# Yet Another Event-Driven Process Chain (Extended Version)

Technical Report JM-2005-05-27
Vienna University of Economics and Business Administration

Jan Mendling[1], Gustaf Neumann[1], and Markus Nüttgens[2]

[1] Department of Information Systems and New Media
Vienna University of Economics and Business Administration
Augasse 2-6, A-1090 Vienna, Austria {firstname.lastname}@wu-wien.ac.at
[2] Chair of Information Systems
University of Hamburg
Von-Melle-Park 9, D-20146 Hamburg, Germany
nuettgens@hwp-hamburg.de

**Abstract** The 20 workflow patterns proposed by Van der Aalst et al. provide a comprehensive benchmark for comparing control flow aspects of process modelling languages. In this paper, we present a novel class of Event-Driven Process Chains (EPCs) that is able to capture all of these patterns. This class is called "yet another" EPC as a tribute to YAWL that inspired this research. yEPCs extend EPCs by the introduction of the so-called empty connector; inclusion of multiple instantiation concepts; and a cancellation construct. Furthermore, we illustrate how yEPCs can be used to model the workflow patterns. Finally, we describe how yEPC extensions can be represented in EPC Markup Language (EPML).

## 1 Introduction

The 20 workflow patterns gathered by Van der Aalst, ter Hofstede, Kiepuszewski and Barros [1] are well suited for analyzing different workflow languages: workflow researchers can refer to these control flow patterns in order to compare different process modelling techniques. This is of special importance considering the heterogeneity of process modelling languages (see e.g. [2]). Building on the pattern analysis and on the insight that no language provides support for all patterns, Van der Aalst and ter Hofstede have defined a new workflow language called YAWL [3]. YAWL takes workflow nets [4] as a starting point and adds non-petri-nets constructs in order to support each pattern (except implicit termination) in an intuitive manner.

Besides Petri nets, Event-Driven Process Chains (EPC) [5] are another popular technique for business process modelling. Yet, their focus is rather related to semi-formal process documentation than formal process specification, e.g., the SAP reference model has been defined using EPC business process models [6].

The debate on EPC semantics (see e.g. [7,8,9]) has recently inspired the definition of a mathematical framework for a formalization of EPCs in [10]. As a consequence, we argue that workflow pattern support can also be achieved by starting with EPCs instead of Petri nets. This paper presents an extension to EPCs that is called yEPCs. The letter y is an abbreviation for "yet another" and a tribute to YAWL that inspired this research. In Section 2 we introduce EPCs and yEPCs. yEPCs introduce three extensions to EPCs that are sufficient to provide for direct support of the 20 workflow patterns reported in [1]. As EPCs are frequently used for business process modelling, we expect yEPC extensions not only to be interesting for the research community, but also useful for the modelling practice. In Section 3 we discuss in detail how workflow patterns can be expressed with yEPCs. In particular, we highlight the non-local semantics of the XOR join, and its implications for workflow pattern support. Finally, we discuss how EPC Markup Language (EPML) can be extended in order to capture yEPCs syntactically (Section 4). After a survey on related work (Section 5), we give a conclusion and an outlook on future research (Section 6).

## 2 Yet Another Event-Driven Process Chain (yEPC)

In [5] EPCs are introduced as a modelling concept to represent temporal and logical dependencies in business processes. Elements of EPCs may be of *function type* (active elements), *event type* (passive elements), or of one of the three *connector types* AND, OR, or XOR. These objects are linked via control flow arcs. Connectors may be split or join operators, starting either with function(s) or event(s). OR split and XOR split are prohibited subsequent to events. This restriction refers to the semantics of events as passive elements which are unable to determine the functions that should follow. In EPCs both OR Join and XOR join have non-local semantics (cf. [8,10]). Concerning the XOR join, this implies that it blocks when there is one incoming branch finished and another still active. For a formal discussion of these semantics refer to Kindler [10]. Furthermore, *process interfaces* and *hierarchical functions* (see e.g. [11,8,12]) can be used to link different EPC models: process interfaces can be used to point from the end of a process to a subsequent process; hierarchical functions point from a function to a refining sub-process. A hierarchical function can be regarded as a synchronous call to that sub-process. After the sub-process has completed, navigation continues with the next function subsequent to the hierarchical function. In BPML such sub-processes are modelled as a `call` activity [13]. The process interface can be regarded as an asynchronous spawning off of a sub-process. There is no later synchronization when the sub-process completes. In BPML such behavior is modelled as a `spawn`. For more details on EPC sub-processes refer to [8].

Figure 1 illustrates the syntax elements of Yet Another Event-Driven Process Chain (yEPC). This extension of EPCs is motivated by incomplete workflow pattern support of EPCs and it is inspired by YAWL [3]. yEPCs reflect three measures that suffice to provide for direct modelling support of all workflow patterns [1]. These measures include the introduction of the so-called empty

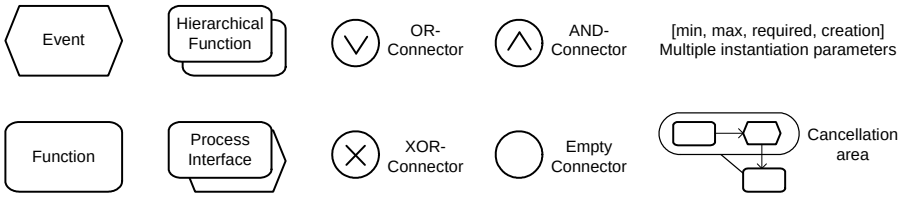


**Figure 1.** yEPC Symbols

connector; an inclusion of a general multiple instantiation concept; and the introduction of a cancellation concept. The EPC extensions differ from Petri net extensions that were needed to define YAWL: Petri nets also had to be extended with multiple instantiation and cancellation concepts, but they lacked advanced synchronization patterns. EPCs, in contrast, miss support for state-based patterns. It should be mentioned that the yEPC extensions have no impact on the validity of existing EPC models: this means that valid EPCs according to the definitions in [5,8] are still valid with respect to this new class of EPCs.

As mentioned above, EPCs cannot explicitly represent state-based workflow patterns. This shortcoming can be resolved by introducing a new connector type that we refer to as the *empty connector*. This connector is represented by a cycle, just like the other connectors, but without any symbol inside. Semantically, the empty connector represents a join or a split without imposing a rule. Consider an event that is followed by an empty split that links to multiple functions. The empty split allows all subsequent functions to pick up the event. As a consequence, there is a run between the functions: the first function to consume the event causes the other functions to be active no more. We will show in Section 3 that this split semantics match the deferred choice pattern. Consider the other case of an empty join with multiple input events. The subsequent function is activated when one of these events has been reached. This behavior matched the multiple merge pattern. We will explain in Section 3 why such semantics are needed as an EPC extension.

The lack of EPC support for *multiple instantiation* has been discussed before (see e.g. [14]). In yEPCs we stick to multiple instantiation as defined for YAWL. For further work on this topic, see e.g. [15,16]. YAWL defines a quadruple of parameters that control multiple instantiation. The parameters `min` and `max` define the minimum and maximum cardinality of instances that may be created. The `required` parameter specifies an integer number of instances that need to have finished in order to complete multiple instantiation. The `creation` parameter may take the values `static` or `dynamic` which specify whether further instances may be created at run-time (`dynamic`) or not (`static`). In the context of multiple instantiation, it is helpful to define sub-processes in order to model complex blocks of activities that can be executed multiple times as a whole. Accordingly, multiple instantiation parameters can be specified for functions as well as for hierarchical functions and process interfaces.

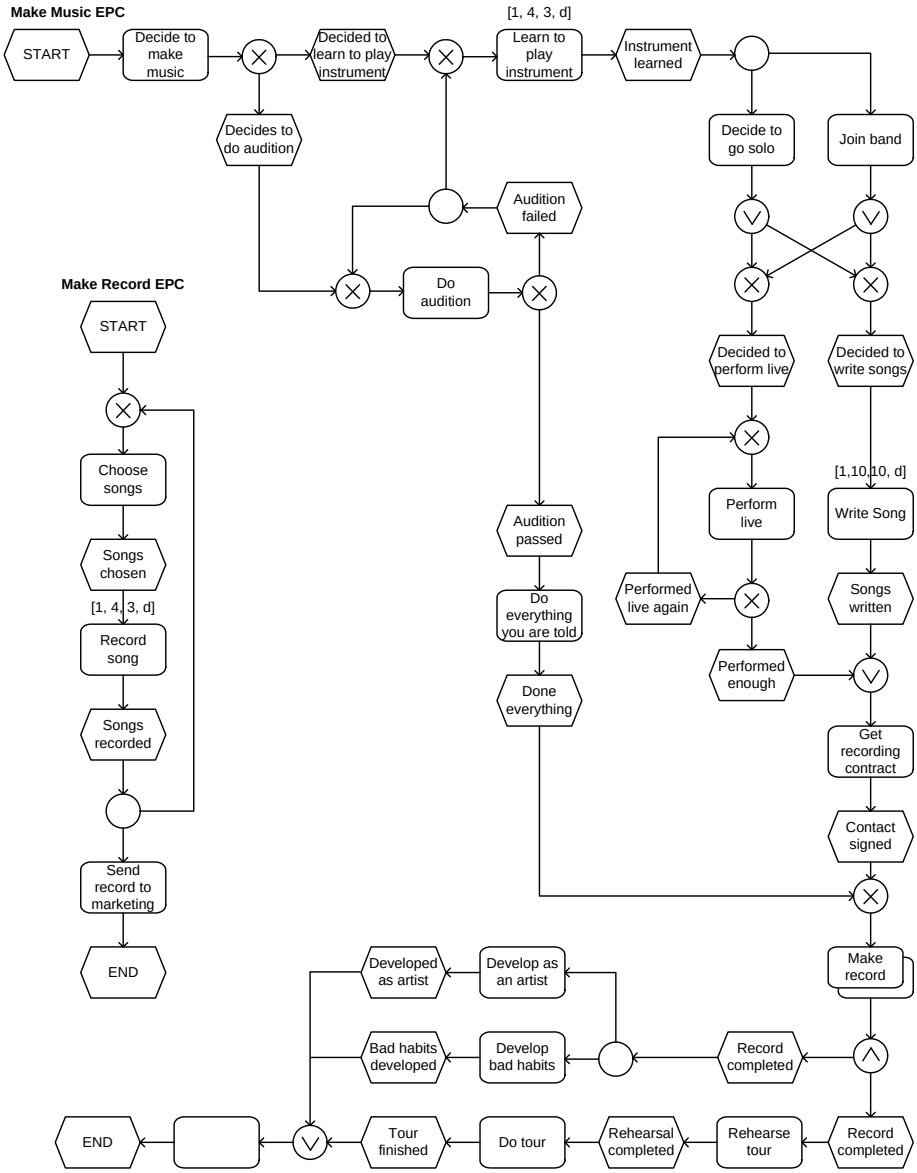


**Figure 2.** The Make Music Process from the YAWL website in yEPC Notation

*Cancellation* patterns have not yet been discussed for EPCs. We adopt the concept included in YAWL. Cancellation areas (symbolized by a lariat) may include several functions and activities. The end of the lariat has to be connected with a function. When this function completes, all functions and events in the lariat are cancelled.

Figure 2 illustrates yEPCs with the Make Music process from the YAWL website which was especially designed to capture various workflow patterns. The process starts with a start event. The following function leads to alternative events via an XOR split. The first alternative is to do an audition. If the artist passes the audition, she only has to follow the instructions of the producers in order to come up making a record. If the artist fails to pass the audition, she may try it again or decide to take the traditional path to become a professional musician by learning to play an instrument. The parameters of this function indicate multiple instantiation: at least one and at most four instruments have to be started learning; new instances may be created dynamically. Three instruments have to be learned in order to complete this function. Afterwards, the artist has to decide whether to go solo or to join a band. In either case she can perform live repeatedly and write songs which is modelled by an OR split. Afterwards, the artist gets a recording contract – actually, this determinism is in contrast with reality. After signing the contract the record is made following the Make Record subprocess. Subsequently, the tour is rehearsed and put through. In parallel, the musician develops as an artist or develops bad habits. Finally, these concurrent functions are synchronized by an OR join and the process is finished.

## 3 Workflow Pattern Analysis of EPCs

In this section we will consider the EPC control flow semantics of Kindler [10]. They basically reflect the ideas of [5,8]. These semantics have been implemented in the simulation tool EPC Tools [17]. For yEPC extensions, the following considerations have to be made. As Kindler places process folders (the EPC analogue to tokens of Perti nets) on arcs, the empty split may be interpreted as a hyperarc from the event before the empty split to the functions subsequent to it; the empty join analogously as a hyperarc from multiple functions before it to its subsequent event. For multiple instantiation and cancellation the concepts from YAWL are adopted. In the following we illustrate how yEPCs can be used to model the workflow patterns (WP) presented in [1]. In the following we will speak of EPCs each time we make statement that hold for both yEPCs and EPCs. Otherwise we will explicitly refer to yEPCs always when presenting concepts that are not included in EPCs.

*Workflow Pattern 1 (Sequence):* Figure 3 shows an EPC model of workflow pattern 1 (sequence). In EPCs each activity or task is modelled as a so-called *function*. Such functions are symbolized by rounded rectangles. Functions can be connected by so-called *events* symbolized as hexagons. Events represent prerequisites for a subsequent function, i.e., the event must have occurred before the

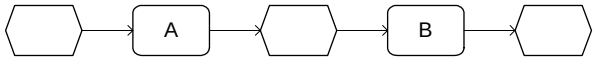


**Figure 3.** EPC model for WP1

following function may be executed. Furthermore, completed functions trigger events which may be pre-requisite for other functions. The alternation of events and functions defines a business process which also explains the name "Event-Driven" Process Chain (EPC). In Figure 3 function $A$ triggers an event which is the pre-requisite of function $B$ defining a sequence of activities as described by workflow pattern 1.

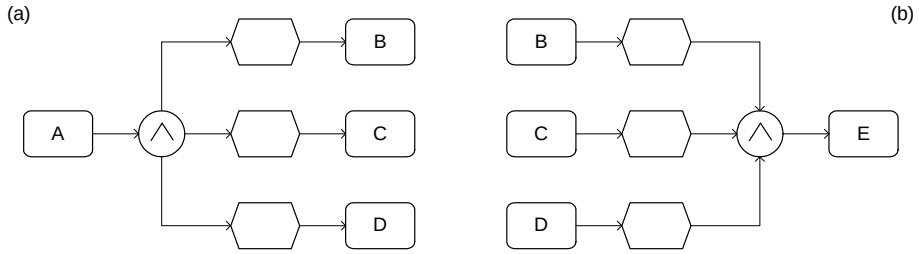


**Figure 4.** EPC model for (a) WP2 with AND split and (b) WP3 with AND join

*Workflow Pattern 2 (Parallel Split) and 3 (Synchronization):* EPCs define a restriction on the number of incoming and outgoing arcs of events and functions. Each function must have exactly one incoming and one outgoing arc, each event at most one incoming and one outgoing arc. In order to allow for complex routing of control flow so-called *connectors* are introduced. A connector may have one incoming and multiple outgoing arcs (split) or multiple incoming and one outgoing arc (join). Figure 4 (a) illustrates the application of an AND split connector to achieve control flow behavior as defined by workflow pattern 2 (parallel split). That means after function $A$ all the three subsequent functions $B$, $C$, and $D$ are activated to be executed concurrently. The connector is represented by a circle. The and-symbol $\wedge$ indicates its type. Connectors have no influence on the alternation of events and functions (see e.g. [8,18]). That means, for example, that an event is always followed by a function no matter if there are no, one, or more connectors between them. Figure 4 (b) shows the AND connector as a join. Each of the functions $B$, $C$, and $D$ have to be completed before $E$ can be executed. The AND join synchronizes the parallel threads of execution just as described by workflow pattern 3 (synchronization). The symbols for AND split and AND join are the same. They can only be distinguished by the cardinality of incoming and outgoing arcs.

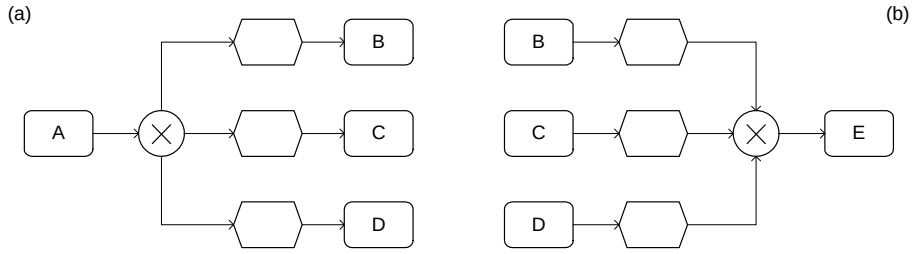


**Figure 5.** EPC model for (a) WP4 with XOR split and (b) for WP5 with XOR join

*Workflow Pattern 4 (Exclusive Choice) and 5 (Simple Merge):* Pattern 4 (exclusive choice) describes a point in a process where a decision is made to continue with one of multiple alternative branches. This situation can be modelled with the XOR split connector of EPCs, compare Figure 5 (a). After function $A$ has completed, a decision is taken to continue with one of functions $B$, $C$, and $D$. Figure 5 (b) shows the XOR join that precisely captures the semantics of pattern 5. There has been a debate on the non-local semantics of the XOR join. While Rittgen [7] and Van der Aalst [19] proposes a local interpretation, recent research agrees upon non-local semantics (see e.g. [8,20,17]). This means that the XOR join would only allowed to continue when one of the functions $B$, $C$, and $D$ has finished, and it is not possible that the other functions will ever be executed. Accordingly, EPC's XOR join works perfect when used in an XOR block started with an XOR split, but may block e.g. when used after an OR split depending on whether more than one branch has been activated. Regarding this non-local semantics it is similar to a synchronizing merge (see workflow pattern 7) but with the difference that it blocks when further process folders may be propagated to the XOR join.

In contrast to this, pattern 5 (simple merge) defines a merge without synchronization, but building on the assumption that the joined branches are mutually exclusive. The XOR join in YAWL [3] can implement such such behavior with local semantics: when one of parallel activities is completed the next activity after the XOR join is started. But when the assumption does not hold, i.e., when another of the parallel activities has finished the activity after the XOR join is activated another time, and so forth. This observation allows two conclusions. First, there is a fundamental difference between the semantics of the XOR join in EPCs and YAWL: the XOR join in EPCs has non-local semantics and blocks if there are multiple paths activated; the XOR join in YAWL has local semantics and propagates each incoming process token without ever blocking. Accordingly, the YAWL XOR join can also be used to implement pattern 8 (multiple merge). Second, as the XOR join in EPCs has non-local semantics, there is no mechanism available to model workflow pattern 8 with EPCs.

*Workflow Pattern 6 (Multiple Choice) and 7 (Synchronizing Merge):* Figure 6 (a) gives an EPC model of workflow pattern 6 (multiple choice) using the OR

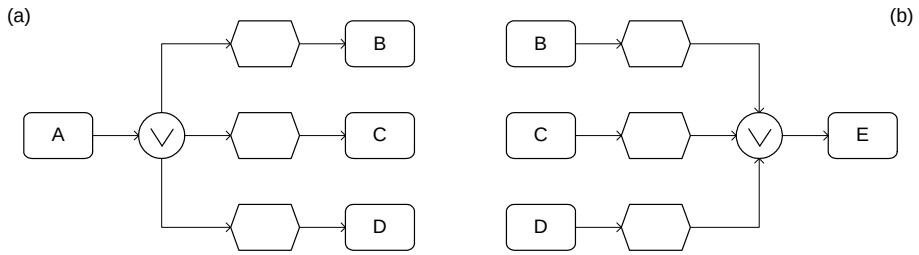


**Figure 6.** EPC model for (a) WP6 with OR split and (b) WP7 with OR join

split connector. This connector activates multiple branches based on conditions. The OR join connector depicted in Figure 6 (b) synchronizes multiple paths of execution as described in workflow pattern 7 (synchronizing merge). The OR join has both in EPCs and in YAWL non-local semantics. This means that function $E$ can only be executed when all concurrently activated branches have completed. This is different to workflow pattern 3 (synchronization) where all branches have to complete, no matter if they have been activated or not. Accordingly, the OR join in Figure 6 needs to consider not only if functions $B$, $C$, or $D$ have been completed, but also if there is the chance that they can potentially be activated in the future. If this is the case, the OR join has to wait until an execution of these functions is no longer possible or until they have completed.

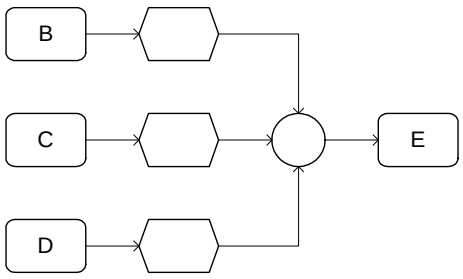


**Figure 7.** yEPC model for WP8

*Workflow Pattern 8 (Multiple Merge):* In the discussion on pattern 5 we have already highlighted the difference between the XOR join in YAWL and EPCs. As the XOR join in EPCs has non-local semantics, it is not suitable to implement the multiple merge, i.e., a situation where multiple concurrent branches are joined without synchronization. Figure 7 illustrates the empty connector to model a multiple merge. Accordingly, its semantics are similar to those of the YAWL XOR join. Yet, it needs to be mentioned that a design choice has to be made between a multi-set representation as described e.g. in [8] and a simple set representation

as specified in e.g. [20]. The multi-set variant would consume further process folders of $C$ and $D$ even if $B$ had been executed and $E$ not yet started. The simple set semantics would block incoming folders until the execution of $E$ had consumed the folder on the event.

*Workflow Pattern 9 (Discriminator):* For this pattern, the cancellation concept can be combined with the deferred choice to model the discriminator (workflow pattern 9). Figure 8 shows a respective model fragment. The functions $B$, $C$, and $D$ may be executed concurrently. When the first of them is completed the subsequent event is triggered. This allows function $E$ to start. The completion of $E$ leads to cancellation of all functions in the cancellation area that still might be active.

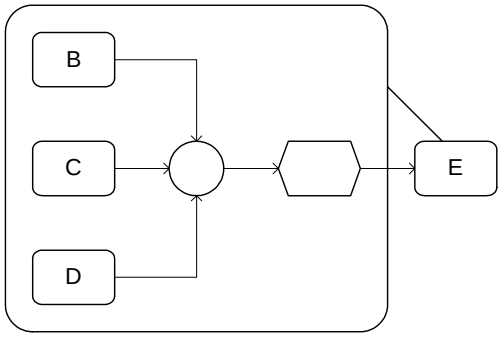


**Figure 8.** yEPC Model for WP9

*Workflow Pattern 10 (Arbitrary Cycles):* EPCs also provide for direct support of workflow patterns 10. Arbitrary cycles (workflow pattern 10) are explicitly allowed in EPCs. Yet, one needs to be aware that arbitrary cycles in conjunction with uncontrolled entrances via OR join or XOR join connectors may lead to EPC process models with so-called *unclean* semantics [20]. Furthermore, it is not allowed to have cycles composed of connectors only [8].

*Workflow Pattern 11 (Implicit Termination):* Implicit termination is also supported by EPCs [21]. Figure 9 gives the example of an EPC process fragment with multiple end events. EPCs do not terminate before all activities have completed or process folders are locked in non-local XOR joins or AND joins [21]. As a consequence, the model of Figure 9 is equivalent to a model that synchronizes these three end events with an OR join connector to only one new end event.

*Workflow Pattern 12 (Multiple Instantiation without Synchronization):* Figure 10 (a) shows a model fragment including a process interface. Process interfaces

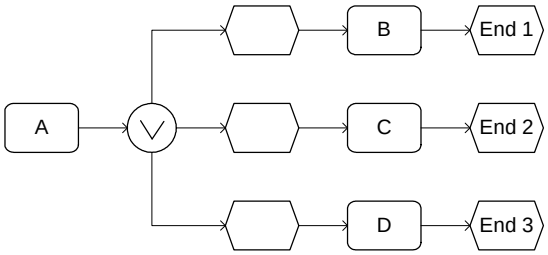


**Figure 9.** EPC Model for WP11 Implicit Termination

may be regarded as a short-hand notation for a hierarchical function that is followed by an end event. Figure 10 (b) illustrates how workflow pattern 12 (multiple instantiation without synchronization) can be modelled using a process interface. The parameters in square brackets indicate that the function may be instantiated multiple times. The parameters `min` and `max` define the minimum and maximum cardinality of instances; `required` specifies the amount of instances to be finished to complete multiple instantiation. Furthermore, `creation` specifies whether further instances may be created at run-time (`dynamic`) or not (`static`).

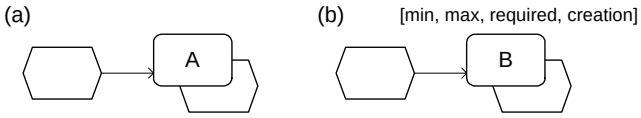


**Figure 10.** yEPC Model for WP12

*Workflow Pattern 13-15 (Multiple Instantiation with Synchronization)* Figure 11 (a) gives a model fragment of a simple function that may be instantiated multiple times (indicated by the parameters in square brackets). Figure 11 (b) shows a hierarchical function that supports multiple instantiation. In contrast to the process interface the multiple instances are synchronized and the subsequent event is not triggered before all instances have completed.

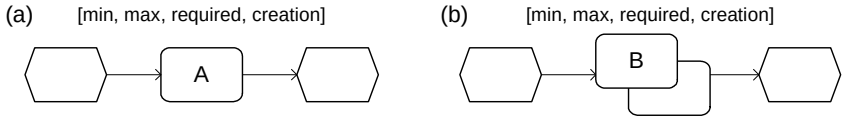


**Figure 11.** yEPC Model for WP13-15

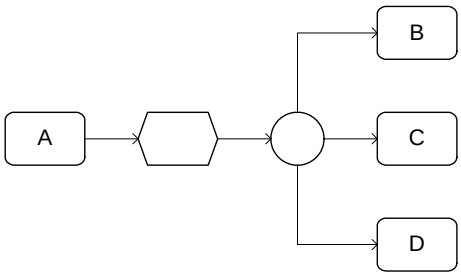


**Figure 12.** yEPC Model for WP16 Deferred Choice

*Workflow Pattern 16: Deferred Choice* Figure 12 (a) illustrates the application of the empty split connector to represent workflow pattern 16 (deferred choice): after function $A$ has completed, the subsequent event is triggered. The empty split represents that the subsequent functions may consume this event. Accordingly, the input pre-conditions of all three functions $B$, $C$, and $D$ are satisfied. Yet, the first of these functions to be activated consumes the event. As a consequence, the pre-conditions of the other functions no longer hold.

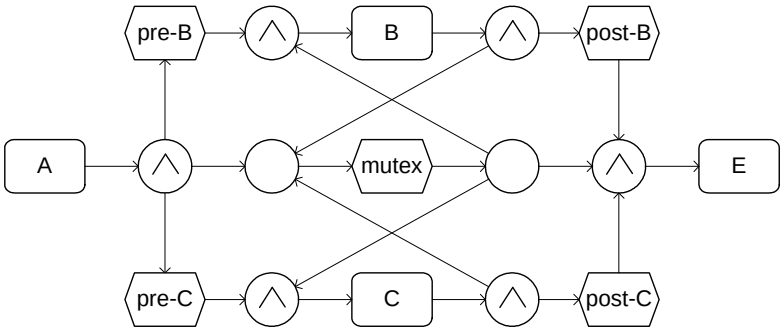


**Figure 13.** yEPC Model for WP 17 Interleaved Parallel Routing

*Workflow Pattern 17: Interleaved Parallel Routing* Empty connectors can also be used for other state-based workflow patterns. Figure 13 shows the process model of pattern 17 (interleaved parallel routing) following the ideas presented in [1]. The event at the center of the model manages the sequential execution of functions $B$ and $C$ in arbitrary order. It corresponds to the "mutual exclusion place (*mutex*)" introduced in [1]. The AND-split after function $A$ adds a folder to this *mutex* event via an empty connector. The AND-joins before the functions $B$ and $C$ consume this folder and put it back to the mutex event afterwards. Furthermore, they consume the individual folders in *pre-B* and *pre-C*, respectively. These events control that each function of $B$ and $C$ is executed only once. After

both have been executed, there are folders in *post-B*, *post-C*, and *mutex*. Accordingly, $E$ can be started. In [22] sequential split and join operators are proposed to describe control flow behavior of workflow pattern 17. Yet, it is no clear what the formal semantics of these operators would be when these operators are not used pairwise.

*Workflow Pattern 18: Milestone* Figure 14 shows the application of empty connectors for the modelling of workflow pattern 18. The event between $A$ and $B$ serves as a milestone for $D$. This means that $D$ can only be executed if $A$ has completed and $B$ has not yet started. This model exploits the newly introduced empty connector to model such behavior: if $B$ is started before $D$, the milestone is expired and $D$ can no longer be executed. If $D$ is started before $E$, a folder is put to the subsequent event to $D$ which implies that $B$ and $E$ can then be started. Thus, the introduction of the empty connector allows for a straight-forward modelling of workflow patterns 5 and 16 to 18.

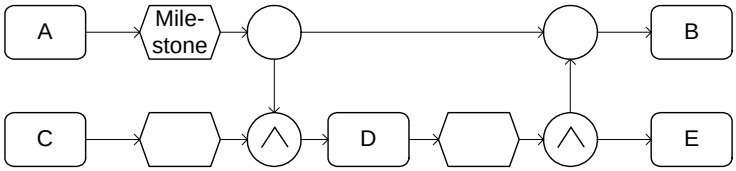


**Figure 14.** yEPC Model for WP 18 Milestone

*Workflow Pattern 19-20: Cancel Activity, Cancel Case* Cancellation is related to the workflow patterns 9, 19, and 20. We here adopt the concept that is used with the YAWL workflow language. Figure 15 shows the modelling notation of the cancellation concept. It specifies that when function $B$ has completed, function $A$ and the event is cancelled. This concept can further be used to implement workflow pattern 20, the cancellation of a whole case.

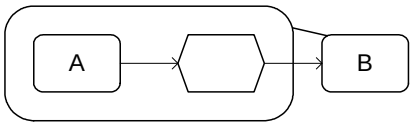


**Figure 15.** yEPC Model for WP19

Altogether, workflow patterns 1 to 7, 10, and 11 are supported by EPCs. In contrast yEPCs, provides additional modelling support of workflow patterns 5 (simple merge), 8 (multiple merge), 9 (discriminator), 12-15 (multiple instantiation), 16 (deferred choice), 17 (interleaved parallel routing), 18 (milestone), and

19-20 (cancellation). As a consequence, business processes including control flow behavior that is related to previously unsupported workflow patterns can now be represented appropriately using yEPCs.

## 4 EPML Alignment with yEPCs

In this section, we discuss in how far the proposed yEPC extensions may have an impact on the EPML representation. The EPC Markup Language (EPML) is an XML-based interchange format for EPC business process models proposed in [12]. In this section, we particularly want to identify which syntax elements need to be added to EPML in order to represent yEPCs.

```
<epml>
...
<epc epcId='1' name='example'>
     <function id='1'>
          <multiple
               minimum='3'
               maximum='6'
               required='4'
               creation='static'/>
     </function>
     <arc>
          <flow source='1' target='2'/>
     </arc>
     <empty id='2'/>
     <function id='3'>
          <cancel id='1'/>
          <cancel id='3'/>
          <cancel id='6'/>
     </function>
     ...
</epc>
</epml>
```

**Figure 16.** EPML Representation of multiple instantiation and cancellation

First, the introduction of the empty connector can be easily represented in the EPML schema. Figure 16 gives the example of an empty connector with an `id=2`. The arc indicates that it follows a function with `id=1`. Second, there are dedicated elements needed for multiple instantiation. Figure 16 gives an illustration of the required EPML elements. The `multiple` subelement indicates that the parent function or process interface can be instantiated multiple times. The four attributes capture the semantics of the parameter described above and defined in [3]. Third, the second function of Figure 16 shows how multiple `cancel` elements can be attached to a function or a process interface. Each cancel element carries an `id` attribute referencing the function, event, or process interface that

should be cancelled. These slight extensions show that EPML can easily aligned with the syntactical requirements of yEPCs.

## 5 Related Work

The workflow patterns proposed by [1] provide a comprehensive benchmark for comparing different process modelling languages. A short workflow pattern analysis of EPCs is also reported in [3], yet it does not discuss the non-local semantics of EPCs XOR join. In this paper, we highlighted these semantics as a major difference between YAWL and EPCs. Accordingly, we propose the introduction of the empty connector in order to capture workflow pattern 8 (multiple merge). There is further research discussing notational extensions to EPCs. In Rittgen [7] a so-called XORUND connector is proposed to partially resolve semantical problems of the XOR join connector. Motivated by space limitations of book pages and printouts, Keller and Teufel introduce process interfaces to link EPC models on different pages [11]. We adopt process interfaces in this paper to model spawning off of sub-processes. Rosemann [22] proposes the introduction of sequential split and join operators in order to capture the semantics of workflow pattern 17 (interleaved parallel routing). While the informal meaning of a pair of sequential split and join operators is clear, the formal semantics of each single operator is far from intuitive. As a consequence, we propose a state-based representation of interleaved parallel routing inspired by Petri nets. Furthermore, Rosemann introduces a connector that explicitly models a decision table and a so-called $OR_1$ connector to mark branches that are always executed [22]. Rodenhagen presents multiple instantiation as a missing feature of EPCs [14]. He proposes dedicated begin and end symbols to model that a branch of a process may be executed multiple times. Yet, this notation does not enforce that a begin symbol is followed by a matching end symbol. As a consequence, we adopt the multiple instantiation concept of YAWL that permits multiple instantiation only for single functions or sub-processes, but not for arbitrary branches of the process model.

## 6 Conclusion and Future Work

In this paper, we have presented a novel class of EPCs that is able to capture all 20 workflow patterns as presented in [1]. We refer to this extended class of EPCs as yEPCs, which is a tribute to YAWL [3]. Basically, yEPCs introduce three extensions to EPCs. These are in particular the introduction of the empty connector; the inclusion of a multiple instantiation concept for both simple functions as well as for hierarchical functions and process interfaces; and the inclusion of a cancellation concept. These extensions permit some conclusions on the relation of Petri nets and EPCs in general. Both had to include extensions for multiple instantiation and cancellation. In addition to this, Petri nets had to be extended with advanced synchronization concepts in order to capture the workflow patterns. On the other hand, EPCs had to be modified in order to address the

state-based workflow patterns. As a consequence, yEPCs and YAWL are quite similar concerning their modelling primitives. The XOR join is the major difference between both. Furthermore, we have shown that these extensions can be easily included in EPML. In future research, we aim to implement a transformation between yEPCs available in EPML format and the interchange format of YAWL.

## References


1. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Distributed and Parallel Databases **14** (2003) 5–51
2. Mendling, J., Nüttgens, M., Neumann, G.: A Comparison of XML Interchange Formats for Business Process Modelling. In: Proceedings of EMISA 2004 - Information Systems in E-Business and E-Government. LNI (2004)
3. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language. Information Systems **30** (2005) 245–275
4. van der Aalst, W.M.P.: Verification of Workflow Nets. In Azéma, P., Balbo, G., eds.: Application and Theory of Petri Nets 1997. Volume 1248 of Lecture Notes in Computer Science., Springer Verlag (1997) 407–426
5. Keller, G., Nüttgens, M., Scheer, A.W.: Semantische Prozessmodellierung auf der Grundlage "Ereignisgesteuerter Prozessketten (EPK)". Technical Report 89, Institut für Wirtschaftsinformatik Saarbrücken, Saarbrücken, Germany (1992)
6. Keller, G., Meinhardt, S.: SAP R/3 Analyzer. Business process reengineering based on the R/3 reference model. SAP AG (1994)
7. Rittgen, P.: Quo vadis EPK in ARIS? Ansätze zu syntaktischen Erweiterungen und einer formalen Semantik. WIRTSCHAFTSINFORMATIK **42** (2000) 27–35
8. Nüttgens, M., Rump, F.J.: Syntax und Semantik Ereignisgesteuerter Prozessketten (EPK). In J. Desel and M. Weske, ed.: Promise 2002 - Proceedings of the GI-Workshop, Potsdam, Germany. Volume 21 of Lecture Notes in Informatics. (2002) 64–77
9. van der Aalst, W.M.P., Desel, J., Kindler, E.: On the semantics of EPCs: A vicious circle. In M. Nüttgens and F. J. Rump, ed.: Proc. of the 1st GI-Workshop on Business Process Management with Event-Driven Process Chains (EPK 2002), Trier, Germany. (2002) 71–79
10. Kindler, E.: On the semantics of EPCs: Resolving the vicious circle. In J. Desel and B. Pernici and M. Weske, ed.: Business Process Management, 2nd International Conference, BPM 2004. Volume 3080 of Lecture Notes in Computer Science., Springer Verlag (2004) 82–97
11. Keller, G., Teufel, T.: SAP(R) R/3 Process Oriented Implementation: Iterative Process Prototyping. Addison-Wesley (1998)
12. Mendling, J., Nüttgens, M.: EPC Markup Language (EPML) - An XML-Based Interchange Format for Event-Driven Process Chains (EPC). Technical Report JM-2005-03-10, Vienna University of Economics and Business Administration, Austria (2005)
13. Arkin, A.: Business Process Modeling Language (BPML). Specification, BPMI.org (2002)
14. Rodenhagen, J.: Ereignisgesteuerte Prozessketten - Multi-Instantiierungsfähigkeit und referentielle Persistenz (Event-Driven Process Chains (EPC) - Multiple Instantiation and Referential Persistence - in German). In: Proceedings of the 1st GI

Workshop on Business Process Management with Event-Driven Process Chains. (2002) 95–107
15. Guabtni, A., Charoy, F.: Multiple Instantiation in a Dynamic Workflow Environment. In Persson, A., Stirna, J., eds.: Advanced Information Systems Engineering, 16th International Conference, CAiSE 2004. Volume 3084 of Lecture Notes in Computer Science., Springer-Verlag (2004) 175–188
16. Mendling, J., Strembeck, M., Neumann, G.: Extending BPEL4WS for Multiple Instantiation. In Dadam, P., Reichert, M., eds.: INFORMATIK 2004 - Band 2, Proceedings of the 34th Annual Meeting of German Informatics Society (GI), Workshop Geschäftsprozessorientierte Architekturen (GPA 2004). Volume 51 of Lecture Notes in Informatics., Gesellschaft für Informatik (2004) 524–529
17. Cuntz, N., Kindler, E.: On the semantics of EPCs: Efficient calculation and simulation. In: Proceedings of the 3rd GI Workshop on Business Process Management with Event-Driven Process Chains (EPK 2004). (2004) 7–26
18. Mendling, J., Nüttgens, M.: EPC Modelling based on Implicit Arc Types. In M. Godlevsky and S. W. Liddle and H. C. Mayr, ed.: Proc. of the 2nd International Conference on Information Systems Technology and its Applications (ISTA), Kharkiv, Ukraine. Volume 30 of Lecture Notes in Informatics. (2003) 131–142
19. van der Aalst, W.M.P.: Formalization and Verification of Event-driven Process Chains. Information and Software Technology **41** (1999) 639–650
20. Kindler, E.: On the semantics of EPCs: A framework for resolving the vicious circle (Extended Abstract). In M. Nüttgens, F. J. Rump, ed.: Proc. of the 2nd GI-Workshop on Business Process Management with Event-Driven Process Chains (EPK 2003), Bamberg, Germany. (2003) 7–18
21. Rump, F.J.: Geschäftsprozessmanagement auf der Basis ereignisgesteuerter Prozessketten - Formalisierung, Analyse und Ausführung von EPKs. Teubner Verlag (1999)
22. Rosemann, M.: Erstellung und Integration von Prozeßmodellen - Methodenspezifische Gestaltungsempfehlungen für die Informationsmodellierung. PhD thesis, Westfälische Wilhelms-Universität Münster (1995)