Using the π-calculus for Formalizing Workflow Patterns

Frank Puhlmann and Mathias Weske

Hasso-Plattner-Institute for IT Systems Engineering
at the University of Potsdam
D-14482 Potsdam, Germany
{puhlmann,weske}@hpi.uni-potsdam.de

Abstract. This paper discusses the application of a general process theory—the π-calculus—for describing the behavioral perspective of workflow. The π-calculus is a process algebra that describes mobile systems. Mobile systems are made up of components that communicate and change their structure as a result of communication. The ideas behind mobility, communication and change, can also enrich the workflow domain, where flexibility and reaction to change are main drivers. However, it has not yet been evaluated whether the π-calculus is actually appropriate to represent the behavioral patterns of workflow.
This paper investigates the issue and introduces a collection of workflow patterns formalizations, each with a sound formal definition and execution semantics. The formalizations can be used as a foundation for pattern-based workflow execution, reasoning, and simulation as well as a basis for future research on theoretical aspects of workflow.

1 Introduction

Recently, the π-calculus has been discussed as a formal foundation for workflow [1, 2]. The advocators of the so called Third Wave claim that the π-calculus is a natural foundation for workflow as it is based on communication and change. Indeed, communication is required for inter-organizational workflow and service oriented languages like BPML, XLang, or BPEL4WS [3–5]. The ability to dynamically change workflows on demand is already an important topic in workflow research [6–8]. Despite these discussions, no formal and reasonably complete investigation of the π-calculus regarding the workflow domain has been made. This paper takes a first step by analyzing the capabilities of the π-calculus regarding workflow patterns [9]. It introduces a collection of workflow patterns formalizations, each with an unambiguous formal definition and execution semantics.

The formalizations can be used in two major directions. First, they build a foundation for pattern-based workflow execution, reasoning, and simulation, which is based upon the execution semantics and proving capabilities of a formal algebra. Second, the formalizations show that the π-calculus is indeed a base for a precise definition of behavioral workflow requirements. At the same time it might

open the door for future research, i.e. integrating other workflow perspectives like organizational, operational, or informational [10, 11].

The remainder of this paper is organized as follows. Section 2 discusses related work. A brief introduction to the π-calculus is given in Section 3. Section 4 contains the formal definitions of workflow patterns; the main concepts are illustrated by examples. The paper is concluded with an outlook and directions for future work.

2 Related Work

Another approach of giving a detailed representation of the workflow patterns has been made with YAWL [12]. Starting as an endeavor as a workflow language of high expressiveness, YAWL has received considerable attention recently. The focus of YAWL is the convenient representation of all workflow patterns, as well as tool support and interfacing to various workflow tools. In the context of YAWL, a detailed representation of workflow patterns has been proposed [12]. As such, it is an important area of related work. However, the work presented in this paper aims at providing a broader exploitation and areas for future work, since the concepts provided by π-calculus allow for further representation, analysis, and reasoning, such as compliance of multiple processes.

From the context of process algebra, there has little been done for workflow purposes up to now. A Ph.D. thesis by Twan Basten researches basic process algebra and Petri nets [13]. A more practical approach of using CCS [14] to formalize web service choreography can be found in [15]. The only approach known to the authors on the use of the π-calculus for workflow definitions is from Yang Dong and Zhang Shen-Sheng and centers on basic control flow constructs and the definition of activities [16]. An approach close to process algebra is the logic based modeling and analysis of workflows by the use of concurrent transaction logic [17]. However, the expressiveness of this approach regarding to the workflow patterns has still to be investigated. Further approaches regarding the formalization of workflow patterns might include procedural techniques, which combine imperative, object-oriented and concurrent programming, logic–based attempts as well as graphgrammar- and net-based ones. Some approaches could be combined like the event-based and the process algebra has been used together in this paper.

3 The π-calculus

The π-calculus is a process algebra that describes mobile systems in a broader sense [18]. The calculus is based on the concept of mobility, which includes communication and change. Communication takes place between different π-calculus processes. The structure of the processes changes over time by communication e.g., a process can dynamically include other processes which he received through communication. The communication itself is based on the concept of names. A

name is a collective term for previous existing concepts like links, pointers, references, identifiers, etc., each of which has a scope. Assuming a name represents the reference to a process that currently processes a workflow activity, the scope of the name includes at that time only the active process. As soon as the process has finished, the scope is extruded to the process that handles the next workflow activity. Based on the flexibility of the π-calculus, which has only been sketched, many different possibilities arise to formalize the workflow patterns. We adopt an event, condition, action (ECA) approach, where each activity of a workflow is mapped conceptually to an independent π-calculus process. The processes use events in the form of communication to coordinate the behavior of a workflow. Several processes together form a behavioral pattern, which represents a workflow pattern.

Syntax

As several different notations of the π-calculus exist [18–21], the one used throughout this paper is outlined. Details can be found in [22].

Basically, the π-calculus consists of processes and names, where names define links. The processes are defined through:

$$P \;::=\; M \mid P|P \mid \mathbf{v}zP \mid !P \;.$$

The composition $P|P$ is the concurrent execution of P and P, $\mathbf{v}zP$ is the restriction of the scope of the name z to P, which is also used to generate a unique, fresh name z and $!P$ is the replication operator that satisfies the equation $!P = P \mid !P$. M contains the summations of the calculus:

$$M \;::=\; \mathbf{0} \mid \pi.P \mid M + M$$

where $\mathbf{0}$ is inaction, a process that can do nothing, $M+M$ is the exclusive choice between M and M', and the prefix $\pi.P$ is defined by:

$$\pi \;::=\; \overline{x}\,\langle y\rangle \mid x(z) \mid \tau \mid [x = y]\pi \;.$$

The output prefix $\overline{x}\,\langle y\rangle .P$ sends the name y over the name x and then continues as P. The input prefix $x(z)$ receives any name over x and then continues as P with z replaced by the received name (written as $\{^{name}/_{z}\}$). The unobservable prefix $\tau.P$ expresses an internal action of the process, and the match prefix $[x = y]\pi.P$ behaves as $\pi.P$, if x is equal to y.

Throughout this paper, upper case letters are used for process identifiers and lower case letters for names. Some additional process identifiers and names that represent special functions are introduced later on. Furthermore defined processes from the original paper on the π-calculus are used for parametric recursion, that is $A(y_1, ..., y_n)$ [18]. For the definitions given in this paper, defined processes are more applicable than the recent form with recursive definitions $K \triangleq (\widetilde{x}).P$ and constant applications $K\,\lfloor\widetilde{a}\rfloor$ where $\widetilde{x}$ and $\widetilde{a}$ represent sets of names [22].

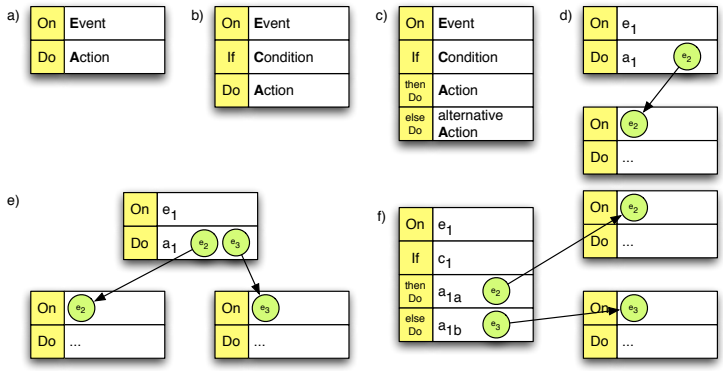


Fig. 1. The EA (a), ECA (b), and ECAA (c) notation for business rules and sequential (d), parallel (e), and optional (f) control flow.

We use the abbreviation $\sum_1^m(M)$ to denote the summation of m choices; e.g. $\sum_1^3(M_i) = M_1 + M_2 + M_3$. The abbreviation $\prod_1^m(P)$ is used to denote the composition of m parallel copies of P, e.g. $\prod_1^3(P) = P \mid P \mid P$. Also, $\{\pi\}_1^m$ denotes m subsequent executions of π, e.g. $\{\pi\}_1^3 = \pi.\pi.\pi$. All abbreviations could be used with an indexing variable, e.g. $\prod_{i=1}^3(d_i(x)) = d_1(x) \mid d_2(x) \mid d_3(x)$. Round brackets are used to define the ordering of a process definition. Given $\tau.P$ for instance, P might be expanded to $M + M'$ by using the summation rule from the π-calculus grammar. To avoid ambiguity, round brackets are put around the expanded symbol, e.g. $\tau.(M + M')$ instead of $\tau.M + M'$.

4 Pattern Representation

The formalization of the workflow patterns in the π-calculus starts with a mapping from activities to π-calculus processes.[1] Let every activity be an independent process. Each process has pre- and postconditions. A precondition for a process B could be that it should only start working after a process A has finished. A postcondition for process B could state that B has completed execution and then signals this to other processes.

The core idea is based on the ECA approach that originates from active database systems. ECA means *Event, Condition, Action* [23]. The event component specifies when a rule must be evaluated. After the rule has been evaluated, the conditional component must be checked and if it matches, the action component is executed. This approach has been adapted to specify control flow between different activities in a workflow [24]. The adapted paradigm is called $\mathrm{EC}^m\mathrm{A}^n$. It allows m conditions and n actions. In the workflow domain ECAA, ECA and EA

[1] We abbreviate the term *π-calculus process* to *process* in the following.

rules are most common (see figure 1). The figure also shows sequential, parallel, and optional control flow.

We can map the ECA approach to process definitions. The preconditions of the processes comply to the event and conditional part of the ECA rules. Every process that has no event part represents an initially starting activity, as the process has no further dependencies. The events are modeled in the process definitions as input prefixes. After the input prefixes have been triggered (that is, the event has occurred) an optional condition has to be checked. This is modeled by a match prefix. It can be used to model global constraints like testing a cancellation flag. The action part is divided into two parts. First, the functional perspective of the activity is represented as an unobservable action. Second, the process can trigger other processes by output prefixes. Output prefixes represent postconditions. If a process does not trigger other processes it represents a final workflow activity. The complete process definition for a basic activity is:

$$x.[a = b].\tau.\overline{y}.\mathbf{0} \ . \tag{1}$$

A process receives a trigger x mapping to an event, makes a comparison $[a = b]$ mapping to a condition, does some internal work τ and finally triggers another process with $\overline{y}$ as the resulting action. This notation can be generalized to:

$$\{x_i\}_{i=1}^{m}.\{[a = b]\}_1^n.\tau.\{\overline{y_i}\}_{i=1}^{o}.\mathbf{0} \ . \tag{2}$$

A generic process can have m incoming triggers, n conditions, and o outgoing triggers. A process that represents an activity must have a functional part represented by τ. Note that it is explicitly allowed to have zero incoming triggers, conditions or outgoing triggers. The consequences have been discussed earlier. If a process representing an activity can be triggered more than once, the replication operator must be used.

The description given applies only to basic control flow structures. Advanced structures require slightly different approaches. Additionally to the processes that represent the workflow activities, system and helper processes are required. These processes do not belong directly to the workflow, but are needed for reasoning and execution control.

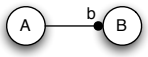
The patterns given in the next paragraphs can be seen as small pieces of a workflow definitions. The postconditions of the processes that link to other processes are indicated by a process identifier with an apostrophe, like the process A has process A' as a postcondition. The process A' might represent any other workflow pattern.

4.1 Basic Control Flow Patterns

The basic control flow patterns capture elementary aspects of workflow control flow. They are structured like shown in equation 2.

Sequence. A sequence between two processes A and B is achieved by A sending a name over b to process B, which executes τ_B and afterward activates the continuation as B':

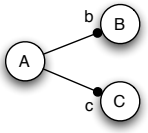
$$A = \tau_A.\overline{b}.\mathbf{0}$$
$$B = b.\tau_B.B'$$



As can be seen, the actual process definition transmits no name, because the name is irrelevant for triggering another process. We abbreviate $\overline{b}\langle x\rangle \mid b(x).B'$ to $\overline{b} \mid b.B'$, when the argument count is zero. As explained earlier, this is called triggering.

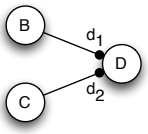
Parallel Split. To achieve a parallel split from a process A to two processes B and C, A triggers two names b and c at the processes B and C.

$$A = \tau_A.(\overline{b}.\mathbf{0} \mid \overline{c}.\mathbf{0})$$
$$B = b.\tau_B.B'$$
$$C = c.\tau_C.C'$$



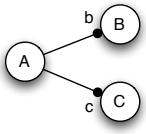
Synchronization. The synchronization between two processes B and C at another process D is represented by B and C each triggering the names d_1 or d_2 at D. The process D waits on those two names until it can continue as D'.

$$B = \tau_B.\overline{d_1}.\mathbf{0}$$
$$C = \tau_C.\overline{d_2}.\mathbf{0}$$
$$D = d_1.d_2.\tau_D.D'$$



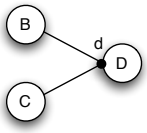
Exclusive Choice. The exclusive choice between two alternative processes B or C after A is modeled by the π-calculus summation operator. Thereby A triggers either b or c.

$$A = \tau_A.(\overline{b}.\mathbf{0} + \overline{c}.\mathbf{0})$$
$$B = b.\tau_B.B'$$
$$C = c.\tau_C.C'$$



Simple Merge. The simple merge of two control flows from either processes B or C in D is achieved by B and C triggering a name d. Per definition of this pattern, B and C will never be executed in parallel, so D only needs to wait on one incoming name d. If B and C should be executable in parallel, the synchronizing merge pattern applies.

$$B = \tau_B.\overline{d}.\mathbf{0}$$
$$C = \tau_C.\overline{d}.\mathbf{0}$$
$$D = d.\tau_D.D'$$

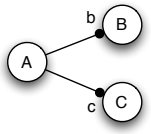


4.2 Advanced Branching and Synchronization Patterns

This section covers advanced branching and synchronization patterns. They require advanced concepts and map only partly to equation 2. One pattern, the synchronizing merge, needs to know the number of incoming flows that depend on preceding multi–choices. However, this is only important at the execution level. At the design level considered here, all possibilities must be captured.

Multi–choice. The choice between processes B or C or B and C after A is modeled by A having three possibilities of execution. Either A triggers B or C or both, B and C.

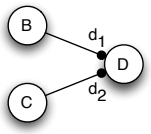
$$\begin{aligned} A = &\ (\mathbf{v}exec)\tau_A.(A_1 \mid A_2) \\ A_1 = &\ \overline{exec}\,\langle b\rangle\,.\mathbf{0} + \\ &\ \overline{exec}\,\langle c\rangle\,.\mathbf{0} + \\ &\ \overline{exec}\,\langle b\rangle\,.\overline{exec}\,\langle c\rangle\,.\mathbf{0}) \\ A_2 = &\ !exec(x).\overline{x}.\mathbf{0} \\ B = &\ b.\tau_B.B' \\ C = &\ c.\tau_C.C' \end{aligned}$$



Note that this pattern uses the concept of an executor ($exec$) represented by process A_2. An executor receives a name and afterward triggers that name. The executor always immediately responds and decouples the triggering of the subject received in a parallel thread, thus not blocking the original caller. The executor workaround is needed, because a process $\overline{b}.\mathbf{0} + \overline{c}.\mathbf{0} + (\overline{b}.\mathbf{0} \mid \overline{c}.\mathbf{0})$ cannot be derived from the π-calculus grammar given. If we just specify $\overline{b}.\overline{c}.\mathbf{0}$ to denote that both names, b and c should be triggered, the semantic is incorrect. For example image a more complicated construct. The preconditions of process B are extended so that he has to wait additionally on a name b_1. This could be written as $B = b_1.b.\tau_B.B'$. The name b_1 has not yet been triggered, so the process $\overline{b}.\overline{c}.\mathbf{0}$ could not yet trigger the name c. Process C that only has c as a precondition cannot start execution. This is clearly not the intention of the multi–choice pattern.

Synchronizing Merge. The triggers for activating a process D can either come from B or C as well as from B and C. If B and C are executed in parallel, D has to wait on d_1 and d_2, otherwise only for d_1 or d_2.

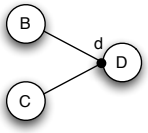
$$\begin{aligned} B = &\ \tau_B.\overline{d_1}.\mathbf{0} \\ C = &\ \tau_C.\overline{d_2}.\mathbf{0} \\ D = &\ d_1.\tau_D.D' + \\ &\ d_2.\tau_D.D' + \\ &\ d_1.d_2.\tau_D.D' \end{aligned}$$



This pattern has no synchronization problem. Even if process C is able to signal d_2 earlier than B can signal d_1, the process C is blocked until B has signaled d_1. This confirms with the reduction rules of the π-calculus. Note that the semantics of this pattern does not describe how a runtime actually decides which summation of D is chosen.

Multi-merge. Process D can be triggered arbitrary times by incoming triggers from B or C. Each time D gets triggered, a new copy of D is created by replication.

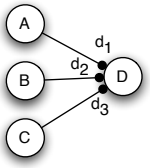
$$B = \tau_B.\overline{d}.\mathbf{0}$$
$$C = \tau_C.\overline{d}.\mathbf{0}$$
$$D = !d.\tau_D.D'$$



Note that by using the replication operator to create multiple copies of a process D, all processes that are triggered by D must also support replication and so on. This also refers to all other patterns that create multiple copies by replication.

Discriminator. The discriminator pattern activates τ_D by triggering D_2 if process D_1 receives either d_1, d_2 or d_3. After D_2 has activated τ_D it waits for the triggers h from the remaining incoming branches of D_1. Finally D_2 resets the discriminator by using recursion.

$$A = \tau_A.\overline{d_1}.\mathbf{0}$$
$$B = \tau_B.\overline{d_2}.\mathbf{0}$$
$$C = \tau_C.\overline{d_3}.\mathbf{0}$$
$$D = (\mathbf{v}h, exec)(D_1 \mid D_2)$$
$$D_1 = d_1.\overline{h}.\mathbf{0} \mid d_2.\overline{h}.\mathbf{0} \mid d_3.\overline{h}.\mathbf{0}$$
$$D_2 = h.\overline{exec}.h.h.D \mid exec.\tau_D.D'$$



The process definitions A, B and C are trivial. The process definition D that represents the discriminator is split into two parts D_1 and D_2 with two fresh names h and *exec*. D_1 waits in parallel for all incoming triggers d_1, d_2 and d_3. If a trigger is received, D_1 anonymizes the trigger by signaling h to D_2. Afterward process D_1 waits for the remaining triggers. If another process signals a name that D_1 has already received, the signaling process is blocked. The process D_2 waits for an incoming name h and afterward executes τ_D in parallel, achieved through an internal trigger *exec*. This is needed due to the decoupling of the subsequent actions represented by D'. Afterward it waits for the remaining triggers from D_1 and then resets itself by the use of recursion. Note that all processes that are called by D' must have the capability of multiple execution.

A generic discriminator with m incoming control triggers is defined by:

$$D = (\mathbf{v}h, exec)((\prod_{i=1}^{m} d_i.\overline{h}.\mathbf{0}) \mid h.\overline{exec}.\{h\}_1^{m-1}.D \mid exec.\tau_D.D').$$

The generic discriminator uses the product operator $\prod$ from 1 to m to denote m different incoming triggers. After receiving the first h trigger, it uses the sequence operators $\{\}$ to wait on $m-1$ anonymized incoming triggers. Those operators are just notational sugar; they have to be expanded before the process can be executed.

Example: Discriminator. To illustrate the discriminator, one possible evolution[2] of the system defined by A, B, C, D_1 and D_2 is given:

$$DISC = A \mid B \mid C \mid (\mathbf{v}h, exec)(D_1 \mid D_2) \ .$$

The processes are defined initially as given in the discriminator paragraph. The evolution of $DISC$ begins with either A, B or C signaling a name to D_1. We start with A signaling name d_1 to process D_1:

$$DISC \longrightarrow DISC_1 = B \mid C \mid (\mathbf{v}h, exec)(D_{11} \mid D_2) \ .$$

The process A has vanished as no more prefixes other than $\mathbf{0}$ exist after signaling the name d_1. The process D_1 has evolved to D_{11} and is defined by $D_{11} = \overline{h}.\mathbf{0} \mid d_2.\overline{h}.\mathbf{0} \mid d_3.\overline{h}.\mathbf{0}$. Immediately after, a communication between D_{11} and D_2 is possible:

$$DISC_1 \longrightarrow DISC_2 = B \mid C \mid (\mathbf{v}h, exec)(D_{12} \mid D_{21}) \ .$$

D_{11} signals the name h to D_2 and evolves to $D_{12} = d_2.\overline{h}.\mathbf{0} \mid d_3.\overline{h}.\mathbf{0}$. The left hand component has vanished as it reached inaction. The process D_2 evolves to $D_{21} = \overline{exec}.h.h.D \mid exec.\tau_D.D'$. Now $exec$ can be triggered inside D_{21}:

$$DISC_2 \longrightarrow DISC_3 = B \mid C \mid (\mathbf{v}h, exec)(D_{12} \mid D_{22}) \ .$$

D_{22} is given by $D_{22} = h.h.D \mid D'$. Note that the right hand side of D_{22} now only consists of D'. We can assume D' to be $\mathbf{0}$ in our example. So the right hand side of D_{22} vanishes. Now process B can trigger d_2 and D_{12} can trigger h:

$$DISC_2 \longrightarrow DISC_3 = C \mid (\mathbf{v}h, exec)(D_{13} \mid D_{23}) \ .$$

Process B vanishes after triggering d_2. D_{12} evolves to $D_{13} = d_3.\overline{h}.\mathbf{0}$. Process D_{23} is given by $D_{23} \overline{\overline{h}}.D$. Finally process C can trigger d_3:

$$DISC_3 \longrightarrow DISC_4 = D \ .$$

Process C vanishes after triggering d_3. D_{13} vanishes after receiving d_3 and triggering h. The only process now left is D which resets the discriminator through recursion. To make the discriminator work another time, we need new processes that trigger d_1, d_2 and d_3 again. A, B and C could also declared replicative, e.g. $A = !\tau_A.\overline{d_1}.\mathbf{0}$, etc. We could further trace other evolutions of the system described, e.g. starting with d_2 or d_3.

N-out-of-M-Join. The n-out-of-m join generalizes the discriminator by executing the activity τ_D after n out of m triggers have arrived at process D. After the remaining triggers have been received, D resets itself by recursion.

$$D = (\mathbf{v}h, exec)((\prod_{i=1}^{m} d_i.\overline{h}.\mathbf{0}) \mid \{h\}_1^n.\overline{exec}.\{h\}_{n+1}^m.D \mid exec.\tau_D.D')$$

The n-out-of-m-join simply expands the middle expression of the generic discriminator by waiting for m incoming triggers in a sequence. The remaining triggers are then counted from $n+1$ to m.

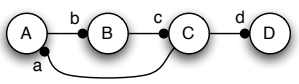
[2] We use the π-calculus semantics of reduction for this example, see [22].

4.3 Structural Patterns

Structural patterns show restrictions on workflow languages, as for instance that arbitrary loop are not allowed or that only one final node should be present. The π-calculus easily handles both of the following patterns.

Arbitrary Cycles. Arbitrary cycles are inherently given by the event based approach. The only thing that must be taken care of is the re–instantiation of processes that execute repeatedly.

$$
\begin{aligned}
A &= !a.\tau_A.\overline{b}.\mathbf{0} \\
B &= !b.\tau_B.\overline{c}.\mathbf{0} \\
C &= !c.\tau_C(\overline{a}.\mathbf{0} + \overline{d}.\mathbf{0}) \\
D &= d.\tau_D.D'
\end{aligned}
$$



The re-instantiation is modeled using the replication operator for all processes that could be executed more than once (A, B, C). Process C must decide if the loop is called another time by triggering a or to continue by triggering d. If arbitrary cycles are allowed in a workflow definition, the formal reasoning will be much more difficult.

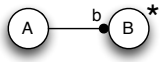
Implicit Termination. The implicit termination pattern terminates a sub–process if no other activities can be made active. The π-calculus contains the special symbol $\mathbf{0}$ for this purpose. As $\mathbf{0}$ is the only final termination symbol of the π-calculus grammar, each (sub)–process must finally have an implicit termination.

4.4 Multiple Instance Patterns

Multiple instance patterns create several copies of workflow activities. A trivial pattern uses no synchronization whereas more advanced patterns synchronize the created copies afterward.

Multiple Instances without Synchronization. Any amount of multiple copies of a process B can easily spawn from a process A by replication.

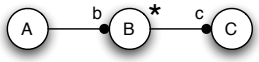
$$
\begin{aligned}
A &= \tau_A.!\overline{b}.\mathbf{0} \\
B &= !b.\tau_B.B'
\end{aligned}
$$



A recursive definition for A could be $A = \tau_A.A_1$ with $A_1 = \overline{b}.A_1 + \mathbf{0}$. This notation explicitly states that A_1 can spawn of new copies of B or stop execution.

Multiple Instances with a priori Design Time Knowledge. When the number of copies of B is known at design time and the copies have to be synchronized before the execution of τ_C, the following pattern is used (the example shows three copies of B).

$$A = \tau_A.\overline{b}.\overline{b}.\overline{b}.\mathbf{0}$$
$$B = !b.\tau_B.\overline{c}.\mathbf{0}$$
$$C = c.c.c.\tau_C.C'$$

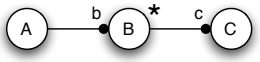


For n design time copies, the pattern is as follows:

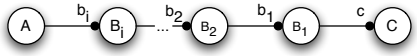
$$A \mid B \mid C \equiv \tau_A.\{\overline{b}\}_1^n.\mathbf{0} \mid !b.\tau_B.\overline{c}.\mathbf{0} \mid \{c\}_1^n.\tau_C.C' \ .$$

Multiple Instances with a priori Runtime Knowledge. This pattern is runtime dependent like the synchronizing merge. At design time it can be modeled that A can spawn of an unknown number of processes B and only after A has finished creating the processes, τ_B gets activated by receiving a *start* trigger each. After all copies of B have finished, the name initially passed to A is triggered. The pattern needs a fresh name *start* private to A and B to work: $(\mathbf{v}start)(A \mid B)$. Note that this pattern uses defined processes for recursion.

$$A = (\mathbf{v}run)\tau_A.A_1(c) \mid run.!\overline{start}.\mathbf{0}$$
$$A_1(x) = (\mathbf{v}y)\overline{b}\,\langle y\rangle\,.y\,\langle x\rangle\,.A_1(y) + \overline{run}.\overline{x}.\mathbf{0}$$
$$B = !b(y).y(x).start.\tau_B.y.\overline{x}.\mathbf{0}$$
$$C = c.\tau_C.C'$$



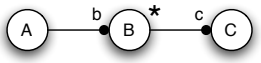
This pattern works like a dynamic linked list:



Initially A holds the name of the next process, that is c. An arbitrary number of processes B can be inserted between A and C using recursion. The created copies are started by triggering *run* in A which results in triggering *start* in all copies of B. Each copy of B triggers his predecessor after finishing τ_B. The initial predecessor is passed as a parameter to A; it is the name of the trigger that is activated after all copies of B have successfully executed τ_B. This pattern is a special case of the multiple instances without a priori runtime knowledge; an example is given later on.

Multiple Instances without a priori Runtime Knowledge. This pattern is much the same as the preceding one, with the difference that copies of B could be created all the time and start immediately.

$$A = \tau_A.A_1(c)$$
$$A_1(x) = (\mathbf{v}y)\overline{b}\,\langle y\rangle\,.y\,\langle x\rangle\,.A_1(y) + \overline{x}.\mathbf{0}$$
$$B = !b(y).y(x).\tau_B.y.\overline{x}.\mathbf{0}$$
$$C = c.\tau_C.C'$$



The only difference is the removal of the *start* and *run* triggers as well as the depending process parts.

Example: Multiple Instances without a priori Runtime Knowledge. We derive a trace of the multiple instances without a priori runtime knowledge pattern. The example shows how the recursive structure of the processes is build up while creating instances and how it is broken down while completing.

The process A is initialized with the link to the process that should be executed after all copies of B have been completed. That is c in our case:

$$A = \tau_A.A_1(c) \ .$$

Process A calls process A_1 with c as a parameter. A_1 has the choice between creating a copy of the process B or call the final process, that is c:

$$A_1(c) = (\mathbf{v}y)\overline{b}\,\langle y\rangle\,.y\,\langle c\rangle\,.A_1(y) + \overline{c}.\mathbf{0} \ .$$

We suppose process A_1 to create a new copy of process B. Therefore the left part of A_1 is executed: $(\mathbf{v}y_1)\overline{b}\,\langle y_1\rangle\,.y_1\,\langle c\rangle\,.A_1(y_1)$. First, a fresh name y_1 is generated. We enumerate y with a subscript to mark different fresh names. The name y_1 is sent to process B which creates a new copy of itself through replication. Afterward, A_1 sends the name of the predecessor (that is c) to the new copy of process B. A_1 then calls itself with the fresh name y_1 as a parameter. Thereby the fresh name y_1 acts as the new predecessor. The processes A_1 and B now look like:

$$\begin{aligned} A_1(y_1) &= (\mathbf{v}y_2)\overline{b}\,\langle y_2\rangle\,.y_2\,\langle y_1\rangle\,.A_1(y_2) + \overline{y_1}.\mathbf{0} \\ B &= !b(y).y(x).\tau_B.y.\overline{x}.\mathbf{0} \mid \underbrace{\tau_B.y_1.\overline{c}.\mathbf{0}}_{\text{1st copy}} \ . \end{aligned}$$

The process A_1 now has again the choice between creating a new copy of the process B or call the previous created process by y_1. Note that the 1st copy of B is already executing τ_B. We choose to create yet another copy of B:

$$\begin{aligned} A_1(y_2) &= (\mathbf{v}y_3)\overline{b}\,\langle y_3\rangle\,.y_3\,\langle y_2\rangle\,.A_1(y_3) + \overline{y_2}.\mathbf{0} \\ B &= !b(y).y(x).\tau_B.y.\overline{x}.\mathbf{0} \mid \underbrace{\tau_B.y_1.\overline{c}.\mathbf{0}}_{\text{1st copy}} \mid \underbrace{\tau_B.y_2.\overline{y_1}.\mathbf{0}}_{\text{2nd copy}} \ . \end{aligned}$$

This is continued until A_1 decides to call the previous fresh name that was created; that is the parameter of A_1. In our example this is y_2. We suppose the τ_B of the copies of B to have been finished by now. So a communication between A_1 and B by y_2 could take place; otherwise we had to wait until the τ_B of the second copy has finished:

$$\begin{aligned} A_1(y_2) &= \mathbf{0} \\ B &= !b(y).y(x).\tau_B.y.\overline{x}.\mathbf{0} \mid \underbrace{y_1.\overline{c}.\mathbf{0}}_{\text{1st copy}} \mid \underbrace{\overline{y_1}.\mathbf{0}}_{\text{2nd copy}} \ . \end{aligned}$$

Now the second copy of B has reduced to $\overline{y_1}.\mathbf{0}$. A communication between the second and first copy by y_1 is now possible:

$$\begin{aligned} A_1(y_2) &= \mathbf{0} \\ B &= !b(y).y(x).\tau_B.y.\overline{x}.\mathbf{0} \mid \underbrace{\overline{c}.\mathbf{0}}_{\text{1st copy}} \mid \underbrace{\mathbf{0}}_{\text{2nd copy}} \ . \end{aligned}$$

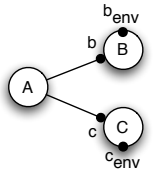
The second copy of B has reached inaction. The first copy can trigger the name c that references to the process that should be executed after all copies of B have finished. Thereafter the first copy reaches inaction. If we suppose A_1 as the only source of names b than no further communication is possible. The multiple instances without a priori runtime knowledge pattern is completed.

4.5 State Based Patterns

State based patterns capture implicit behavior of processes that is not based on the current case rather than the environment or other parts of the process. Some of the following patterns require the existence of an external process that represents the environment. This process is used as a source for external events. We denote the environmental process with the special process identifier $\mathcal{E}$. The names that are triggered from within $\mathcal{E}$ are marked with a subscripted env, as for instance a_{env} denotes an environmental trigger.

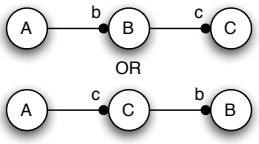
Deferred Choice. A deferred choice is much like the exclusive choice with the distinction that the choice if τ_B or τ_C get executed is not made explicit in A rather than by the environment. The environment is modeled as an external process $\mathcal{E}$ that signals either the name b_{env} or c_{env} but not both. The moment of choice is thereby as late as possible. Afterward the successful process signals the name $kill$ to the other process which leads to the empty process $\mathbf{0}$. B and C must share a fresh name $kill$: $(\mathbf{v}kill)(B \mid C)$

$$A = \tau_A.(\overline{b}.\mathbf{0} \mid \overline{c}.\mathbf{0})$$
$$B = b.(b_{env}.\overline{kill}.\tau_B.B' + kill.\mathbf{0})$$
$$C = c.(c_{env}.\overline{kill}.\tau_C.C' + kill.\mathbf{0})$$



Interleaved Parallel Routing. The interleaved parallel routing or unordered set is achieved by non–determinism in the π-calculus. A, B and C share two fresh names $(\mathbf{v}x, y)(A \mid B \mid C)$ of which x is used to trigger B and C in any order. The name y is used to signal the complete execution of the triggered process. After all activities have been executed, the control is again at A.

$$A = \tau_A.\overline{x}.y.\overline{x}.y.A'$$
$$B = x.\tau_B.\overline{y}.\mathbf{0}$$
$$C = x.\tau_C.\overline{y}.\mathbf{0}$$



Milestone. A milestone is a test for a process A, if another parallel process B is in a given state. Thereby the two parallel processes share a private name *check* which returns either true (represented by the special name $\top$) if the condition holds or false ($\bot$) if not. A process definition $M(x)$ is used as a memory cell that keeps the condition. It is called by a private name m with $(\mathbf{v}m)(B)$:

$$
\begin{aligned}
A =& check(x).([x = \top]\tau_{A1}.A' + [x = \bot]\tau_{A2}.A'') \\
B =& M(\bot) \mid b.\overline{m}\,\langle\top\rangle\,.\tau_B.\overline{m}\,\langle\bot\rangle\,.B' \\
M(x) =& m(x).M(x) + \overline{check}\,\langle x\rangle\,.M(x) \quad .
\end{aligned}
$$

4.6 Cancellation Patterns

The cancellation pattern describe the withdrawal of one or more processes that represent workflow activities.

Cancel Activity. The cancel activity pattern allows a process, that is waiting to get triggered, to be canceled. This pattern is modeled by the optional reception of a *cancel* trigger from an external environment process $\mathcal{E}$ with $(\mathbf{v}cancel)(A \mid \mathcal{E})$:

$$A \mid \mathcal{E} \equiv a.\tau_A.A' + cancel.\mathbf{0} \mid !\tau_{\mathcal{E}}.\overline{cancel}.\mathbf{0}\ .$$

Note that currently executed activities represented by τ could not be canceled due to the unobservability of τ.

Cancel Case. The cancel case pattern cancels a whole workflow instance. This is equal to Cancel Activity with the exception that all remaining processes receive a global cancel trigger.

5 Conclusion

In this paper, we introduced a formal semantics for workflow patterns, which is based on the π-calculus. All of the documented workflow patterns from [9] have been formalized with concise and unambiguous expressions. Based on the execution semantics of the π-calculus, the behavior of each workflow pattern has been defined precisely.

However, this paper is not to be understood as *the* formal semantics of the workflow patterns. Other notations, like Workflow Nets [25] or YAWL [12] use different approaches from Petri nets to transition systems to realize a formal specified behavior for some or all of the workflow patterns. Rather, this paper can be seen as a foundation for using modern process algebra in the workflow domain. The π-calculus supports mobility, communication and change. While it has not yet been shown how mobility can actually enrich the workflow domain, requirements like flexibility and reaction to change are ever more challenging [1]. Since the π-calculus was designed to model such highly dynamic systems, it might offer new ways to face the challenges in the workflow domain. As a starting point, this paper showed that the π-calculus is indeed able to handle all of the behavioral workflow requirements given by workflow patterns.

Based on the formalizations presented in this paper, further research has to be made. The π-calculus could be used as a formal foundation for graphical notations. Furthermore, formalized workflows can opening the door for reasoning on workflow process structures.

References

1. Smith, H., Fingar, P.: Business Process Management – The Third Wave. Meghan-Kiffer Press, Tampa (2002)
2. van der Aalst, W.M.P.: Pi calculus versus petri nets: Let us eat "humble pie" rather than further inflate the "pi hype". (`http://is.tm.tue.nl/research/patterns/download/pi-hype.pdf` (May 31, 2005))
3. BPMI.org: Business Process Modeling Language. Technical report (2002)
4. Microsoft: XLang Web Services for Business Process Design. (2001)
5. BEA Systems, IBM, Microsoft, SAP, Siebel Systems: Business Process Execution Language for Web Services Version 1.1. (2003)
6. van der Aalst, W.: Flexible Workflow Management Systems: An Approach based on Generic Process Models. In Bench-Capon, T., Soda, G., Tjoa, A., eds.: Database and Expert Systems Applications: 10th International Conference, DEXA'99, volume 1677 of LNCS, Berlin, Springer (1999) 186–195
7. van der Aalst, W.M.P.: Exterminating the Dynamic Change Bug: A Concrete Approach to Support Workflow Change. Information System Frontiers **3** (2001) 297–317
8. Rinderle, S., Reichert, M., Dadam, P.: Evaluation of Correctness Criteria for Dynamic Workflow Changes. In van der Aalst, W.e.a., ed.: Business Process Management 2003, volume 2678 of LNCS, Berlin, Springer (2003) 41–57
9. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.: Workflow patterns. Distributed and Parallel Databases **14** (2003) 5–51
10. Curtis, B., Kellner, M.I., Over, J.: Process Modeling. Communications of the ACM **35** (1992) 75–90
11. Weske, M.: Workflow Management Systems: Formal Foundation, Conceptual Design, Implementation Aspects. Habilitationsschrift, Fachbereich Mathematik und Informatik, Universität Münster, Münster (2000)
12. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language (Revised version. Technical Report FIT-TR-2003-04, Queensland University of Technology, Brisbane (2003)
13. Basten, T.: In Terms of Nets: System Design with Petri Nets and Process Algebra. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands (1998)
14. Milner, R.: Communication and Concurrency. Prentice Hall, New York (1989)
15. Brogi, A., Canal, C., E.Pimentel, Vallecillo, A.: Formalizing Web Service Choreographies. In: Proceedings of First International Workshop on Web Services and Formal Methods. Electronic Notes in Theoretical Computer Science, Elsevier (2004)
16. Dong, Y., Shen-Sheng, Z.: Approach for workflow modeling using π-calculus. Journal of Zhejiang University Science **4** (2003) 643–650
17. Davulcu, H., Kifer, M., Ramakrishnan, C.R., Ramakrishnan, I.V.: Logic Based Modeling and Analysis of Workflows. In: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, ACM Press (1998) 25–33
18. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, Part I/II. Information and Computation **100** (1992) 1–77
19. Milner, R.: The polyadic π–Calculus: A tutorial. In Bauer, F.L., Brauer, W., Schwichtenberg, H., eds.: Logic and Algebra of Specification, Berlin, Springer-Verlag (1993) 203–246

20. Milner, R.: Communicating and Mobile Systems: The π-calculus. Cambridge University Press, Cambridge (1999)
21. Parrow, J.: An Introduction to the π–Calculus. In Bergstra, J.A., Ponse, A., Smolka, S.A., eds.: Handbook of Process Algebra, Elsevier (2001) 479–543
22. Sangiorgi, D., Walker, D.: The π-calculus: A Theory of Mobile Processes. Paperback edn. Cambridge University Press, Cambridge (2003)
23. Dayal, U., Hsu, M., Ladin, R.: Organizing long-running activities with triggers and transactions. In: Proceedings of the 1990 ACM SIGMOD international conference on Management of data, New York, ACM Press (1990) 204–214
24. Knolmayer, G., Endl, R., Pfahrer, M.: Modeling Processes and Workflows by Business Rules. In Aalst, W.v.d., Desel, J., Oberweis, A., eds.: Business Process Management: Models, Techniques, and Empirical Studies, volume 1806 of LNCS, Berlin, Springer-Verlag (2000) 16–29
25. van der Aalst, W., van Hee, K.: Workflow Management. MIT Press (2002)