Query Nets: Interacting Workflow Modules that Ensure Global Termination*

Rob J. van Glabbeek and David G. Stork

Ricoh Innovations
2882 Sand Hill Rd. Suite 115
Menlo Park, CA 94025-7022, USA
rvg@cs.stanford.edu
stork@rii.ricoh.com

Abstract. We address cross-organizational workflows, such as document workflows, which consist of multiple workflow modules each of which can interact with others by sending and receiving messages. Our goal is to guarantee that the global workflow network has properties such as termination while merely requiring properties that can be checked locally in individual modules. The resulting *query nets* are based on predicate/transition Petri nets and implement formal constructs for business rules, thereby ensuring such global termination. Our method does not require the notion of a global specification, as employed by Kindler, Martens and Reisig.

Introduction

This paper deals with the formal modeling of business processes that transform an input into an output by performing several tasks, or by delegating these tasks to other such business processes. Examples are an insurance company, that takes as input a claim and yields as output a decision on that claim, or a car dealership that takes as input a broken car and yields as output a repaired one. Both businesses follow a pre-established protocol to transform input into output. Such a protocol is called a *workflow*.

In this paper we focus on workflows that span several organizations. Each organization employs a local workflow, and these local workflows may delegate tasks to local workflows of other organizations. This is done by presenting an input to the other organization's workflow, awaiting the generation of an output, and proceeding with the task at hand using that output. (This paradigm is essentially the *Nested Subprocesses Model*, the second interoperability scenario identified by the Workflow Management Coalition [8].) Naturally, our work also applies to modular workflows within one organization.

Take as example the car dealership. The car dealership's workflow is initiated by a customer dropping off a broken car. In case the car dealership can repair the car, that's what they do; otherwise the problem is described to the car's manufacturer in Detroit, and their answer will be used to carry out the repair. The repaired car constitutes the output of the workflow; it can be picked up by the customer. The manufacturer's workflow can be initiated by a question about the repair of a car, and its output is an answer to that question. Some questions are dealt with by asking the local dealership that is supposed to carry out the repair, and using the answer of the dealership as output.

* To appear in: Wil M.P. van der Aalst, editor: Proceedings *Business Process Management 2003*, Eindhoven, The Netherlands, LNCS, Springer, June 2003.

In this paper we allow workflows that can deal with different types of input and output. The dealership's workflow for instance can also accept a client shopping for a car as input, in which case the output may be a sale. Or it can take anyone's question on how to repair a car as input, and present an answer to that question as output. In the latter case, difficult questions may be dealt with by asking the manufacturer.

The described workflows of the car dealership and the manufacturer are graphically displayed in Fig. 1. In Section 1 we will formalize this representation as a Petri net.

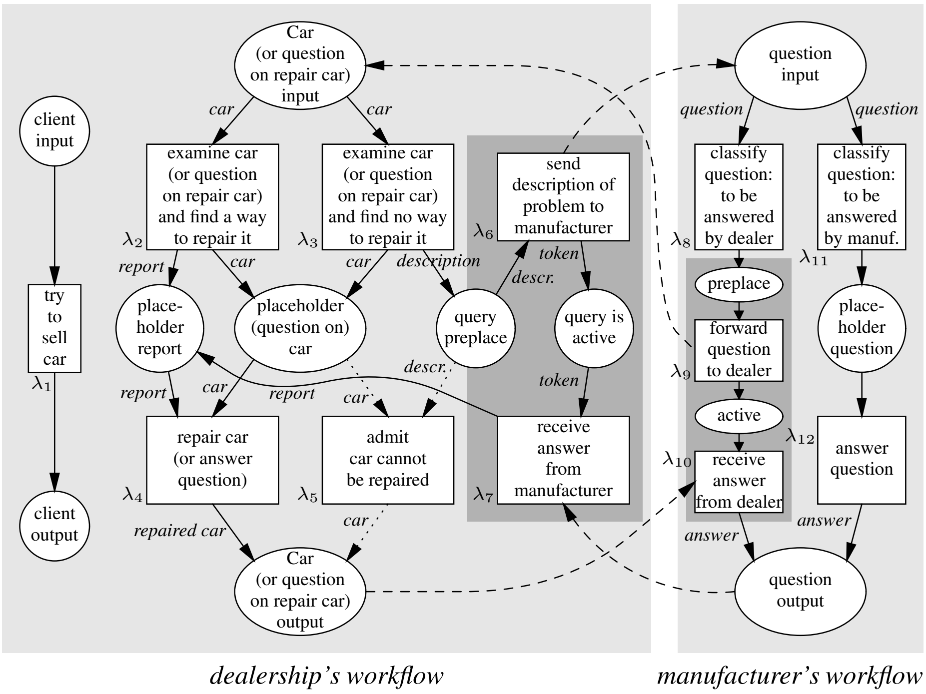

Fig. 1. *Petri net representation of the car dealership's workflow and the manufacturer's workflow, as well as their interaction (the dashed arcs). The dealership accepts as input a client shopping for a car, a broken car, or a question on the repair of a car. As the latter two types of input are treated similarly, they are collected in the same input place.*
Faced with an input of a car, the first task of the dealership is to examine it. Depending on whether the exam yields a way to repair the car or not, the output of this task is either the car and a report on how to fix the car, or the car and a description of the problem for which no fix has been found. In Petri nets, tasks are represented by transitions (the boxes) whose output types are fixed, so the examination needs to be modeled by two transitions.
The shaded part of the dealership's workflow deals with the querying of the manufacturer. With dotted lines we added the alternative of returning the car to the customer unrepaired.

An important property of a workflow is *termination*. On any given input the workflow should eventually produce an output and stop. Let's continue the example of the dealership. If a repair does not succeed, the workflow may prescribe to try to repair the car again, possibly using another method. When faced with a car that can never be fixed, such a workflow leads to an unending series of attempts to repair the car, and the process will not terminate. This is unacceptable behavior. When all reasonable methods have been tried, the dealership should give up and return the car to the customer unrepaired. Workflows should embody protocols to avoid infinite loops and ensure termination.

When the management of the car dealership inspects its workflow to see if termination is ensured, it has little information on the workflow used by the manufacturer, which is called as a subroutine. Yet, if the manufacturer's workflow fails to terminate, so will the dealership's workflow. Given the workflow sketched above, and the absence of time-outs or other tricks to ensure termination of the invocation of the manufacturer's workflow, the best that can be ensured is a conditional termination property. The car dealership's workflow may be shown to terminate on the condition that the manufacturer's workflow does. We call this *local termination*.

It is in the car dealership's interest to ensure *global* or unconditional termination of its workflow. This can be achieved by verifying that the car dealership's workflow is locally terminating, and the manufacturer's workflow is unconditionally terminating. When the car dealership trusts the manufacturer enough, the verification of the termination of the manufacturer's workflow may happen locally by the manufacturer, and the verdict taken on faith by the car dealership. However, verifying global termination of the manufacturer's workflow may not be so easy. It could be that the manufacturer delegates tasks to others, and those might delegate subtasks even further. In general we are dealing with a network of local workflows, and it may be verified locally that each local workflow in the network is locally terminating, i.e., terminating on the condition that all other workflows in the network are terminating. The question now arises whether such local termination properties of the local workflows are sufficient to guarantee termination of the global workflow.

This needs not always be the case. In could for instance be that the dealership receives as input a car that they do not know how to repair. According to protocol, they ask the manufacturer how to deal with this specific problem. The manufacturer on the other hand may have classified this question as one that ought to be answered by the local dealership. According to protocol, the manufacturer's workflow will pose the question to the local dealership that asked it in the first place. Strictly following protocol, the dealership now deals with the question by asking the manufacturer (again), and an infinite loop results. Thus the car dropped of at the dealership will never be repaired and the global workflow fails to terminate.

In practice, an infinite loop as sketched above will be prevented by some *business rule*. The dealership will never ask the same question twice to the manufacturer in the course of dealing with the same repair. Once the manufacturer echoes back the dealership's original question, the dealership either asks a different question to the manufacturer—a question that doesn't lend itself to being thoughtlessly echoed back to the dealership—or it solves the problem at hand without involving the manufacturer.

The current paper presents a framework for specifying cross-organizational workflows in which such business rules can be implemented. It introduces the concept of a *query net*, which is a locally terminating workflow module that, when embedded in a cross-organizational workflow, will never delegate the same task twice to another workflow module. We establish that in cross-organizational workflow networks built out of such query nets global termination is ensured.

In order to ensure global termination of a cross-organizational workflow network, all that is required is that the individual workflow modules in the network are query nets, a requirement that can be checked locally for each of these modules. Unlike in the work of Kindler, Martens and Reisig [6] there is no need to show correctness of the modules with respect to a *fairness-closed specification*, specifying the interactions between all modules in the global network.

1 Cross-organizational workflow nets

This section presents a Petri net model for cross-organizational workflow. For the purpose of establishing notation and terminology we start out by reviewing basic *place/transition* Petri nets. Then we present a form of the more powerful *predicate/transition nets*, which regard tokens as structured entities. Finally, we arrive at our model by equipping these nets with input and output places and a novel mechanism and implicit protocol for one net to call another as a subroutine.

1.1 Place/transition nets

A *place/transition net* is a tuple (S, T, F) where S and T are disjoint sets of *places* (*Stellen* in German) and *transitions*, and $F\colon\ (S \times T) \cup (T \times S)\ \rightarrow\ \mathbb{N}$ is the *flow relation*. The elements of S and T are represented graphically by circles and boxes, respectively. For $p, q \in S \cup T$ there are $F(p, q)$ *arcs* from x to y.

When a place/transition net represents a concurrent system, a global state of such a system is given as a *marking* $M\colon S \rightarrow \mathbb{N}$. Such a state is depicted by placing $M(s)$ *tokens* in each place s. A *marked place/transition net* is a tuple (S, T, F, M) comprising a place/transition net (S, T, F) and a marking M.

For two markings M and $M'\colon S \rightarrow \mathbb{N}$ we write $M \subseteq M'$ if $M(s) \leq M'(s)$ for all $s \in S$. The marking $M + M'\colon S \rightarrow \mathbb{N}$ is given by $(M + M')(s) = M(s) + M'(s)$. The function $M - M'\colon\ S\ \rightarrow\ \mathbb{Z}$ is given by $(M - M')(s)\ =\ M(s) - M'(s)$; this function need not yield a marking because it might specify a negative number of tokens in a place.

The multisets of *preplaces* ${}^\bullet t$ and *postplaces* $t^\bullet\colon\ S\ \rightarrow\ \mathbb{N}$ of a transition $t \in T$ in a place/transition net are given by ${}^\bullet t(s)\ =\ F(s, t)$ and $t^\bullet(s)\ =\ F(t, s)$ for $s \in S$. A transition t is *enabled* under a marking M, written $M[t\rangle$, if ${}^\bullet t \subseteq M$. In that case t can *fire* under M, yielding the marking $M' = M - {}^\bullet t + t^\bullet$, written $M[t\rangle M'$.

If a transition t fires, for every arc from a place s to t, a token moves along that arc from s to t. These tokens are consumed during the firing, but also new tokens are created, namely one for every outgoing arc of t. These new tokens end up in the places at the end of those arcs. The firing of t is possible only if there are sufficiently many tokens in the preplaces of t.

1.2 Predicate/transition nets

A *predicate/transition net* [3] (sometimes called *coloured Petri net* [5]) is a Petri net in which the tokens are structured entities. We use a set of *variables* $V\ =\ \{x, y, \ldots\}$ to range over the possible tokens in such a net, and a suitable collection $\mathbb{F}$ of formulas over those variables, expressing combinations of properties of the tokens referenced by the variables occurring in those formulas. The specification of $\mathbb{F}$ varies with the application.

A predicate/transition net is given as a quadruple (S, T, F, λ) where, as above, S and T are disjoint sets of places and transitions, but now the flow relation F is a subset of $(S \times V \times T) \cup (T \times V \times S)$, and λ: $T \rightarrow \mathbb{F}$ allocates to each transition a formula called the *transition guard* [7]. As above, the elements of S and T are represented graphically by circles and boxes, respectively, while an element $(p, x, q) \in F$ is represented as an *arc* from p to q, labeled with variable x. A formula $\lambda(t)$ is written next to the box representing the transition t.

An arc $(s, x, t) \in F$ with $s \in S$, $x \in V$ and $t \in T$ indicates that upon firing the transition t, a token x is taken from place s. An arc $(t, y, s') \in F$ with $t \in T$, $y \in V$ and $s' \in S$ indicates that upon firing t a token y is deposited in place s'. The transition guard $\lambda(t)$ selects properties of the input tokens that have to be satisfied for the transition to fire, and simultaneously specifies the relation between the input and the output tokens. The formula $\lambda(t)$ may contain free occurrences of the variables allocated to the arcs leading to or from t. These variables refer to the transition's input and output tokens. Consider as an example a transition that consumes input tokens x and y, and produces an output token z. The transition guard could then be a formula that says that the transition may only fire if y is a cryptographic key, and x is a PGP document that successfully decrypts with that key; if these conditions are met, the decrypted document z is emitted.

The Petri net of Fig. 1 is a predicate/transition net. The arc-labels such as "car" and "report" are variables. The transition guard λ_2 says that the variable "car" refers to a car for which a repair can be found, or alternatively to a question on the repair of a car for which a repair can be found; if this condition is not satisfied, the λ_2-labeled transition cannot fire. Moreover, λ_2 specifies the relation between the car and the report on how to fix it. The same variable "car" labels both an input and an output arc of the transition; this indicates that the car or question passes through this transition unchanged.

1.3 Marked predicate/transition nets and the firing rule

A framework for predicate/transition nets specifies a set V of variables, a collection $\mathbb{F}$ of formulas over V, a domain $\mathcal{D}$ of possible tokens, and an *evaluation function*. The latter specifies, for each formula $\varphi \in \mathbb{F}$ and each assignment ξ: $V \rightarrow \mathcal{D}$ of tokens to variables, whether φ evaluates to `true` of `false` under ξ; let $\varphi[\xi]$ denote the result of this evaluation. Any specification of a predicate/transition net presupposes such a framework.

A *marking* of a predicate/transition net is an allocation of tokens to the places of the net, formally defined as a function M: $S \times \mathcal{D} \rightarrow \mathbb{N}$, that specifies for every place $s \in S$ and token $d \in \mathcal{D}$ how many copies of d reside in s. A *marked* predicate/transition net is a tuple (S,T,F,λ,M) comprising a predicate/transition net (S,T,F,λ) and a marking M.

In a marked predicate/transition net a transition t can fire if there is an assignment ξ of tokens to variables such that the transition guard $\lambda(t)$ evaluates to `true`, and for every x-labeled arc from a place s to t there is an *input* token $\xi(x)$ in s. As a result of firing t, these input tokens are taken away and for every y-labeled arc from t to a place s' an *output* token $\xi(y)$ is deposited in s'.

Here is a more formal description of the firing rule, where the notions $M \subseteq M'$, $M + M'$ and $M - M'$ are defined just as for markings with only one argument. For a transition $t \in T$ in a predicate/transition net and an assignment $\xi\colon V \to \mathcal{D}$, the *input* and *output* markings ${}^\bullet t[\xi]$ and $t[\xi]^\bullet : S \times \mathcal{D} \to \mathbb{N}$ of t under ξ are given by

$$^\bullet t[\xi] = \{\!\!\{(s, \xi(x)) \mid (s, x, t) \in F\}\!\!\} \qquad \text{and} \qquad t[\xi]^\bullet = \{\!\!\{(s, \xi(x)) \mid (t, x, s) \in F\}\!\!\}$$

in which $\{\!\!\{, \}\!\!\}$ are multiset brackets. A transition t is *enabled* under a marking M, written $M[t\rangle$, if there exists an assignment $\xi\colon V \to \mathcal{D}$ such that $\lambda(t)[\xi]$ is $\texttt{true}$ and ${}^\bullet t[\xi] \subseteq M$. In that case t can *fire* under M, yielding the marking $M' = M - {}^\bullet t[\xi] + t[\xi]^\bullet$, written $M[t\rangle M'$.

1.4 Fairness

A *firing sequence* in a Petri net is a possibly infinite alternating sequence of markings and transitions $M_0, t_1, M_1, t_2, M_2, \ldots$ such that $M_0[t_1\rangle M_1[t_2\rangle M_2 \cdots$. We say that marking M' is *reachable* from marking M if there is a firing sequence starting with M and ending with M'.

In a Petri net, complete runs of the represented system are represented by firing sequences. However, not every firing sequence represents a complete run. Some finite firing sequences merely represent partial runs, and some infinite ones do not correspond with runs that could occur in practice. Those firing sequences that do model complete runs are called *fair*. Fairness can be formalized in many ways, often requiring a more involved definition of a Petri net, and often depending on certain *progress* or *fairness* assumptions made on the behavior of nets. In our simple nets we call a firing sequence *fair* if there is no transition t such that from a certain marking in the sequence onwards, t is continuously enabled but never fires. In particular, a finite firing sequence is fair if and only if in its last marking no transition is enabled. For a more subtle approach to fairness see for example Kindler, Martens & Reisig [6].

1.5 Workflow nets

Workflow nets are defined in van der Aalst [1, 2] employing place/transition nets. Here we extend the definition to predicate/transition nets. A *workflow net* (S, T, F, λ, i, o) is a predicate/transition net (S, T, F, λ) with two special places, i and o. A workflow net represents a business process that converts an input into an output. The input and output are represented by tokens, presented by the environment to the net's *input place* i, and removed from its *output place* o. The specification of a workflow nets does not involve the notion of an *initial marking*; a workflow net starts out empty, and may start firing using tokens dropped in the input place i by the environment of the net.

1.6 Multi-organizational workflow nets

In this paper we view the parallel composition of any number of business processes as a single global business process. Imagine two shops, each with their own procedures for handing input and output. While we generally consider them as two separate business processes, we might choose instead to treat them as a single global business process, perhaps because of joint ownership or some other binding relation between the two. When representing processes as Petri nets, their parallel composition is represented by the disjoint union of these nets. In the case of workflow nets, this representation leads to a proliferation of input and output places. Therefore we will consider workflow nets with multiple pairs of input and output places. These will also be suitable to represent business processes that can deal with different types of input that are treated in different ways and lead to different types of output.

A *multi-organizational workflow net* $W = (S, T, F, \lambda, IO)$ is a predicate/transition net (S, T, F, λ) equipped with a set IO of *input/output ports*, each port $p \in IO$ consisting of an *input place* $p_{in} \in S$ and an *output place* $p_{out} \in S$.

Each of the two workflows of Fig. 1, not including the dashed arcs between them, can be regarded as a multi-organizational workflow net. The dealership's workflow has two input/output ports, namely "client" and "car (or question on repair car)".

1.7 Workflow modules

We define a *workflow module* $(S, T, F, \lambda, IO, Q)$ to be a multi-organizational workflow net equipped with a set Q of *query ports*. Each query port $q \in Q$ consists of two transitions $q_?$ and $q_! \in T$ and two places q_{pre} and $q_{active} \in S$, connected by arcs $(q_{pre}, x_1, q_?), (q_?, x_2, q_{active})$ and $(q_{active}, x_3, q_!) \in F$. There are no other arcs to or from q_{active}. The arcs leading to q_{pre} should be such that q_{pre} receives at most one token for each input received by the workflow module. In some cases this is achieved by allowing only arcs to q_{pre} from transitions that have an input place p_{in} with $p \in IO$ as preplace (Fig. 2).

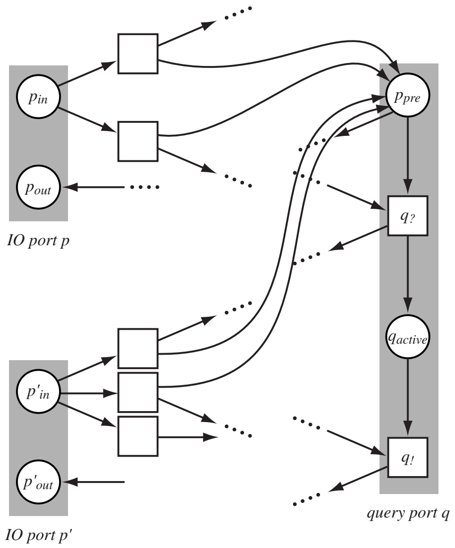

Fig. 2. *A query port in a workflow module. The query port q consists of a query transition $q_?$ sending a query to another workflow module, a query transition $q_!$ receiving the answer from the other module, a query place q_{pre} ensuring that in each run of the workflow module the query transition $q_?$ can be fired at most once, and a query place q_{active} ensuring that $q_?$ fires before $q_!$.*

Workflow modules represent business processes with the ability to invoke other business processes as subroutines. Each query port $q \in Q$ models (a) the invocation of another business process (the *target process* of the port) by sending it an input, and (b) the receipt of the corresponding output. The query transition $q_?$ represents the act of invoking the target process. The transition guard $\lambda(q_?)$ may contain free occurrences of the variable $x_?$. This variable represents the input that is presented to the target process. The query transition $q_!$ represents the receipt of an output from the target process. The transition guard $\lambda(q_!)$ may contain free occurrences of the variable $x_!$, referring to that output. This guard may relate $x_!$ with variables labeling outgoing arcs of $q_!$; however, it is not allowed to restrict the enabling of $q_!$ based on the value of $x_!$.

The query place q_{pre} ensures that in each run of the workflow started by single token in an input place, the query transition $q_?$ can fire at most once. In order to model a business process that in one such run invokes another business process twice as a subroutine, one needs two different query ports. When the target process has been invoked, but its output has not yet been collected, the query place q_{active} contains a token. This design ensures that $q_!$ can fire only after the firing of $q_?$.

1.8 Modular workflow architectures

A *modular workflow architecture* is a finite set of workflow modules, together with an allocation, to each query port in each module in the architecture, of an input/output port of another (or the same) workflow module (Fig. 3). The input/output port allocated to a query port is called the *target* of that query port. We assume that query ports in different modules in the architecture have different names.

A modular workflow architecture models a collection of interconnected business processes. Each of these business processes is represented by a workflow module. Occasionally such a business process delegates a subtask to another business process. This delegation is modeled by the query ports. The target of a query port indicates to which business process the subtask is delegated. A business process may dynamically choose a module to which a task should be delegated (e.g. in vendor selection) by routing control through a chosen query port.

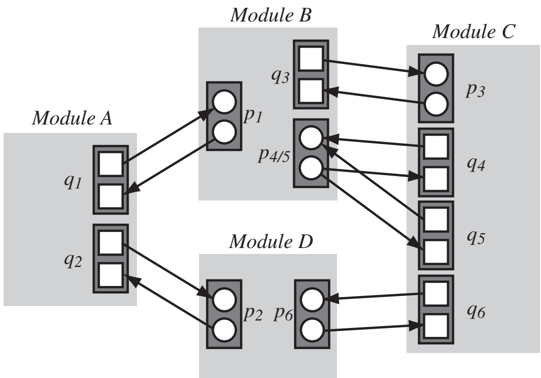

Fig. 3. *A modular workflow architecture consisting of four workflow modules with six query ports. The query ports q_4 and q_5 share the same target $p_{4/5}$.*

A modular network architecture can be represented by a single multi-organizational workflow net, called the *cross-organizational workflow net* representing the architecture. This is done by taking the disjoint union of its constituent workflow modules, and connecting every query port q in every module, through arcs $(q_?, x_?, p_{in})$ and $(p_{out}, x_!, q_!)$, with the input and output places p_{in} and p_{out} of the target port p of q (Fig. 4). The input/output ports of the cross-organizational workflow net are those of the constituent modules. Figure 1 shows a modular workflow architecture that is made into a cross-organizational workflow net by adding the dashed arcs.

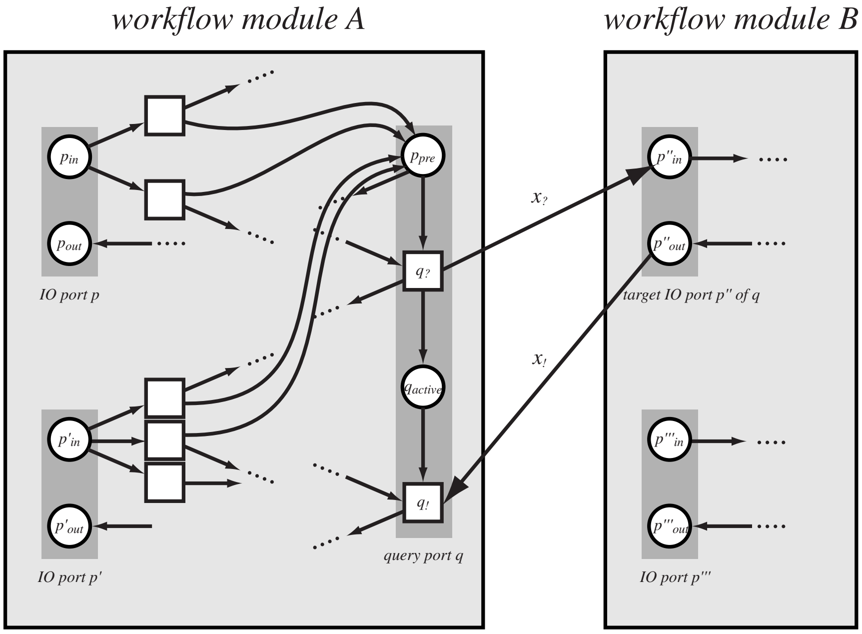

Fig. 4. *A query port in a workflow module and its connection in a cross-organizational workflow net to the target of that port.*

2 Workflow nets for multiple cases

A *case* is a run of a business process invoked by a single input. In a workflow net a case starts when a token is deposited by the environment in the input place of the net, and ends when a token arrives at the output place. In van der Aalst [1, 2], workflows are considered to be *case driven*, meaning that every case can be considered as running in a fresh copy of the workflow net. This approach guarantees that different cases do not influence each other, a property we call *case independence*.

The case driven approach would unduly complicate the constructions of the present paper. We consider different cases to be running in parallel in the same workflow net, which can happen in particular in workflow modules within modular workflow architectures, namely when a module is invoked twice as a subroutine in the course of executing a single case of another module. Nevertheless, case independence can be ensured easily in predicate/transition nets by assigning a unique identifier to every case. In practice, this identifier could consist of the name and address of whomever presented the case to the workflow, and the time of submission. To this end we assume that every token deposited in the input place of a workflow net carries a unique case identifier. The transition guards of each transition will require that that transition fires only when all incoming tokens have the same identifier. Each output token will have that identifier as well. This method precludes any interference between cases.

In this section, we first formally define case independence, and then describe a method to transform any multi-organizational workflow net into a case independent one under the assumption made above. Subsequently we extend this method to workflow modules in such a way that the case-independence of modules is preserved when these modules are embedded in a cross-organizational workflow net.

2.1 Case independence

A multi-organizational workflow net W enjoys *case independence* if any firing sequence $M_0[t_1\rangle M_1[t_2\rangle M_2[t_3\rangle \cdots$ where M_0 is a marking that only puts tokens in input places of W, can be obtained by *interleaving* firing sequences that start with markings that put only one token in an input place. This means that, if the initial marking M_0 has n tokens, each marking M_i can be written as $M_{i1} + \cdots + M_{in}$, such that, for $j = 1, ..., n$, M_{0j} puts only a single token in an input place, and for each transition t_i in the sequence, there is a $j \in \{1, ..., n\}$ such that $M_{(i-1)j}[t_j\rangle M_{ij}$ and $M_{(i-1)k} = M_{ik}$ for $k \neq j$. In words, the given firing sequence can be decomposed into n firing sequences starting with a 1-token marking, and every transition t_i belongs to one of these n firing sequences.

We do not aim for general case independence, but merely for case independence under the assumption that the environment supplies tokens that are tagged with distinct identifiers from a set ID. We call this *ID-case independence*. It is formally defined just as case independence above, but only for initial markings M_0 that supply tokens of the form (id, d) with $id \in ID$ such that all supplied tokens have distinct identifiers.

2.2 Ensuring case independence

Let $W = (S, T, F, \lambda, IO)$ be a multi-organizational workflow net designed for the case driven approach, in a framework with $\mathcal{D}$ the set of possible tokens. Let *ID* be a set of possible identifiers, and take $\mathcal{D}' = \boldsymbol{ID} \times \mathcal{D} = \{(id, d) \mid id \in \boldsymbol{ID} \wedge d \in \mathcal{D}\}$, the set of possible tokens with attached identifiers. Suppose $\{x_1, \ldots, x_n\}$ is the set of variables labeling the arcs entering and leaving a transition $t \in T$, then we define $\lambda'(t)$ to be

$$\exists\, id \in \boldsymbol{ID}\colon x_1 = (id, x'_1) \wedge \cdots \wedge x_n = (id, x'_n) \wedge \lambda(t)[x'_i/x_i\ (i = 1, ..., n)]$$

where $\lambda(t)[x'_i/x_i\ (i = 1, ..., n)]$ denotes the result of substituting x'_i for x_i in $\lambda(t)$ for $i = 1, ..., n$. Now $W' = (S, T, F, \lambda', IO)$ is a multi-organizational workflow net that behaves just like W when presented with a single input token, but in which ID-case independence is guaranteed. The framework in which W' is specified uses the domain $\mathcal{D}'$ of possible tokens and a suitably extended collection $\mathbb{F}'$ of formulas.

2.3 Ensuring case independence in modular workflow architectures

We consider a modular workflow architecture to be ID-case independent if its individual modules are case independent under the assumption that tokens deposited by the environment of the architecture in input places of the modules of the architecture have distinct identifiers, chosen from the set ID. For our main results below, ID-case independence of modular workflow architectures is essential. ID-case independence of modular workflow architectures can be achieved by the method described in Sect. 2.2, provided that all tokens deposited in input places of the modules of the architecture have distinct identifiers. Such tokens may be provided by the environment or by query transitions. Therefore we have to require that query transitions never emit tokens with identical identifiers to input ports of any given module. This requirement will be fulfilled when query ports augment the identifier of a case they are dealing with with their own identity. To this end, for a collection of identifiers ID supplied by the environment, and a collection of $\mathbf{Q}$ of potential names of query ports appearing in a modular workflow architectures, let $ID_{\mathbf{Q}}$ be the collection of identifiers $id * q_1 * q_2 * \cdots * q_k$ with $k \in \mathbb{N}$, $id \in ID$ and $q_j \in \mathbf{Q}$ for $j = 1, ..., k$. Such identifiers consist of a global case identifier id from ID, attached to a token by the environment, together with a sequence of names of query ports that the token passed through in succession. For a query port $q \in \mathbf{Q}$ the transition guards $\lambda'(q_?)$ and $\lambda'(q_!)$ now are $\exists\, id \in ID_Q : x_1 = (id, x'_1) \wedge \cdots \wedge x_n = (id, x'_n) \wedge x_? = (id{*}q, x'_?) \wedge \lambda(q_?)[x'_i/x_i \ (i{=}1, ..., n, ?)]$ and $\exists\, id \in ID_Q : x_1 = (id, x'_1) \wedge \cdots \wedge x_n = (id, x'_n) \wedge x_! = (id{*}q, x'_!) \wedge \lambda(q_!)[x'_i/x_i \ (i = 1, \ldots, n, !)]$. It could be that in a single case a single module invokes another module twice by means of different query ports. Using the names of those ports to augment the case identifier results in the invoked module only receiving tokens with distinct identifiers. Here it is crucial that in a single case any particular query port q be used only once. For this reason we introduced the query places q_{pre}.

We call a workflow module *manifestly ID-case independent* if it has been made *ID*-case independent by the method above. If all workflow modules in a modular workflow architecture are manifestly *ID*-case independent, then surely the architecture itself is *ID*-case independent.

In a modular workflow architecture consisting of manifestly ID-case independent workflow modules the type of token identifiers is that of a stack. Tokens deposited by the environment are assumed to have an identifier $id \in ID$. Whenever a token passes though a query transition $q_?$ into another module (containing the target of q) the name q of the corresponding query port is appended to the identifier, or "pushed on the token's stack". Within a module the identifier of tokens is preserved. When a token is retrieved by the query transition $q_!$ from the output place of the target of q, the name q is popped from the token's stack.

3 Proper termination

Typically, van der Aalst [1, 2] requires workflow nets to be *sound*, in the sense that
(1) if a token is put in the input place, a token will eventually appear in the output place,
(2) when a token appears in the output place, no other tokens are left in the net, and
(3) every transition in the net can under some circumstance be fired.

These properties are formalized as follows (where the marking $\{\!\!\{ i \}\!\!\}$ is the multiset that only contains the input place of the workflow net):
(1) Any fair firing sequence starting with $\{\!\!\{ i \}\!\!\}$ contains a marking with a token in o.
(2) Any marking that is reachable from $\{\!\!\{ i \}\!\!\}$ and has a token in o, must equal $\{\!\!\{ o \}\!\!\}$.
(3) For any transition t there is a firing sequence starting with $\{\!\!\{ i \}\!\!\}$ in which t appears.
In van der Aalst [1, 2] workflow nets are required to satisfy
(4*) The input place i does not have incoming arcs and the output place o does not have outgoing arcs. Furthermore, for every place or transition $r \in S \cup T$ there should be a path in the net from i to o via r.
This structural property implies that (4) in the marking $\{\!\!\{ o \}\!\!\}$ no transition is enabled. In this paper, we will drop the structural requirement (4^*) but retain its consequence (4), which will be treated as an additional soundness requirement.

Soundness property (1) says that the workflow net is guaranteed to provide an output token, and property (2) adds that when the environment retrieves this token from the output place, no tokens remain in the net. In combination with the assumed case independence of workflow nets, properties (1) and (2) imply that for each token put in the input place, exactly one token will eventually appear in the output place. It is not hard to find counterexamples showing that this does not follow without case independence.

Soundness property (3) is one of parsimony, and has no bearing on the operational behavior of the workflow net. Any workflow net can be transformed into an operationally equivalent one that satisfies this property, namely by deleting all transitions that can never be fired. We will not be concerned with this soundness property here, and instead use a concept of soundness consisting of properties (1), (2), and (4). This form of soundness is called *proper termination* [4].

The following definition extends this concept to multi-organizational workflow nets for which all tokens are of the form (id, d) with id a case identifier. From here onwards we work in a framework for predicate/transition nets in which all token have this form. Now a marking is a multiset of elements $(s, (id, d))$ with s a place in the net. As the case identifier is supplied by the environment dropping a token in an input place, and that environment is hoping for an output token pertaining to the same case, only tokens of the form (id, d') count as legitimate output when an input (id, d) was supplied. Tokens with any other identifier may pass through output places casually.

Definition 1. *A multi-organizational workflow net is* properly terminating *if*
(1) Any fair firing sequence starting with a marking $\{\!\!\{ (p_{in}, (id, d)) \}\!\!\}$ for some input/ output port $p \in IO$ contains a marking with a token (id, d') in p_{out}.
(2) Any marking that is reachable from a marking $\{\!\!\{ (p_{in}, (id, d)) \}\!\!\}$ and has a token (id, d') in p_{out}, must equal $\{\!\!\{ (p_{out}, (id, d')) \}\!\!\}$.
(4) In a marking $\{\!\!\{ (p_{out}, (id, d)) \}\!\!\}$ no transition is enabled.

Note that because in a workflow module in a modular workflow architecture a query transition $q_!$ lacks its incoming arc $(p_{out}, x_!, q_!)$ where p is the target of q, proper termination of the workflow module can be understood as proper termination of the workflow module enriched with such an arc, under the assumption that a token is supplied over this arc, i.e., under the assumption that the workflow invoked by $q_?$ terminates. This is what we called *local termination* in the introduction. Also note that, in view of the firing rule for predicate/transition nets, proper termination is required for *any* token supplied over this arc, i.e., for any output returned by the invoked workflow.

3.1 Proper termination of modular workflow architectures

We consider a modular workflow architecture to be properly terminating whenever the cross-organizational workflow net representing the architecture is properly terminating. The main goal of this paper is to formulate conditions on workflow modules in an architecture that guarantee proper termination of the architecture itself.

Lemma 1. *If all workflow modules in a modular workflow architecture satisfy property (4) of Definition 1, then the cross-organizational workflow net W representing the architecture satisfies property (4).*

Proof. Let M be a marking $\{\!\!\{(p_{out}, (id, d))\}\!\!\}$ of W. As in each of the queries q in W the place q_{active} is not marked, none of the query transitions $q_!$ is enabled under M. Any other transition t in W is enabled under M only if in the module containing t, t is enabled under the restriction of M to that module, which is either $\{\!\!\{(p_{out}, (id, d))\}\!\!\}$ or the empty marking. As that module satisfies (4), t is not enabled under $\{\!\!\{(p_{out}, (id, d))\}\!\!\}$, and thus certainly not under the empty marking.

Lemma 2. *If all workflow modules in a manifestly ID-case independent modular workflow architecture satisfy properties (2,4) of Definition 1, then the cross-organizational workflow net W representing the architecture satisfies property (2).*

Proof. Let σ be a firing sequence in W starting with $\{\!\!\{(p^0_{in}, (id_0, d_0))\}\!\!\}$, and let M be a marking in σ with a token (id_0, d'_0) in p^0_{out}. As W is manifestly *ID*-case independent, each token occurring in a marking in σ has a case identifier of the form $id_0{*}q_1{*}q_2{*}\cdots{*}q_k$. Here we also use that in W there are no transitions without incoming arcs, which follows from the assumption that W satisfies (4). Furthermore, each transition in σ has a transition guard of the form $\exists id \in ID_Q\colon \varphi$ and therefore can be annotated with the identifier $id \in ID_Q$ that enabled it. For any $id \in ID_Q$ and any marking M' let $M' \upharpoonright id$ consist of the elements $(s, (id, d))$ of M, and let $\sigma \upharpoonright id$ be obtained from σ by replacing its markings M' by $M' \upharpoonright id$, and by skipping the transitions that are not annotated with id. From the manifest ID-case independence of W it follows that $\sigma \upharpoonright id$ is a firing sequence in one of the modules (that we call W_{id}). It follows immediately from the assumption that W_{id_0} satisfies (2) that $M \upharpoonright id_0 = \{\!\!\{(p^0_{out}, (id_0, d'_0))\}\!\!\}$.

With induction on $k > 0$, we establish that $M \upharpoonright id_0 * q_1 * q_2 * \cdots * q_k$ is empty, which finishes the argument. So assume $M \upharpoonright id$ is either $\{\!\!\{(p^0_{out}, (id_0, d'_0))\}\!\!\}$ or empty. It has to be shown that $M \upharpoonright id * q$ is empty. In case $\sigma \upharpoonright id$ does not contain the transition $q_?$ this is trivial. In case $\sigma \upharpoonright id$ does contain $q_?$, it must also contain $q_!$, as $M \upharpoonright id$ has no tokens in q_{active}. Hence $\sigma \upharpoonright id * q$ starts with a marking $\{\!\!\{(p_{in}, (id * q, d))\}\!\!\}$ and has a marking M' with a token $(id * q, d')$ in p_{out}, where p is the target of q. As W_{id*q} satisfies (2), $M' \upharpoonright id * q = \{\!\!\{(p_{out}, (id, d'))\}\!\!\}$, and $M \upharpoonright id * q$ must be empty.

It is not hard to see that a modular workflow architecture may fail to be properly terminating if it fails to be *ID*-case independent, or if some its workflow modules fail to be properly terminating. However, (manifest) ID-case independence and proper termination of the workflow modules in a modular workflow architecture are not sufficient conditions to guarantee proper termination of a modular workflow architecture. A failure of proper termination of a modular workflow architecture may occur in the case of loops in the connections between its workflow modules. If, for example, module A keeps calling module B and vice versa, as can happen in the car dealership example from the introduction, the resulting architecture has a loop and the associated queries will never be answered.

Below we define a subclass of *acyclic* modular workflow architectures for which the requirements that its modules are manifestly *ID*-case independent and properly terminating are sufficient to ensure proper termination of the architecture. In Sect. 3.3 we will show that the condition of acyclicality can be omitted when equipping workflow modules with a simple business rule that, in essence, prohibits the posing of the same query twice. Workflow modules that are so equipped will be called *query nets*.

3.2 Acyclic architectures

The *connectivity graph* of a modular workflow architecture has as its nodes the workflow modules in the architecture, and a directed edge $A \rightarrow B$ whenever workflow module A has one or more query ports with a target in B. Figure 5 shows the connectivity graph of the modular workflow architecture of Fig. 3. An architecture is called *acyclic* if its connectivity graph has no cycles.

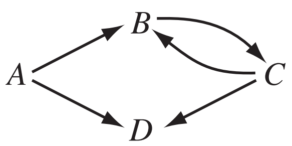

Fig. 5. *Connectivity graph of the modular workflow architecture of Fig. 3. The letters represent workflow modules and the arcs represent queries. This architecture is cyclic because module B contains a query port with a target in module C and vice versa.*

Theorem 1. *If all workflow modules in an acyclic modular workflow architecture are manifestly ID-case independent and properly terminating, then the architecture itself is properly terminating.*

Proof. That the cross-organizational workflow net W representing the architecture satisfies properties (4) and (2) of Definition 1 follows from Lemma's 1 and 2. Now consider a fair firing sequence σ in W starting with a marking $\{\!|(p_{in}, (id, d))|\!\}$. Using the notation and results from the proof of Lemma 2, $\sigma \restriction id$ is a firing sequence of the module W_{id}. In case this firing sequence is fair, as W_{id} satisfies (1), $\sigma \restriction id$, and hence also σ, must contain a marking with a token (id, d') in p_{out}, which had to be established. The only way $\sigma \restriction id$ can fail to be fair in W_{id}, even though σ is fair in W, is when there is a query transition $q_!$ in W_{id} that, from a certain marking in $\sigma \restriction id$ onwards, is continuously enabled but never fires. In σ this query transition cannot be continuously enabled, which is possible only when $\sigma \restriction id*q$ is a firing sequence in W_{id*q} starting with a marking $\{\!|(p^1_{in}, (id{*}q, d_1))|\!\}$ but having no marking with a token $(id{*}q, d'_1)$ in the output place p^1_{out} of the target p^1 of q. (If a such token does arrive in p^1_{out} and $q_!$ never fires, that token is stuck in p^1_{out} by properties (2) and (4) of W_{id*q}, contradicting the fairness of σ.) As W_{id*q} satisfies (1), $\sigma \restriction id{*}q$ cannot be fair, even though σ is. This can only be explained by a query transition $q'_!$ in W_{id*q}, with similar properties as $q_!$ above. Continuing in this vein, we find an infinite sequence of query transitions $q_?$ visited by σ, contradicting the acyclicality of the architecture.

3.3 Query nets

A *query net* is a properly terminating manifestly *ID*-case independent workflow module that never poses the same query twice. The latter can be achieved by a clause in the transition guards of query transitions $q_?$ forbidding the transition to fire when the token identifier contains the name q of that query already. Thus a query net implements a business rule that prevents getting stuck in an infinite loop. The requirement of proper termination moreover implies that the workflow module should embody a backup plan to deal with the situation that the interaction with other workflow modules would have given rise to such a loop. This backup plan may involve a transition that can fire as an alternative to $q_?$ when $q_?$ appears in the identifier of an token. Such an alternative transition must have q_{pre} as one of its input places. An example of this is the transition "admit car cannot be repaired" in Fig. 1. We can now state our main result.

Theorem 2. *If all workflow modules in a modular workflow architecture are query nets, then the architecture is properly terminating.*

Proof. Exactly as for Theorem 1, but this time an infinite sequence of query transitions $q_?$ without matching $q_!$'s cannot be visited, because there are only finitely many queries in the architecture, and no query can occur twice in the sequence.

In fact, Theorems 1 and 2 can be combined and strengthened by merely requiring that all workflow modules in the modular workflow architecture are manifestly *ID*-case independent and properly terminating , and that in any cycle in the connectivity graph of the architecture there is at least one query net.

The dealership's workflow of Fig. 1 is, for adequate choices of the transition guards λ_i, a query net. The manufacturer's workflow on the other hand is not, as its query is not equipped with a backup plan. Nevertheless, the cross-organizational workflow net that combines both modules is, for adequate choices of the λ_i's, properly terminating.

4 Conclusion and comparison with related work

Petri nets have been established as a powerful model for workflow applications. Van der Aalst [1, 2] has proposed *soundness criteria* that guarantee that in a business application modeled by a workflow net every case submitted to the workflow will be completed, and with no references to it remaining in the net. In this paper we examined cross-organizational business applications that are modeled by collections of communicating workflow nets. The amalgamation of all these workflows into a single workflow net may be too large for possibly automated formal analysis. Moreover, individual business partners that operate one of the workflows in the collection may be reluctant to provide the complete specification of their workflow, as result of which nobody can know the complete specification of the amalgamated workflow. For this reason, we explored ways to establish global termination properties for the amalgamated workflow by investigating whether local termination properties hold for the individual workflows in the collection. We proposed local properties that can be checked for individual workflow nets in the collection (by the organizations that operate these individual workflows), without the need for any knowledge of the other workflows in the collection. These local properties guarantee global termination of the amalgamation.

Addressing the same issue, Kindler, Martens and Reisig [6] establish that global termination of the amalgamated workflow is implied by local termination of the component workflows, provided those components are *locally correct with respect to a fairness-closed specification*. In fact, their definition of a *fairness-closed specification* is carefully crafted in such a way that this result holds. In many realistic applications global termination fails even when local termination of the component workflows holds (see Section 1.4 in [6]). It turns out that in such cases there is no fairness-closed specification for which the component workflows are locally correct. Thus these specifications are essential. A problem is that fairness-closed specifications specify the interactions between all workflow modules in the amalgamated workflow. Thus checking correctness of a workflow module with respect to such a specification requires more than local knowledge about that module.

In our approach there is no need for such fairness-closed specifications. Instead, for each of the modules we check locally that a simple business rule is obeyed, that in essence prohibits asking the same question twice. In this way we ensure global termination by checking local properties only.

References

1. WIL M. P. VAN DER AALST (1999): *Interorganizational Workflows: An Approach Based on Message Sequence Charts and Petri Nets.* Systems Analysis—Modelling—Simulation 34(3), pp. 335–367.
2. WIL M. P. VAN DER AALST & KEES M. VAN HEE (2002): *Workflow Management: Models, Methods, and Systems.* MIT Press.
3. HARTMANN J. GENRICH (1987): *Predicate/Transition nets.* In Wilfried Brauer, Wolfgang Reisig & Grzegorz Rozenberg, editors: Petri nets: Central Models and Their Properties, Advances in Petri nets 1986, Part I, LNCS 254, Springer, pp. 207–247.
4. K. GOSTELLOW, V. CERF, G. ESTRIN & S. VOLANSKY (1972): *Proper Termination of Flow-of-control in Programs Involving Concurrent Processes.* ACM Sigplan $7^{(11)}$, pp. 15–27.
5. KURT JENSEN (1994): *An Introduction to the Theoretical Aspects of Coloured Petri Nets.* In Jaco W. de Bakker, Willem-Paul de Roever & Grzegorz Rozenberg, editors: A Decade of Concurrency, LNCS 803, Springer, pp. 230–272.
   Available from `http://www.daimi.au.dk/~kjensen/papers_books/rex.pdf`.
6. EKKART KINDLER, AXEL MARTENS & WOLFGANG REISIG (2000): *Inter-operability of Workflow Applications: Local Criteria for Global Soundness.* In Wil van der Aalst et al., editor: Business Process Management, LNCS 1806, Springer, pp. 235–253.
7. EINAR SMITH (1998): *Principles of High-level Petri Nets.* In Wolfgang Reisig & Grzegorz Rozenberg, editors: Lectures on Petri nets I: Basic models, Advances in Petri nets, LNCS 1491, Springer, pp. 174–210.
8. WORKFLOW MANAGEMENT COALITION (1995): *The Workflow Reference Model.* Available from `http://www.wfmc.org/`.