration affect process execution? In collaborative scenarios the flow of work often bounces from one user to another. For example, one user prepares a piece of work, needs some contribution from another user and then can continue his work. To model such a scenario with conventional process modeling means leads to highly complex process models, i.e. is one of the pragmatic reasons for path complexity. Together, compact process modeling capabilities and powerful process execution guidance provides an add-on to conventional process management that is heavily requested in literature [8], [9] and [10].

What is the technical reason that still path complexity is not dealt with adequately? We see one of the major reasons in the adoption of execution rules from imperative programming

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8299
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8299

languages like sequential execution, alternative execution (if-then-else; XOR between execution paths) or independent execution (parallel execution paths). It is not that we blame (pure) imperative programming languages it is just that we state that this programming style is not adequate for process modeling. The fact that programs are going to become complex is not that bad since programs are just read by programmers, i.e. software experts that are able to cope with that complexity. In contrast to that process model complexity is problematic. Process models must also (besides professional process modelers) be comprehended by end users like medical doctors or nurses, who are not so familiar with formal process modeling techniques. Thus, when process models are becoming too complex, these people cannot interpret them anymore. That also means that they cannot assess their quality anymore and therefore cannot improve them. As a consequence we really want to promote applying process modeling techniques which reduces complexity such that complex applications can be described by comprehendible process models and can therefore be understood much easier.

We propose to apply a completely new process modeling techniques that specifically reduce path complexity. In contrast to imperative modeling – here the path how to go through a process is defined explicitly – declarative modeling concentrates on describing what has to be done and the exact step-by-step execution order is not directly prescribed. However, since pure declarative approaches often lack clarity (users do not see the process flow any more) we remain aloof from pure declarative process modeling and foster a combined approach out of declarative and imperative process modeling.

Our second focus lies on the support for the user during the execution of process models. Through the gained flexibility concerning the choice of different executable process steps there is a need for improved process navigation software that guides the user through a process model. This guidance can i.e. be to let him simulate that a process step has been done and offer him a look ahead what lies behind this situation (what is next). Sometimes it is desired to know what cannot be done any more if a certain step is executed. This information is also delivered by our planning component and helps the user to avoid decisions he will later regret.

In the following paper we describe on the one hand solutions for new modeling constructs of enriched semantic that helps to model processes much easier and more comprehensible. On the second we offer the user a planning component that helps him during the execution of a process model to keep the overview of his work and task.

II. Related Work

DECLARE [1] is a constraint based system developed by the University of Eindhoven that is focused on modeling constraints between processes. It supports the behavioral and the functional aspect of the perspective oriented process modeling (POPM) [2]. It can be combined with the workflow management system YAWL to support a decomposition of procedural and constraint based models. DECLARE provides a promising approach to declarative process modeling. However, the directly adopted declarative style of constraint based languages prevents the users and modeler to get a feeling of the standard flow through a process model. The empirical way of how a workflow should be executed is lost and cannot be modeled with this framework.

EM-BrA^{2}CE (Enterprise Modeling using Business Rules, Agents, Activities, Concepts and Events) is a Framework for unifying vocabulary and execution models for declarative process modelling [3]. The vocabulary is described in terms of the Semantics for Business Vocabulary and Rules (SBVR) standard and the execution model is presented as a Colored Petri Net (CP-Net). EM-BrA^{2}CE also follows the same concept we use in this paper to specify a state space transition relation based on rules. In particular, it does not make use of control flow modeling to indicate when and how business rules are to be enforced. Instead, it is left to the execution semantics of the declarative process models to define an execution model in which different kinds of business rules are automatically enforced. We think that there is still a demand for process modeling in a graphical way that is slightly along the lines of well known procedural modeling; this is necessary in order to get user acceptance. However, we agree with the authors of EM-BrA^{2}CE to switch to a more declarative style.

Sadiq et al. [4] show that it can be advantageous to combine both declarative and procedural aspects in a process model. They present a foundation set of constraints for partial process modeling. A process model can contain, in addition to predefined activities and control flows, several so-called pockets of flexibility. Such pockets consist of activities, sub-processes and so-called order and inclusion constraints. Each time during enactment when a pocket of flexibility is encountered, the elicitation of the work within the pocket is done by a human end-user through a so-called "build" activity. Although such a combined approach has advantages we think that a framework should be designed in a way whether it uses the declarative style or the procedural style.

III. New Elements for Compact Process Modeling

This section presents three new modeling elements which form the basis of our approach to declarative process modeling. Since we focus on the reduction of path complexity we introduce three new modeling elements, special arrows (with two different semantics), boxes (to group processes), and quantification (to define the number of executions of a process). Besides these new modeling constructs we rely on the typical modeling elements of the perspective oriented process modeling method [2]. However, in this paper we mainly focus on the functional perspective and the behavioral perspective, whereas we neglect the data, operational and organizational perspectives.

A. Two Different Types of Arrows

The first modeling construct that will be associated with a new semantics is the arrow. The semantic of the well known arrow symbol in process modeling is that if an arrow goes from process A to process B then process B has to be performed after process A. Accordingly, if process B is connected with an arrow to process C then C may start after process B has finished (Figure 2.). We also say: B requires the execution of A before it can run; C requires the execution of B (and consequently of A, too) before it can run. We want to keep this

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8299
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8299

very common construct and put it in our modeling toolbox. We present this modeling construct as a solid line.

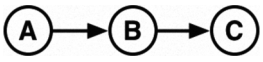

Figure 2. Sequential process flow

Beside this arrow construct depicted by a solid line we want to add an arrow depicted by a dashed line; this dashed arrow holds a different meaning. Two processes that are connected through a dashed arrow can be executed in any order. For instance, if process A and process B are connected by a dashed arrow A can be performed before B or vice versa, B can be performed before A. Nevertheless, having defined a dashed arrow from process A to process B expresses a preference (recommendation) that process A should be performed before process B. This feature can be utilized when processes are put on a work list for execution. If more than two processes are connected through a dashed line then a permutation of all process executions is feasible, e.g. ABC, BCA, CBA. This scenario is modeled in Figure 3.

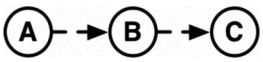

Figure 3. Model of flexible scenario

It is certainly possible to combine the solid and dashed arrows. In Figure 4. process A and B are connected through a dashed arrow; process B and process C are connected through a solid arrow. This means that there is flexible ordering between processes A and B while process B must always be executed before process C. This semantics results in the following three execution orders: ABC, BAC and BCA.

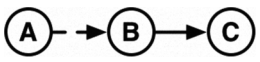

Figure 4. Combination of solid and dashed arrows

B. The Box Modeling Element

The box modeling element ensures that all the processes inside a box are regarded as a unit. Thus a box can substitute a process. That means that instead of executing a single process A or B the box must be performed, that means the processes within the box must be executed. For instance, in Figure 5. the box must be executed completely before process D can be started. Executing the box means to execute processes A, B, and C in an arbitrary order. This execution results in the following sequences: ABCD, BCAD, CABD, CBAD, ACBD and BACD. D is always the last step that requires the completion of all previous steps respectively the box in which the steps are contained.

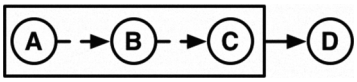

Figure 5. Box which is requirement for process D

C. Quantification

Often it is necessary to specify that a process can be executed several times. For that purpose we add a quantificational aspect to process steps. Every process gets a minimum and maximum counter that indicates how often a process may be executed. We call them *domain*. If it shall be executed exactly a certain number of times then minimum and maximum are equal. To express that a process step is not essential for the whole process but can be done in the sense of "possible but not necessary", then a minimum quantification of zero should be selected. Now let's consider a simple process example that is set up with the upper modeling constructs and quantification and is presented in Figure 6. :

Figure 6. Process model with new modeling constructs

When we take a look at the overall process of the referred Figure 6. that the process modeler has developed, then at a first glance the two boxes attract attention. They are connected through a dashed arrow, meaning that it is possible to start with one of the boxes no matter which one. Inside the left box there is a connection between process A and B, indicating that process B requires the completion of process A. Process C is dashed connected so it can be executed before A or before B if it is not desired to start with A. In the right box processes D and E are also dashed connected meaning it is not relevant from the modeler's point of view which process shall be executed first. Concerning the quantification of the upper process steps then the following dot list gives an overview how often each process shall be executed:

- Process A: minimum = 1, maximum = 2
- Process B: minimum = as often as A, maximum = 3
- Process C: minimum = 1, maximum = 1 (exactly once)
- Process D: minimum = 0, maximum = * (optional, any repetition)
- Process E: minimum = 2, maximum = 3

As an introduction to the correlation between the quantification and the arrows we'd like to exemplify that on two process steps A and B which are connected through a solid arrow. As already mentioned this connection models that B requires the completion of A to be executable. Completion means that step A is in its domain, which means it is done once or twice (indicated by 1...2). The quantification in B means it has to be done at least as often as A, but maximum three times. Having executed A once, it is not possible to do A again, after having executed an instance of B, although process step A can be done twice in general. This is important to ensure a valid solution, because otherwise a valid solution could get invalid afterwards. This would be the case if A is executed once, B has to be executed at least as often as A, so once would be a valid value also. If we allowed A to be executed afterwards, incrementing the counter of A would make the formerly valid solution invalid. By assuming a encapsulation of steps, we can guarantee a valid solution whilst allowing dependent domain borders. Another example for a correlation between process steps are D and E which are connected through a dashed arrow. The user has the flexibility to start with one of the two processes. Now let's assume the

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8299
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8299

process executor has executed E twice, so it can be seen as done. After starting the execution of process D, he must be aware that E cannot be executed anymore afterwards. The restrictions, not being able to jump back again (i.e. in this case or in the case when B is done and process A executed once) are fundamental rules in the system which cannot be influenced by the modeler. We'd like to mention that there is not much effort to change these rules to another desired semantic. The described structure offers a huge field of application for producer-consumer scenarios. They could model situations like: A may produce an arbitrary amount of parts; B has to check all of them. This of course is very interesting for collaborative work, as the tasks of every party can be defined precisely and even considering the exact count, which is determined on runtime.

IV. PROCESS PLANNING

Taking the process model form Figure 6. we can talk about process planning. Why do we have to talk about process planning at all? In contrast to conventional process models which typically rely on imperative modeling our modeling approach combines declarative and imperative modeling principles. We did this in order to provide flexible process execution, i.e. many alternative paths for executing a process are offered. However, this feature also comes with a challenge: sometimes it is hard for users to see what processes can still be performed and what processes cannot be performed any more. Further on it is hard to know who has to execute a certain step. In imperative process models, it's usually clear which steps have to be performed after each other, as they only describe a few different cases. Providing the described flexibility of step execution in imperative process models makes it much more difficult to figure out, who in a collaboration has to perform a certain step and when. In order to support users in this respect we offer a planning component. This planning component provides an "overview" to users: they know which steps can/cannot be executed and which consequences the execution of process steps has. For example, executing a specific step prevents other process steps to be executed. In the example of Figure 6. this is the case when the user decides to begin with one of the boxes.

There are a few basic questions that can be asked in every phase of process execution that helps the executor of a process model to assess what to do next. We'd like to mention that for collaboration between process executors the organizational aspect [2] has to be implemented in ESProNa, which is currently under research. With this aspect it is possible to add a "Who" to the question "What is executable next?" presenting the process executor the information if he can execute this process or which person he needs to contact to execute it. In the following subsections we assume different situations in which a user is supported by our system. This supports refers to the following three issues: how the user can continue process execution, how he can simulate things, and how he is guided to achieve the execution of a specific process step.

A. What can I do at the beginning?

This question aims at the feasible process steps that can be executed in the actual state of process execution. Assuming nothing has been done yet in example [Figure 6.] the system would deliver the steps A, C, D and E. As the two boxes are connected by a dashed arrow and the execution order of A and C as well as the one of D and E is arbitrary the upper delivered process steps are possible to execute. This provides an overview of the initial doable working steps to the user.

B. Simulate the execution of steps

As soon as a user clicks onto a possible process step, the system simulates its execution and delivers the consequences of his choice. The consequences are presented in three different lists, named *after, somewhen* and *not_after*. As the names do already suggest, the first list displays the steps that can be executed after the execution of the selected process step. The second one displays process steps that will never be allowed to be executed after having finished the selected process step. The last list displays all steps that still will have to be executed to finish all process steps successfully.

If we now assume the user preselected process step C as initial process step for execution then the system delivers the following information:

after	somewhen	not after
A	A B D E	C

Figure 7. The three planning lists

The *after* list displays process step A, as the whole box would have to be executed first, before being allowed to execute one of the other steps in the right box. This is because of the rule that there is no jumping between the boxes allowed as long the content of the box is not completed. The *somewhen* list would display the steps A, B, D and E, as those will have to be executed some when. The *not_after* list would display only C, as it is the only process step that will never be allowed to be executed anymore. This provides a full overview of the consequences of the user's decision. The decision if the previously selected process step is really executed is taken by the executor himself. But with the consequences we offer him through *after*, *not_after* and *somewhen* we support him in his decision whether to start the selected process step or to choose another one.

C. How to achieve the execution of a specific process step

For example, the only process that has been done is step E. The system will now displays D as next step as it is contained in the box. Now assume that the user could be interested in the fastest possibility to execute a certain process step i.e. process B. Therefore he navigates over one of the processes in the *somewhen* list. The system displays all process steps he will have to execute before he will be able to execute the desired process step (see Figure 8.). In our case if the user navigates over B, the system would display steps D and A. He would first need to finish the execution of the box and it is necessary

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8299
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8299

to execute A before B, as they are connected by a solid arrow, meaning B depends on the completion of A.

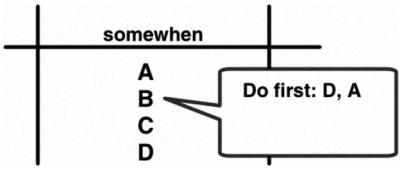

Figure 8. Displayed required process steps for B

V. IMPLEMENTATION OF ESPRONA

In the following section we will have a look at the implementation of our system which is called Engine for Semantic Process Navigation (ESProNa). We use declarative programming, namely Prolog [5] and Logtalk [6], [7] to implement the described functionality. Declarative programming on the one hand is not like the usual imperative one, where you have to specify which operations are performed (What?) in a specific order (How?). Instead, only rules (What?) are defined leaving the aspect how things are done unspecified as long as no rules are violated. Consider a simple statement like:

```
test(A) :-     between(1,5,A).
```

This means the system checks if the value of a given variable *A* is between one and five. If so, the system returns *true*. But things are getting really interesting if we leave the variable *A* unspecified. Then the system looks for valid solutions:
This means we get all solutions of a statement, one at a time. It is obvious, that this approach is very applicable for a highly

```
A = 1 ;
A = 2 ;
A = 3 ;
A = 4 ;
A = 5.
```

flexible approach like ours. We submit the rules of interpretation to the system. It generates statements that have to be fulfilled by any order of executed process steps and then checks which orders are valid, i.e. satisfy the previously generated statements.

In the following subsection A we will show how we generate the answers to the questions we posed in section IV using the techniques of the previous paragraph. Therefore we use a predicate called *getDeps/2*. Its importance will be described in the following sections. Afterwards we will have a look at the process model we generate and its interpretation to generate static (subsection C) and dynamic (subsection D) dependencies. Finally we will describe the systems behavior on runtime in subsection E.

A. Navigation features

Now we can ask which process steps can be executed next in a certain state. Therefore we call the predicate *executable/2* which takes two arguments. The first one is the state; the second one is a process step. As we use Prolog, the predicate succeeds if the given process step is executable in the current state. But we can also leave the second argument unspecified. Then the predicate will deliver a list of process steps, for which the called predicate would have succeeded. This is the information we were looking for, as this means we get all process steps that could be executed in the current state.
Inside *executable/2* we use the simulation of a state transition to determine if the tested process step is allowed to be executed in the current state. It tests several factors.

1) Is the process step still to do?
This is tested first, as it is the cheapest operation. This is especially important, when we ask without specifying a process step. It simply tests if the process step is already marked as done in the current state.

2) Are all dependencies already done?
This operation gets all dependencies from the process object, using *getDeps/2*. Then it checks if all process steps, the current step depends on, are already done. This is a quite special behavior and can easily be changed into arbitrary rules, like if a step is started, worked on, etc.

3) Does the state transition deliver a valid result?
This checks if the new state provided as an argument is reachable regarding the given state and process step. If all of those three queries succeed, the given process step is executable in the current state and will create a transition into the new state, given as an argument.
If we leave some of the arguments unspecified, the predicate will deliver pairs of arguments which would succeed, i.e. all feasible process steps and the resulting new state after their execution.
This leads to the second question we asked in section IV. What happens if we would do a certain step? We can use the described predicate to simulate more than one execution. Thus we can provide all consequences of starting the execution of a process step as a decision support. We could do this up to arbitrary depth, but offer it only for one step at the moment because otherwise we could get an overwhelmingly huge number of possibilities, which means lots of computational complexity as well as possible confusion of the user.
That is why we take the resulting new state we get from the state transition simulation and ask again for processes that could be executed, couldn't be executed after the current process step and a general roadmap.

```
executable(NewState,ProcessStep).
```

e
with the new state. It's important to mention, that we calculate the possibilities not before the user selected a process step. This keeps the computation low, as we have only one state. If we precomputed the possibilities, we would need to calculate the corresponding possibilities for all elements in the list, which would mean quadratic complexity.
We get the additional information for free, as the new state already contains information about the process steps that are marked as *done* as well as the ones that are still marked as *todo*.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8299
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8299

The third question is a more complex one. To determine which process steps would have to be made to be able to perform a certain step, we need to ensure that all dependencies are already satisfied.
We call the corresponding predicate *what_to/3*.

```
what_to(_, [], []).
what_to([done:Done|_], [X|R], B) :-
        what_to([done:Done|_], X, Bef),
        what_to([done:Done|_], R, Bef2),
        subtract(Bef2, Bef, Result2),
        append(Result2, Bef, B).

what_to([done:Done|_], X, [])     :-
        X::getDeps(Done, Deps),
        subset(Deps, Done),!.
what_to([done:Done|_], X, Before):-
        X::getDeps(Done, Deps),
        subtract(Deps, Done, Result),
        what_to([done:Done|[]], Result, Todo),
        subtract(Todo, Result, Result2),
        append(Result2, Result, Before),!.
```

procedures. We will start with the second two, which handle the dependencies of a single process step in a certain state. This is the default case that is called if the user navigates over a process step in the roadmap. The first one is a trivial case: It askes for all dependencies of the given process step X and checks if all of them are already done. If so, we don't need to do anything before being able to execute X.
The second one handles if there are dependencies that are not yet done. So we subtract all the dependencies that are already done and call *what_to* with the remaining dependencies, stored in the *Result* list. This delivers a list of process steps that are still *todo*. In the last two lines we combine the two lists.
The call of *what_to* with the result list is realized in the first two procedures. The first procedure again realizes a trivial case that we need as final statement because we want to realize the walk through the list recursively. It just says, that if we look at an empty list, we don't have to do any steps. The second case is used to handle a not empty list. It calls the *what_to* predicate for every element in the list, and combines the resulting dependencies.
This means that we check the dependencies of the given process step X, check which of them are not already done and check if the remaining process steps have further dependencies we need to satisfy first. Using this kind of backtracking, we ensure to only check the process steps a preceding step depends of. This again reduces complexity.

B. Process model

All of the previously described predicates rely on a predicate *getDeps/2* offered by every process object. It delivers the dependencies of every process step in the current state, which are also saved in Logtalk objects. The function takes the current state as an argument, as the dependencies differ accordingly to the state, because of dashed connections and boxes (Figure 6.). It delivers all dependencies of the process step in the current state, according to the aspects that are important for the current process. Like this we can easily add functionality or make the navigation faster, depending on the application domain requirements. We will describe the aspects in section VI. To generate those dependencies we need to interpret the drawn process model in a way the system can understand. To do this we convert the arrows into a description of connections. Like this we can convert the graphical process model into a textual descriptive one. This will be our general approach. The connections in Figure 6. Figure 4. are shown in the following listing after the conversion.

```
solid_arrow(a,b).
dashed_arrow(b,c).
dashed_arrow(box_1,box_2).
dashed_arrow(d,e).
```

y
we use process steps. The process steps are named with small characters here, because Prolog does not allow to use uppercase, unless for variables.
Now we can start interpreting this system in our described way. There will be two different kinds of dependencies, static dependencies which are independent of the current state and dynamic dependencies which are not independent. We will describe both types in the following subsections.

C. Static dependencies

As a solid arrow holds only information about one direct connection (i.e. the connected element), no further interpretation is needed. We now want to use the modeled behavior to build dependencies for the process steps. This happens in the initialization phase. To model the dependencies out of solid connections, a simple rule is enough, as every perspective is checked separately.

```
require(X,R):-
        sconn(W,X) -> depList(W,R);
        sconn(W,Y), rec_box(Y,X) ->
                depList(W,R);
        R = [].
```

at
would trigger dependencies. The first line checks whether X is object of a solid connection and determines thereof dependencies through *depList/2*, which will be described later. The second line checks if X is member of a box that is object of a solid connection and again calculates dependencies out of this, using *depList/2*. The last line just says that there are no dependencies if the first statements failed.

```
depList(X,R):-
        xor((box(X,List)
        depRList(List,R)), R=[X]).

depRList([],[]).
depRList([X|List], RList) :-
        depList(X, DL), depRList(List, DRL),
        union(DL,DRL,Rlist).
```

a
given element W. If the connected element X is a process step, we write X into the dependency list and finish. If the linked element is a box we need to put every element inside the box into our dependency list. As boxes may contain nested boxes in arbitrary levels, we need to call *depList/2* recursively for every element in the box and combine the resulting dependency lists. This is what *depRList/2* does.

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8299
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8299

D. Dynamic Dependencies

As a dashed arrow is not in fact an arrow, it is connected to all elements within a dashed sequence. To give a rule for this behaviour, we write:

```
dconn(X, Y):- dashed_arrow(X,Y).
dconn(X, Z):- dashed_arrow(Y,Z),  dconn(X,Y).
```

steps within one dashed sequence are interpreted as directly dashed connected; this means they are arbitrarily swappable in their execution order.

Dynamic arrows between process steps don't determine a fixed execution order, so they don't imply dependencies. But we need to ensure the encapsulation of boxes, as we are not allowed to jump between them, before every nested element got executed. This is why we need a special technique, which we will refer to as dynamic dependencies. We use the state to trigger dependencies to ensure the encapsulation. This means as soon as one element of a dashed linked box is executed, all nested elements A of the box become a dependency of the dashed connected element. If the dashed connected element is a box A becomes dependency of the all its nested elements. This models exactly what we described in section III. As the two rules of findDyn/2 already suggest, this can easily be modified, extended or changed, accordingly to the current use case.

```
dyndeps(X, DL) :-
        findall(D, findDyn(X, D), DL).

findDyn(X, D)  :-
        rec_box(Y,X),
        (dconn(Y,Z);dconn(Z,Y))
        ->      depList(Z,BList),!,
                list_to_set(BList, Box),
                D=Box:Box.

findDyn(X, D)  :-
        (dconn(X,Y);dconn(Y,X))
        ->      depList(Y,BList),!,
                list_to_set(BList, Box),
                D=Box:Box.
```

The first rule asks if a process step X is a nested element in a box Y. The procedure rec_box/2 is used to find nested elements in boxes in arbitrary depth. Afterwards we determine whether there is a dashed connection between the box Y and an element Z. Using our former predicate depList/2, we don't need to care about Z being a box or a process step. We grab the list BList, remove duplicates and generate a Triggered Dependency construct, which we return. The other procedure takes care of the reverse case. If the element X is directly connected to an element Y, we again use *depList/2* to gain the dependencies, not taking care of the type of element Y represents. After removing duplicates, we return constructs of the same shape as dependencies.

This gives us the possibility to adopt dependencies to a certain state, which means that we can ensure encapsulation of boxes.

In fact, dynamic dependencies could be used to interpret arbitrary modeling constructs arbitrarily. This makes them a very powerful tool regarding scalability, adaptability and flexibility.

After extracting the static and the dynamic dependencies out of the model we create a new dependency object using Logtalk to save them. After linking the object to our process object, we can simply access its dependencies calling *getDeps/2*. So we do not need to parse the process model on runtime.

We generate some other objects in the initialization phase. Right now there is one domain-object created for every process step. This takes care of the cardinality of the process step, offers predicates to check if it is in its domain and takes care of dependent domains. Dependent domains are needed to model a producer consumer relation (do A at least as often as B, or vice versa). Outsourcing this functionality simplifies the process step objects and lets us concentrate on the three states of a process.

E. States of processes and runtime

The three states of a process step are *todo*, *active* and *done*. This abstraction is possible as we don't care about the domain of the process step, but use the domain objects predicate *inDomain* to determine whether a process step is executed a valid number of times regarding its domain.

This abstraction allows us to take the cardinality out of the generated state space. This makes the space smaller keeping a domain validity check, because we rely on the predicates the domain object provides.

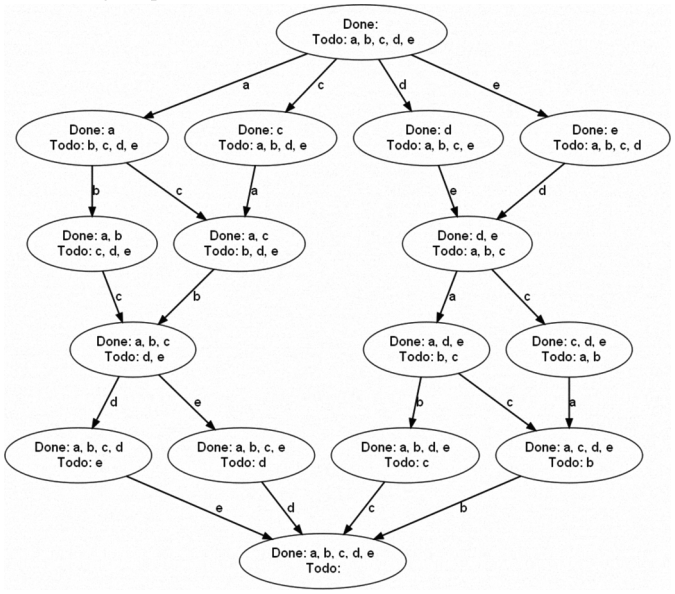

Figure 9. The state machine representing all possible execution orders

Like this we can use a kind of deterministic state machine, to execute a process. We have a well defined start state, where no process step is executed and a well defined final state, where all steps are executed. We don't need to look at optional steps separately, as the function *inDomain* returns *true* even if we didn't execute a single instance of the step. That is why we can assume a single final state.

Start state and final state are connected with lots of paths through lots of states. All those paths represent all valid execution orders. As this is a tremendous number for fairly small processes, we do not precompute them. We also do not

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8299
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8299

define a fixed state transition relation, as this would take way more memory and time than just combining all process steps ($O(N^2)$), as the transitions depend on the current state. That means we would have to combine all process steps and the states with each other. The number of possible states can be at most $N!$ which makes this approach impractical.

We use an implicitly defined state transition relation that is already given through the dependencies. This allows us to determine a valid transition very fast. And as we do not need to look at more than one transition at a time, we simplify our complexity significantly. Listing all possible paths through a state space would be of complexity $O(N!)$. Navigating through it, using implicitly defined state transition relation and a look ahead of one takes $O(N^2/2)$ at maximum, because the steps we have to check are reduced by one at a time.
Like this the calculation effort as well as the storage space needed for navigation, can be reduced from exponential order to polynomial one.

VI. Classification of Planning Queries

A process is way more than an array of executed steps. Especially in ESProNa, it is absolutely necessary to guide a user through process execution since so many alternative execution paths are available. To get grips of the huge amount of information stored in a process model, we offer a classification of two categories. The first one is called *Navigation* and handles issues that aim at finding one or more process steps that fulfill a certain criteria. Questions as discussed in Section IV fall into this category. We distinguish between constructive questions, which aim at process steps that can or should be done and destructive questions aiming at process steps that cannot yet or anymore be done.

To answer those questions we rely on the predicate *getDeps/2* of a process object to obtain dependencies between process steps. It interprets the information of different perspectives of process steps to determine dependencies between them. The method Perspective Oriented Process Modeling (POPM, [2]) provides a useful classification of dependencies. There, process steps are composed of so called perspectives. We use the perspectives of a process step according to POPM to classify reasons for dependencies. We want to emphasize that the sets of proposed queries grouped according to the perspectives are not complete and more/alternative questions can be integrated easily.

A. Functional perspective

This perspective handles technical information about process steps. Typical dependencies that are generated out of this perspective might be the following: Assume we want to model that a process step B has to be done at least as often as A. This again means, B is dependent of A.

B. Behavioral perspective

This perspective encompasses questions that aim at the current execution state of a process step. A typical question is: "Is the process step active right now?" Other questions could ask if the process step is still to do, or if it has already been done. The dependencies contained in this perspective are the ones usually drawn in process models. Every solid arrow between process steps, directly triggers a dependency. So there is no need for interpretation.

C. Organizational perspective

The operational perspective handles information about executing agents. We need to be able to answer questions like: "Who should perform this step?". A dependency triggered by the organizational perspective might be that process B has to be performed by the supervisor of that person who was executing process A. This is often the case if a document hast to be reviewed or a transaction affirmed. But this triggers a dependency, because we first have to know who did A, before we can determine the corresponding supervisor.

D. Data perspective

The data perspective finally handles input and output of a process step. Questions that aim at required data or produced data can be placed here. We even handle real objects as data, because they are represented by documents or data sets in our system. For example, a process step B needs a specific data item that is produced in step A. This causes a dependency from B to A.

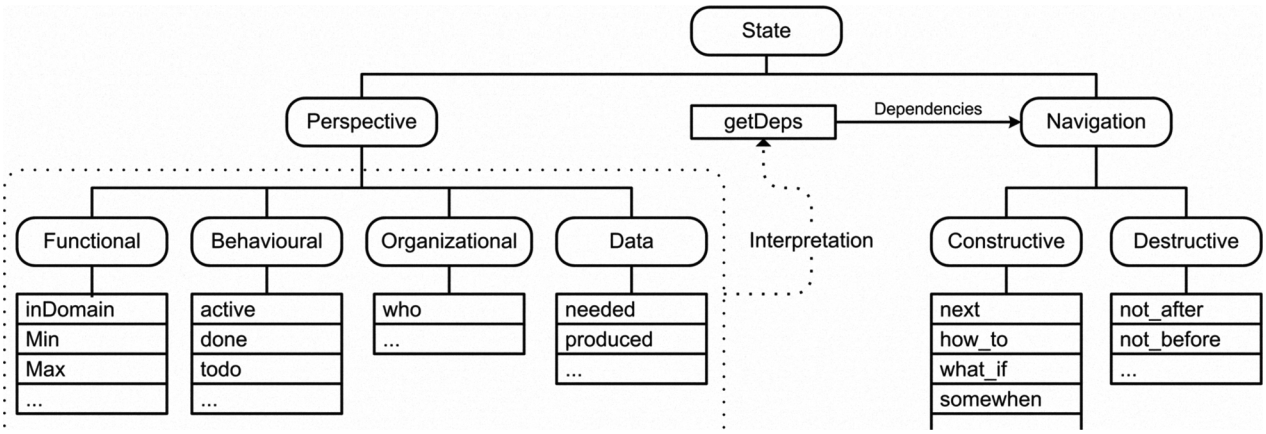

Figure 10. Classification of possible questions respecting the Information and the Navigation view

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8299
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8299

All perspectives are cumulated by the *getDeps/2* predicate. It interprets the contained information applying the predefined semantics of the current use case. The output is a set of dependencies. There is no subdivision of the set, because we don't need to distinguish the perspectives anymore. A dependency means that the dependent step can't be executed before his dependencies finished. It is sufficient if one perspective triggers a dependency. Only if there isn't any dependency to be fulfilled anymore, the process step becomes executable.

VII. USABILITY STUDIES IN THE FORFLOW PROJECT

ESProNa is part of the ForFlow Process Navigator [11]. This system is developed in the joint research project ForFlow [12] among 4 Bavarian Universities and about 30 industrial partners. The Process Navigator (Figure 10. Figure 10.) is divided into a worklist (left side), where the executable work steps are displayed, and a planning component (right side), which offers information concerning the processing of all work steps. When we relate to the process model of Figure 6. and assume that we are in the initial state where no process step has been executed yet, then the ForFlow navigator would display the information that is displayed in Figure 11. On the worklist (shown as "possible steps") all possible process steps are displayed that are in principal executable (A, C, D and E). Furthermore the user of the system has preselected process A and he has the opportunity to start this process through the button "Start". On the planning side a general overview is displayed: "Next step after A" displays information that informs the user which set of steps are directly executable after the user has done step A. In our case he can decide between the process steps B and C. In the column "Roadmap after step A" the user is informed about what steps still have to be done to complete the process model. Notice that here also D and E are displayed which are not directly available in the left column of the planning component ("Next step after A"). The reason is that D and E are in the right box of the process model and for executing these two processes, the left box where A is inside has to be completed first. Now imagine the user is interested in exactly this process D that is not directly available (but later) with the selection of step A. When he places the mouse pointer on the process D in the list "Roadmap after A", then a small popup window appears, as described in section IV.C, telling the user which processes need to be completed first in order to be able to execute process D. With this information the process navigator not only displays processes that are in principal executable or still to do up to the finalization of the process model. It also supports him when he needs information that lies in between two assumptions (step A preselected and the possibility of executing D). This feature influences his decisions in a very positive way. It can be seen as if an expert who has done this exact process model many times will give him advices and supports him when he is uncertain. Through the possibility to preselect steps before really executing them and querying information concerning the whole process scenario we have noticed that in evaluations with the Process Navigator users really appreciate this feature.

Further on there is a huge amount of additional functionality on the Process Navigator ([11]) like document management, multiuser and task management as well as additional navigation levels. Those levels depend on the desired amount of flexibility of the process execution. The strict mode only provides the next step of a reference (*default*) workflow. The next level is the one described in this paper, which ensures valid process execution even though users have the possibility to change the order of the steps. The last one is a fully free order, without any validity check. We want to provide a system that provides maximum flexibility of the process execution, as well as guaranteed validity, regarding the previously modeled behavior.

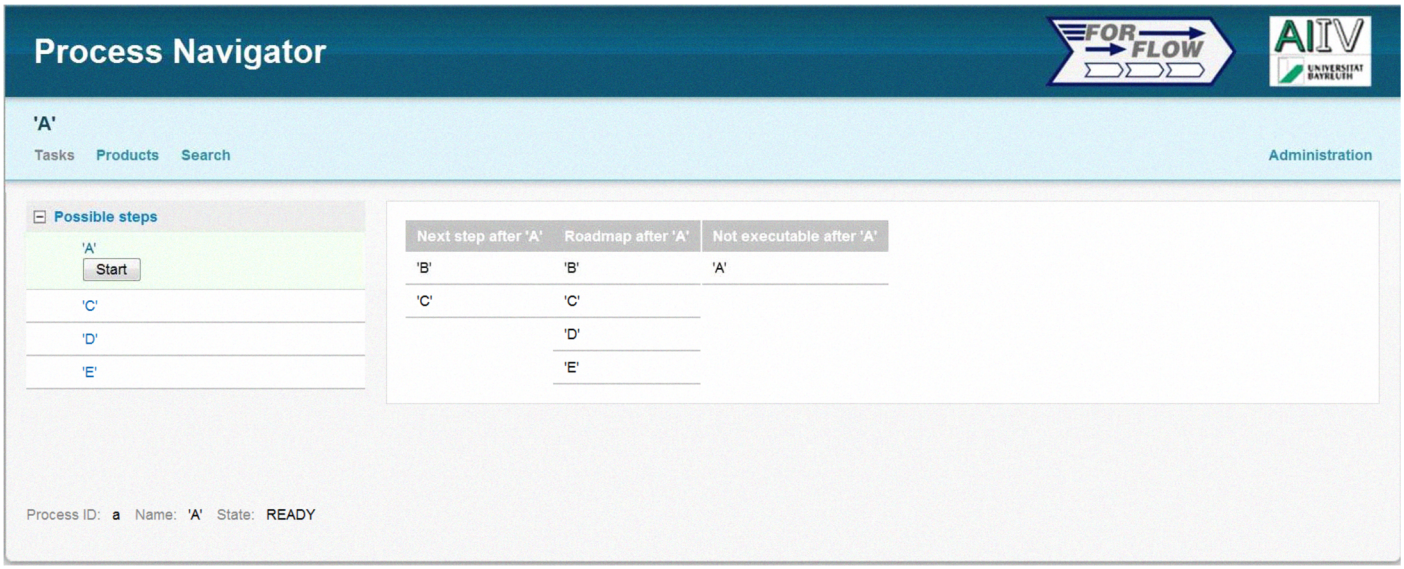

Figure 11. The ForFlow Process Navigator based on ESProNa

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8299
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8299

VIII. CONCLUSION AND OUTLOOK

Through the combination of declarative and imperative modeling styles, the ESProNa system can effectively support collaborative work: Typical collaborative situations when the flow of work often bounces between work steps can be modeled in a compact way. Also process execution supports this kind of collaboration. So ESProNa extends traditional coordination capabilities of process management to collaboration features. The practical application of ESProNa in diverse projects in the context of the ForFlow project proofs the applicability of the concept.

One of the next phases in the ESProNa project will be to revisit the definition of the execution semantics. So far, we heavily rely on deriving this semantics from the semantics of the underlying Prolog implementation. However, we aim at separating semantics off the program implementation and describe it in a more general, more formal way. Another step is the implementation of additional perspectives and views, using the new separated semantics.

REFERENCES

[1] Pesic, M., Schonenberg, H., and van der Aalst, W. M. 2007. DECLARE: Full Support for Loosely-Structured Processes. In Proceedings of the 11th IEEE international Enterprise Distributed Object Computing Conference (October 15 - 19, 2007). EDOC. IEEE Computer Society, Washington, DC, 287.

[2] Jablonski, S.: Functional and behavioural aspects of process modelling in workflow management systems. In: G. Chroust and A. Benczur (eds.): Proceedings of CON'94, Workflow Management: Challenges, Paradigms and Products, Linz, Austria, R. Oldenbourg München, pp. 113-133, 1994

[3] Goedertier, S., Haesen, R., Vanthienen, J. (2007). EM-BrA2CE v0.1: A vocabulary and execution model for declarative business process modeling. FETEW Research Report KBI_728, 74 pp. Leuven: K.U.Leuven.

[4] Sadiq, S. W., Orlowska, M. E., and Sadiq, W. (2005). Specification and validation of process constraints for flexible workflows. Inf. Syst., 30(5):349–378.

[5] http://www.swi-prolog.org

[6] Paulo Moura. Logtalk - Design of an Object-Oriented Logic Programming Language. Department of Computer Science, University of Beira Interior, Portugal. 2003. http://logtalk.org/papers/thesis.pdf

[7] Paulo Moura. Logtalk 2.6 Documentation. University of Beira Interior, Portugal. DMI 2000/1. http://logtalk.org/files/trdmi20001a4.pdf.gz

[8] Petra Heinl, Stefan Horn, Stefan Jablonski, Jens Neeb, Karin Stein, Michael Teschke: A Comprehensive Approach to Flexibility in Workflow Management Systems. Proc. WACC'99, San Francisco, 02.1999

[9] Rinderle, S.; Reichert, M.; Dadam, P.: Correctness Criteria for Dynamic Changes in Workflow Systems - A Survey. Data and Knowledge Engineering, Special Issue on Advances in Business Process Management 50(1), pp. 9-34, 2004

[10] Wil van der Aalst, Stefan Jablonski: Special Issue on „Flexible Workflow Technology Driving the Networked Economy". International Journal on Computer Systems Science & Engineering (CSSE), Vol. 15 (2000), No. 5

[11] M. Faerber, S. Meerkamm, and S. Jablonski, "The ProcessNavigator - Flexible process execution for product development products," in International Conference on engineering design, ICED'09, Stanford, CA, USA, 2009

[12] Meerkamm, H.; Paetzhold, K.: Bayerischer Forschungsverbund für Prozess- und Workflowunterstützung zur Planung und Steuerung der Abläufe in der Produktentwicklung, ISBN 978-3-9808539-7-2

Digital Object Identifier: 10.4108/ICST.COLLABORATECOM2009.8299
http://dx.doi.org/10.4108/ICST.COLLABORATECOM2009.8299