# Compositional modeling and verification of workflow processes

M. Voorhoeve (email: wsinmarc@win.tue.nl)
Eindhoven University of Technology

**Abstract**

Workflow processes are represented as Petri nets with special entry and exit places and labeled transitions. The transition labels represent actions. We give a semantics for such nets in terms of transition systems. This allows us to describe and verify properties like *soundness*: preservation of the option to terminate successfully. We describe the composition of complex WF nets from simpler ones by means of operators: refinement, sequencing, choice, free merge, iteration, synchronous and asynchronous communication. These operators (except the communication operators) preserve soundness, giving correctness by design. A strategy for verification other properties is described.

## 1 Introduction

Workflow management is an important new development in the computerized support of human work. As such, it is an emerging market with scores of commercially available products, not to mention research prototypes at universities. A workflow management system (WFMS) focuses on *cases* flowing through the organization, while *tasks* are executed for them. We assume the reader to be familiar with these notions; they are explained elsewhere in this book.

In this chapter we limit ourselves to the *process* of a single case. Such a process has a large number of *states* and *actions* move the process from state to state. For this aspect of workflow, Petri net models are often used. Petri nets (c.f. [7]) combine expressive power (to a degree) with a formal semantics that allows analysis. Although it is focused towards actions, there is a clear notion of *state*, which is essential as argued by [1].

A WF net is a Petri net describing the flow of a single case. A source place marks its initial state and a sink place its terminal state. The transitions are labeled; labels correspond to tasks.

In Figure 1, such a net is shown, modeling the process of travel arrangement. A travel request initially enters the process and three parallel activities are started. A budget check is performed and the hotel and travel requirements are studied. If necessary, hotel and travel information is obtained. After obtaining enough information and receiving a budget approval, travel and hotel accommodation is booked, the travel documents and budget approval are assembled and sent to the client.

Models such as the one in Figure 1 can be built intuitively, as illustrated above. However, in order to analyze and discuss these models, we need a formal notion of workflow processes and the way in which they are modeled by nets. Assume that two different WF nets have been presented based on an informal description of one and the same process. One immediately asks whether these WF nets represent the same process, and, if not, in which respect they differ. Such questions need an answer; if not, one cannot be sure whether one's intentions are modeled correctly.

Such a formal semantics of WF nets allows verification. One property that every WF net should satisfy is *soundness*. A WF net is sound iff from every state that can be reached (by executing tasks) from the initial state, it is possible to reach the terminal state. More specific properties can be formulated; e.g. that before termination a certain task $t$ should be executed.

It is a good thing to be able to make models, more or less intuitively, and verifying them after-

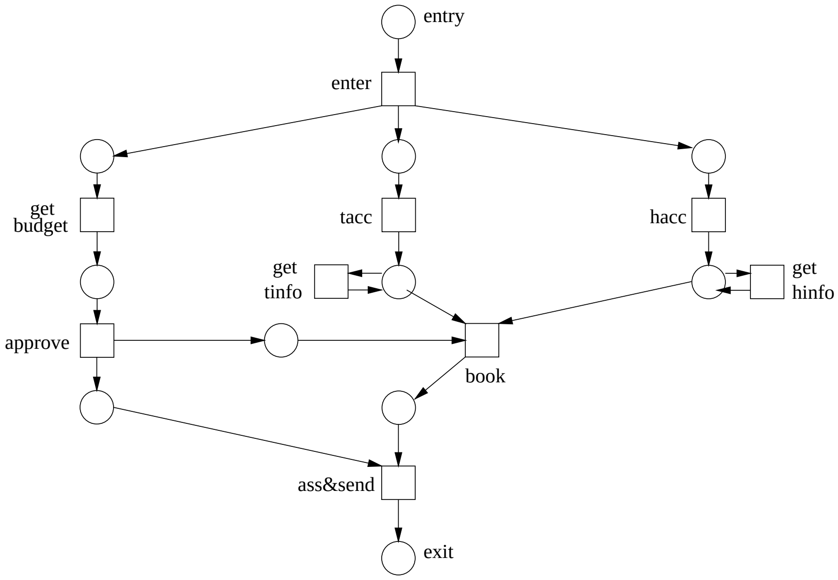


Figure 1: A WF net: travel department

wards. The verification should point out which parts of the model contain erroneous or dangerous constructions and hint at possible improvements. The tool Woflan (WOrkFLow ANalyzer, c.f. [8]) assesses the net's soundness and even supports the modeler by indicating dangerous constructs.

It is even better to create models in an organized way, by combining predefined building blocks in certain prescribed ways. Such models can be verified compositionally, leading to correctness by construction. Often, intuitive and rigorous modeling approaches are combined.

In the sections to come, we introduce transition systems as our basic process model for workflow systems. Then, we define WF nets. We define two ways, called *firing rules* that convert a WF net into a transition system. Next, we introduce operators to compose WF nets, starting with some predefined building blocks. Finally, we treat verification and present some consequences of both our semantics and the compositional approach.

The results presented in this chapter are far from conclusive. Our presentation focuses on the ideas behind them, which we hope the reader will appreciate.

# 2 Definitions

This section starts with introducing *processes* and the bisimilarity equivalence relation. Then we define the class of Petri nets we call WF nets. Finally, we treat two *firing rules* that generate a process from a net.

## 2.1 Processes

A process consists of *states* and *events*. There are two special states: the *initial* and *terminal* state. To each state corresponds a set of events that can *occur* in it. When an event has occurred in some state, another state is reached. The reached state depends on both the old state and the event that has occurred. No events can occur in the terminal state; other states in which no events can occur are called *deadlocks*.

A process (or *transition system*) can be depicted as a *directed graph* with labeled edges. The nodes are states and the edges connect the old state to the new one, where the edge labels denote the events . Two special nodes indicate the initial and terminal states. In Figure 2, such a process

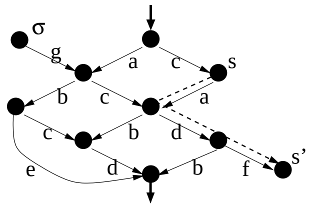


Figure 2: The graph of a process

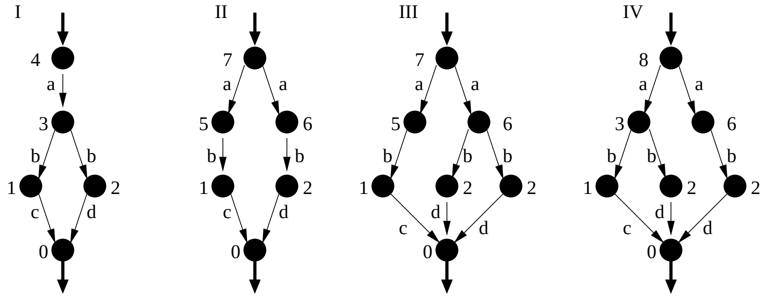


Figure 3: Process colorings

is shown. The thick incoming/outgoing arrow indicates the initial/terminal state, the edges are indicated by solid lines.

We say that a state $s'$ can be *reached* from another state $s$ if there is a sequence of events leading from $s$ to $s'$. In the graph, this corresponds to a directed path as indicated by the dashed line in Figure 2. In a process, we assume that all states are reachable from the initial state, so the state $\sigma$ and edge $g$ in the figure must be removed.

The internal states (neither initial nor terminal) can only be distinguished from one another by the events that they allow and the states that are reached by them. This means that we can divide the states into groups (e.g. by coloring them) of indistinguishable (or *bisimilar*) states. This coloring algorithm starts with three colors (initial, terminal and internal). If a pair of states having the same color can be distinguished (i.e. one state has an $a$-labeled edge leading to a $d$-colored state and the other not), one of the two receives a new color. This is repeated until all states have different colors or states having the same color cannot be distinguished.

In Figure 3, the coloring result is shown for four processes. Processes like $II$ and $III$, that can be colored with exactly the same set of colors are called equivalent or *bisimilar*. Bisimilarity is an equivalence relation. [1] Processes that cannot be colored the same, like $I$ and $IV$ can be distinguished by describing their states; $IV$ has a state in which event $b$ followed by $d$ is possible, but $b$ followed by $c$ is impossible (the state with color 6), but $I$ does not have such a state. Similarly, $II$ and $III$ have a state (with color 5) that $IV$ and $I$ do not possess.

In some cases, we distinguish *visible* and *invisible* events and we want to call processes equivalent if their visible behavior is the same. We represent invisible events by unlabeled edges (often the label $\tau$ is used instead). One cannot tell the difference between a real $a$-labeled edge and a virtual one, i.e. a sequence of zero or more unlabeled edges, an $a$-labeled one and then again unlabeled ones. Neither can one tell the difference between a sequence of unlabeled edges and no edge at all.

Our coloring algorithm then must be adapted: if an $a$-labeled edge leads from one $c$-colored state to a $d$-colored state and no real *or virtual* $a$-labeled edge exists from another $c$-colored state to a $d$-colored state, one of the $c$-colored edges must be recolored. Processes that can be colored with the same colors are called *weakly bisimilar*. The previous notion is called "strong" bisimilarity in

[1] This definition of bisimilarity induces the same equivalence relation as in process algebra literature (see e.g. [6]). The above presentation highlights its essence.

contrast. If all events are visible, the two notions coincide.

Given a process, one can formulate and verify properties it is supposed to have. The logic HML (c.f. [5]) can be used to this end. Bisimilar processes are characterized by the fact that any HML formula that holds for one process also holds for the other and vice versa. If processes are not bisimilar, there is a HML formula that holds for one process but not for the other. Compare the statement we used to distinguish between $I$ and $IV$ in Figure 3.

A property that we want all workflow processes to have is *soundness*. This means that every state of the process (that is reachable from the initial state) can reach the terminal state. A sound process thus cannot reach a deadlock. Soundness can be formulated in HML, so if a process is sound, all (weakly) bisimilar processes are sound too.

### 2.2 WF nets

Simple workflow processes can be modeled directly as transition systems. However, more advanced processes with a lot of parallelism do not allow such an approach, due to an enormous number of states. Petri nets (c.f. [7]) allow a more concise representation.

A net consists of *nodes*, connected by directed edges. Nodes are divided into *places*, depicted as circles and *transitions*, depicted as squares. The transitions are labeled; a transition label corresponds to a *task* in the workflow system. Edges connect places to transitions and vice versa. No edges are allowed between two places or two transitions.

A WF net has an entry place without incoming edges and a exit place without outgoing edges. Each node must lie on at least one directed path from the entry place to the exit place. For technical reasons, we will allow several entry and exit places. In this general case, we require that every node lies on a directed path between some entry and exit place. An example WF net is depicted in Figure 1.

Given a transition $t$, the set of edges leading to $t$ are called *input arcs* of $t$ and the places these arcs lead from are called *input places* of $t$. Similarly, the set of edges leaving $t$ are called *output arcs* of $t$ and the places these arcs lead to are called *output places* of $t$.

A Petri net is a natural and concise way of representing a process. The key idea behind Petri nets is that tasks can occur whenever certain objects (materials, resources or permissions) are present. When a task starts, these objects are consumed. Upon termination, new objects are produced, that may allow new tasks to start.

Each place of a net contains certain objects (called *tokens*). Transitions correspond to certain tasks. The required objects for this tasks reside in its input places, and the result objects in its output places. The execution of an task corresponds to the *firing* of a corresponding transition, consuming tokens for its input places and producing tokens for its output places.

For the sake of simplicity, we assume that only one token per place is consumed or produced when a transition fires. This assumption is not unreasonable for modeling workflow. Without it, the definition of a Petri net and the descriptions below get slightly more complicated. However, this complication is only technical.

### 2.3 Firing rules

The conversion of a Petri net into a process is formalized by so-called *firing rules*. These firing rules thus define the behavior of a net. There do exist algorithms that allow us to decide whether the process of a WF net is sound, and, if so, compute this process. However, a *combinatorial explosion* may occur that makes this decision and computation infeasible. So the firing rules given below are not meant to be a guideline for computing a process but rather serve as a means to derive properties (like soundness) of a WF net's process without having to compute it.

The simplest firing rule is the *interleaving* rule. We assume a net $N$ and derive from it a process $I_N$. A state $s$ of $I_N$ corresponds to a *marking* of $N$, a function that assigns to every place $p$ of $N$ a nonnegative integer $s(p)$. The events of $I_N$ correspond to the transition labels of $N$. We

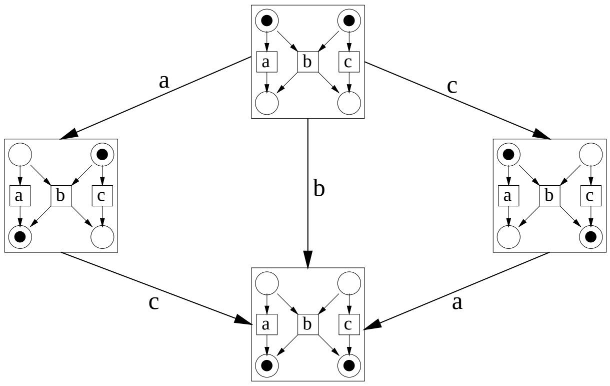


Figure 4: The interleaving process of a net

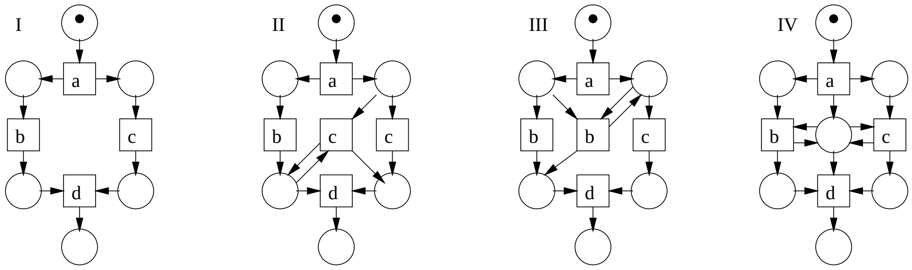


Figure 5: Concurrent versus sequential behavior

denote by $i_t(p)$ the number of input arcs (zero or one) leading from place $p$ to transition $t$ and by $o_t(p)$ the number of output arcs leading from $t$ to $p$. Given a transition $t$ of $N$ with label $a$ and a state $s$ of $I_N$, there is an edge with label $a$ connecting $s$ to a state $s'$ whenever $s(p) > i_t(p)$ and $s'(p) = s(p) - i_t(p) + o_t(p)$ for every place $p$.

The initial state of $I_N$ is the marking where the (each) entry place of $N$ contains one token and the other places none. Similarly, the terminal state of $I_N$ is the marking where the (each) exit place of $N$ contains one token and the other places none.

In Figure 4, the derivation of a process from a Petri net by the interleaving rule is shown. The tokens are indicated by dots; counting the dots in each place yields the state of the net.

The interleaving rule allows us to study most properties of a WF net. In some cases, however, this rule is not completely adequate. Consider the nets in Figure 5 for instance. Each net is mapped to the same process by the interleaving rule, but there is an intuitive difference between e.g. nets I and IV. In net I, tasks $b$ and $c$ can be executed simultaneously (or *concurrently*) after $a$. In net IV, $b$ and $c$ can both be executed after $a$, but if one is busy, the other must wait (this behavior is called *sequential*). As a consequence, the flow modeled by net I is more efficient and will show e.g. lower resource idle times than net IV.

A slightly more subtle difference exists between nets I and III. In net III it is possible to start task $b$ (the transition in the middle) such that task $c$ must wait, although this need not be the case due to the other $b$-labeled transition. This phenomenon cannot happen in I. However, there is no clear difference between nets I and II. If task $b$ has terminated and $c$ not yet started, a second possibility for executing $c$ becomes manifest in net II, but one cannot distinguish between the $c$ transitions firing. [2]

A firing rule that discriminates between concurrent and sequential behavior is the *ST firing rule* (c.f. [4] and [9]). Although the ST rule assumes that tasks are not instantaneous, it is not based

[2] A difference arises when examining the *causal* dependencies between tasks. It seems that we can neglect causality when modeling and verifying workflow processes.

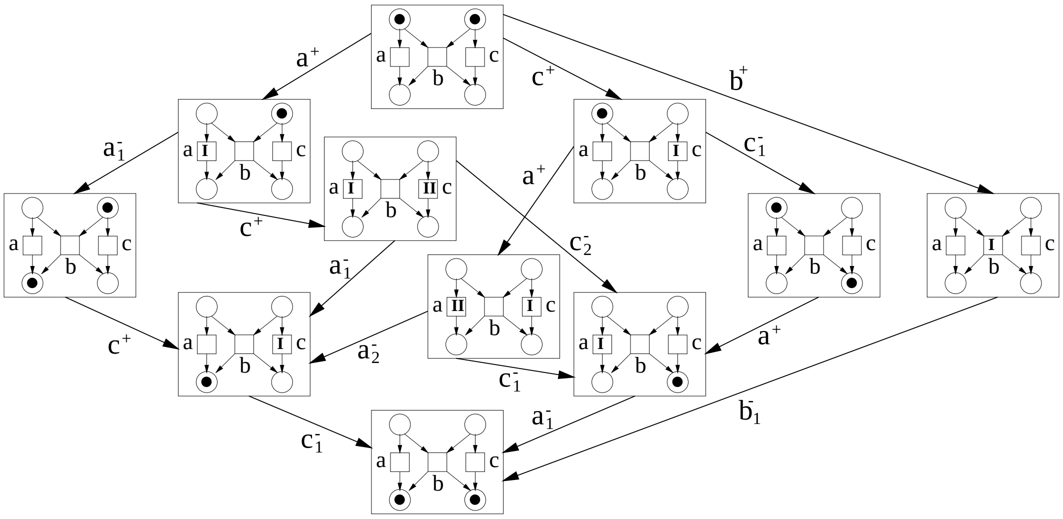


Figure 6: The ST process of a net

on an explicit notion of time, thus allowing to derive properties of WF nets without having to provide e.g. information about the duration of tasks. Most properties of the interleaving process of a net (like soundness) carry over to the its ST process.

We define the ST process $S_N$ of a net $N$ as follows. A state of $S_N$ consists of two parts, a marking (as in $I_N$) and a (possibly empty) list of "busy" transitions. The order of the transitions in this list corresponds to the order in which they started. For net I in Figure 5, $S_I$ contains two distinct states where both transitions are busy: either transition could have started first.

The events of $S_N$ are the starting of a transition and the termination of a busy transition. The labels of these events are $a^+$ for starting an $a$-labeled transition and $a_n^-$ for termination of the $n$-th transition of the busy list, where $a$ is its label.

The initial (terminal) state of $S_N$ is composed of the initial (terminal) marking of $I_N$ and the empty busy list. A state $s$ of $S_N$ with marking $m_s$ and busy list $b_s$ is connected to state $s'$ with marking $m_{s'}$ and busy list $b_{s'}$ by an edge with label $a^+$ if and only if there exists a transition $t$ with label $a$ such that $m_{s'}(p) = m_s(p) - i_t(p)$ for every place $p$ and $b_{s'}$ is $b_s$ with $t$ added to its tail. $s$ is connected to $s'$ by an edge with label $a_n^-$ if and only if $b_s$ contains at its $n$-th position a transition $t$ with label $a$ such that $m_{s'}(p) = m_s(p) + o_t(p)$ for every place $p$.

In Figure 6, the derivation of a process from a Petri net by the ST firing rule is shown. Markings are shown by dots as in Figure 4 and the busy list is depicted by adding numbers to busy transitions.

We conclude this section by pointing out the correspondences between the two processes of a net $N$. The states of $I_N$ correspond to the set $E$ of states of $S_N$ with an empty busy list and an $a$-labeled edge connects two states of $I_N$ if and only if the corresponding states in $S_N$ are connected via $a^+$ and then $a_1^-$. From each state of $S_N$ a state in $E$ can be reached by terminating all busy transitions. States in $E$ can reach one another in $S_N$ if and only if the corresponding states in $I_N$ can reach one another. Hence $S_N$ is sound if and only if $I_N$ is sound.

We call two WF nets $N, M$ *I-equivalent* if $I_N$ and $I(M)$ are bisimilar processes and *ST-equivalent* if $S_N$ and $S(M)$ are bisimilar. It is easy to show that ST-equivalence implies I-equivalence, but not vice versa.

We recommend the use of the ST firing rule and ST-equivalence as equivalence relation between WF nets. The above correspondence indicates that many properties of the ST process can be deduced from the interleaving process. Adopting the ST approach does not rule out the use of interleaving-based analysis tools.

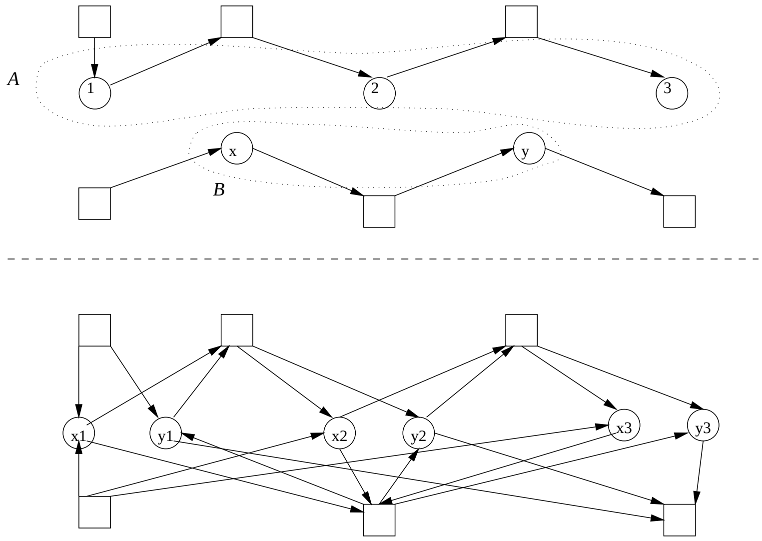


Figure 7: Place fusion

# 3 Construction and verification

Having defined the semantics of WF nets, we investigate the ways in which to construct them. We advocate a compositional approach that allows for both top-down and bottom-up modeling. We then turn to verification, showing how it can profit from the compositional approach. We conclude by presenting the construction of our travel example.

## 3.1 Construction

It is perfectly possible to construct WF nets from intuition, guided by some ad-hoc principles. In fact, the example in Figure 1 has been presented in this vein. However, for large nets more rigor is needed in the net's construction process. We propose here an approach based on building blocks that can be combined in certain prescribed ways into larger nets. The building blocks are themselves WF nets that are well understood and thoroughly verified and validated.

It is of great importance for organizations to create and maintain a set of standard procedures to be aplied in specific situations. Such a "procedure base" is one of the main assets of an organization. The process part of such procedures can be modeled as WF nets and used as building blocks.

We will not further investigate the way to obtain an adequate set of building blocks. Instead, we stress the ways in which WF nets can be combined, i.e. the WF net *operators*. These operators have one or more nets as *operands*; the application of an operator to its operand(s) gives a *result* net. This result then can serve as operand to a new operator. In this way we can build *expressions* that represent nets. This approach is inspired by PBC, the Petri Box Calculus [3].

Most operators are based upon the notion of place fusion. Fusing a set $A$ of places to another set $B$ of places means adding a new place for every pair $(a, b)$ of places from $A \times B$, adding an edge to the place corresponding to $(a, b)$ iff there exists a similar edge to $a$ or $b$ and then removing the places in $A$ and $B$ and edges from or to them. If $A$ and $B$ are singleton sets, this is equivalent to "gluing" the places onto one another. If $A$ or $B$ is empty, it becomes removal of places and edges. In other cases a kind of "weaving" occurs, as illustrated in Figure 7. In the figure, place identifiers are added to illustrate the correspondence between the original and the new places.

We will fuse entry and/or exit places and strive for singleton such places, so fusion of singleton places will occur most often. The complicated weaving of the figure (where both sets have cardinality greater than one) is not likely to occur.

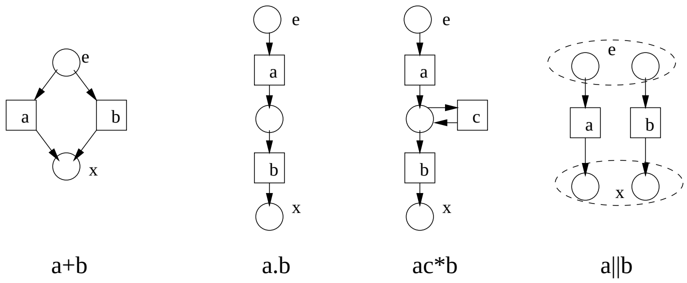


Figure 8: Workflow net operators

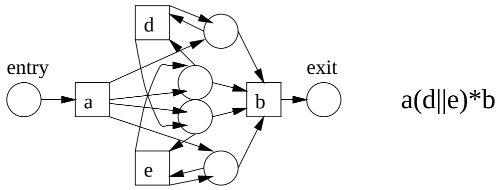


Figure 9: Iterated merge

The operators we propose are sequencing, choice, iteration, free merge, synchronous communication, asynchronous communication and refinement. We describe the intended processes, followed by a net-based explanation. These net-based definitions are similar to [4] and [10].

The sequencing and choice operators have two operands. Putting two processes in sequence means that the first one is executed and upon termination of the first the second one. A choice between two processes means that either process can be executed; the first event determines which one. Graphically, sequencing fuses the exit places of the first operand net to the entry places of the second. Choice fuses the entry and exit places of its operands so that they only share the initial and terminal marking.

Iteration has three operands. The first process is executed; upon termination either the second one or third one can be executed. If the second one is chosen and it terminates, the iterated process has the same state as when the first one terminated, so the choice between the second and third recurs. If the third is chosen and terminates, the iterated process terminates. Graphically, the exit places of the first are fused with the entry and exit places of the second and the entry places of the third, thus creating a loop.

The free merge operator has two operands. Both operand processes can execute independently. Graphically, the two nets are juxtaposed. Since the result of the free merge has more than one entry and exit place, the sequencing operator is often used to add an initial task (the *and-split* of a case into two ot more parallel streams) and a terminal one (the *and-join*).

In Figure 8, the above operators are given when applied to the simplest processes, i.e. isolated tasks ($a, b$ and $c$).

Refinement has one operand and a parameter function $f$ that maps tasks onto processes. In the operand, any transition denoting a task $t$ in the domain of $f$ is replaced by the process $f(t)$. Graphically, another place fusion occurs: the entry places of $f(t)$ are fused with the input places of $t$. Likewise, the output places of the $t$ are fused to the exit places of $f(t)$. A special case is *relabeling*, where $f(t)$ is a single transition.

We can use refinement to formally define the result of the operators in Figure 8 when applied to arbitrary WF nets. For example, the choice $A + B$ between WF nets $A$ and $B$ is obtained by consecutively refining task $a$ and $b$ in the net $a + b$ with $A$ and $B$ respectively. From this observation it is clear that the place fusions from the descriptions above are related. In Figure 9, the net $a(d||e) * b$ is shown that is obtained by refining $c$ with $d||e$ in $ab * c$ from Figure 8. This is a case where weaving cannot be avoided.

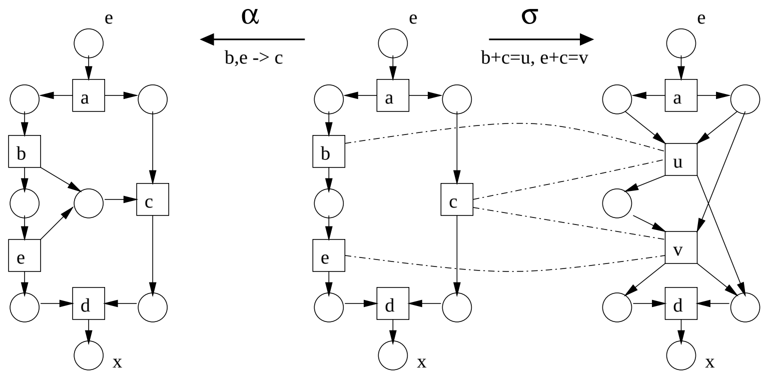


Figure 10: Communication operators

The above operators are the "neat" ones. They are however not sufficient to model all possible processes. Subprocesses that run more or less independently in different parts in an organization (initially modeled by means of the free merge operator) may influence one another at some points: tasks at one side may need information or await decisions from the other side and vice versa. For this phenomenon, "dirty" communication operators are needed. We define synchronous (two-way) and asynchronous (one-way) communication.

Tasks that communicate synchronously must be executed together. This simultaneous execution of these tasks can be seen as a single one. The synchronous communication operator has one operand net and a parameter function $f$ that maps task sets onto tasks. The simultaneous execution of a set $A$ of tasks in the domain of $f$ amounts to executing $f(A)$.

Graphically, synchronous communication amounts to transition fusion and relabeling the transitions thus synchronized. Note that one and the same transition might communicate with several other transitions. If this is the case, another " weaving" (this time with transitions instead of places) will occur.

Asynchronous communication means that a task has to wait until another task has occurred in another stream. There is one operand and a parameter set $R$ that consists of pairs of task sets. If a pair $(A, B)$ is in $R$, then any task from $B$ has to wait until a task from $A$ occurred. Graphically, this amounts to adding a place for every pair $(A, B)$ in $R$, that serves as an output place for the transition(s) in $A$ and as input place for the transition(s) in $B$.

In Figure 10, the communication operators are illustrated. The process $X = a.(b.e||c).d$ is shown in the middle. To its left is $\alpha_R(X)$, where the task $c$ must wait for $b$ or $e$. The set $R = \{(\{b\}, \{c\})\}$ is a singleton pair. To its right is $\sigma_f(X)$, where the task pairs $b, c$ and $e, c$ communicate synchronously and the communication results are $u$ and $v$ respectively. So the domain of $f$ is the set of sets $\{\{b, c\}, \{e, c\}\}$, whereas $f(\{b, c\}) = u$ and $f(\{e, c\}) = v$.

## 3.2 Properties of operators

We like our operators to have the property that combining equivalent operand nets leads to an equivalent result. This means that if nets $N$ and $N'$ are equivalent then replacing $N$ by $N'$ in an expression leads to an equivalent result net. Such a property allows us to prove that two nets are equivalent without having to compute their processes. An operator is called an *ST-congruence* if the above property holds for ST-equivalence and an *I-congruence* if it holds for I-equivalence. Another desirable property is soundness preservation, i.e. that the result is sound if and only if the operands are sound.

The table in Figure 11 shows a list of properties of the operators. Figure 10 shows that the communication operators are not soundness preserving. The original process is clearly sound, whereas the derived processes are not. The right-hand process cannot reach the terminal state (a *deadlock* occurs), whereas the left-hand process can produce a token in the exit place, but

| operator | I-congruence | ST-congruence | soundness preserving |
|---|---|---|---|
| sequencing | yes | yes | yes |
| choice | yes | yes | yes |
| iteration | yes | yes | yes |
| free merge | yes | yes | yes |
| refinement | no | yes | yes |
| sync. comm. | no | yes | no |
| async. comm. | yes | yes | no |

Figure 11: Operator properties

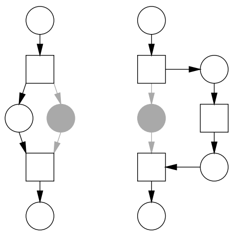

Figure 12: Redundant places

in doing so an extra is token left behind, so it also cannot reach the terminal state. It seems interesting to investigate conditions that are sufficient or necessary for soundness preservation of the communication operators.

The fact that all operators but the synchronizations are soundness preserving can be used when checking the soundness of nets resulting from expressions. The refinement operator is very important, as it allows a hierarchical approach to net construction. The fact that the refinement operator is an ST-congruence and not an I-congruence is one of the main reasons for embracing the ST firing rule.

The construction operators can be incorporated in an editor for workflow processes. This editor could contain a soundness checker that can be invoked for suspect processes that result from the application of dirty operators.

## 3.3 Verification and reduction

The primary reason for modeling workflow is the possibility to control and monitor the work by means of a WFMS. These models will be tested (validated) to create confidence in them. The existence of a formal model also allows for verification. As models become more complex, the need for verification will grow.

There are various properties that need verification. Soundness is a property that all processes must have, but other, more specific properties come to mind, like the need for approval of specific tasks, or protocol conformance. Verification of a large process may become computationally infeasible. The compositional construction of processes allows for compositional verification: defining properties for the operand processes in a certain construct, verifying them and proving that these properties and the given construction yield the desired property of the complete process.

Another property that can be verified is that a certain WF net represents the same process as another given WF net, i.e. their processes are equivalent. As indicated, the congruence properties of the operators allow this to be done compositionally, and even calculationally, using laws from process calculi (or algebra's) like CCS [6] or ACP [2]. Here, we present the *reduction* of a WF net by removing places and/or transitions.

The simplest reduction is achieved by removing *redundant* places. A place $p$ is redundant iff every transition $t$ with $p$ as input place has other input places and whenever these other places contain

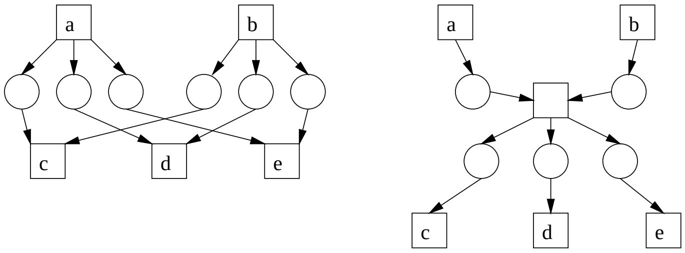


Figure 13: Adding and removing an inert transition

a token, then $p$ contains a token too. Figure 12 shows a few (grey) redundant places. The black net (with the grey arcs and places removed) is the reduced net. The original and reduced nets are ST-equivalent (and thus I-equivalent).

Another reduction consists of removing *inert* transitions. A transition is inert iff it does not represent a real task (i.e. it is unlabeled) and its firing does not remove any options for continuation. This will be the case when the transition does not share any of its its input places with another transition. Removing the inert transition implies fusing its input and output places. Again, the nets are equivalent, this time by weak bisimilarity.

An inert transition can also be *added* in order to simplify complicated place weavings. In Figure 13, this unweaving by adding an inert transition is depicted.

### 3.4 Example construction

As an example, we show the construction of our trip planning example, illustrated in Figure 14. By sequential composition of three actions, the first net is arrived at. Now the middle action `do_work` is refined into the parallel composition (merge) of administration, travel and hotel actions. These actions on their turn are refined into nets involving sequencing and iteration. Asynchronous communication causes the hotel and travel booking to wait for budget approval. The hotel and travel booking is performed synchronously (as they influence one another). By removing a redundant place, we obtain the net in Figure 1.

## 4 Conclusions and further work

This chapter treats the semantics of processes in a workflow context. We combine theoretical notions from Petri nets and process algebra. We hope that we have shown to the non-expert that it is worthwhile to investigate the meaning of the diagrams that represent workflow processes. Backed-up by these investigations, we can construct, discuss, verify and simplify our process models.

It seems worthwhile to further investigate the ST firing rule, since it can be used to assess the net models in terms of efficiency without the need to quantify task durations. The rule also supports refinement, which is important for structured modeling.

Conditions that allow the “dirty” operators to become “cleaner” are an interesting subject. Also, many WFMS systems allow the “rollback” of tasks that are being executed. Incorporating this notion in the firing rules will give yet another process equivalence. The difference between such equivalence notions may be marginal and of theoretical interest only, but it has to be investigated before one can be really sure.

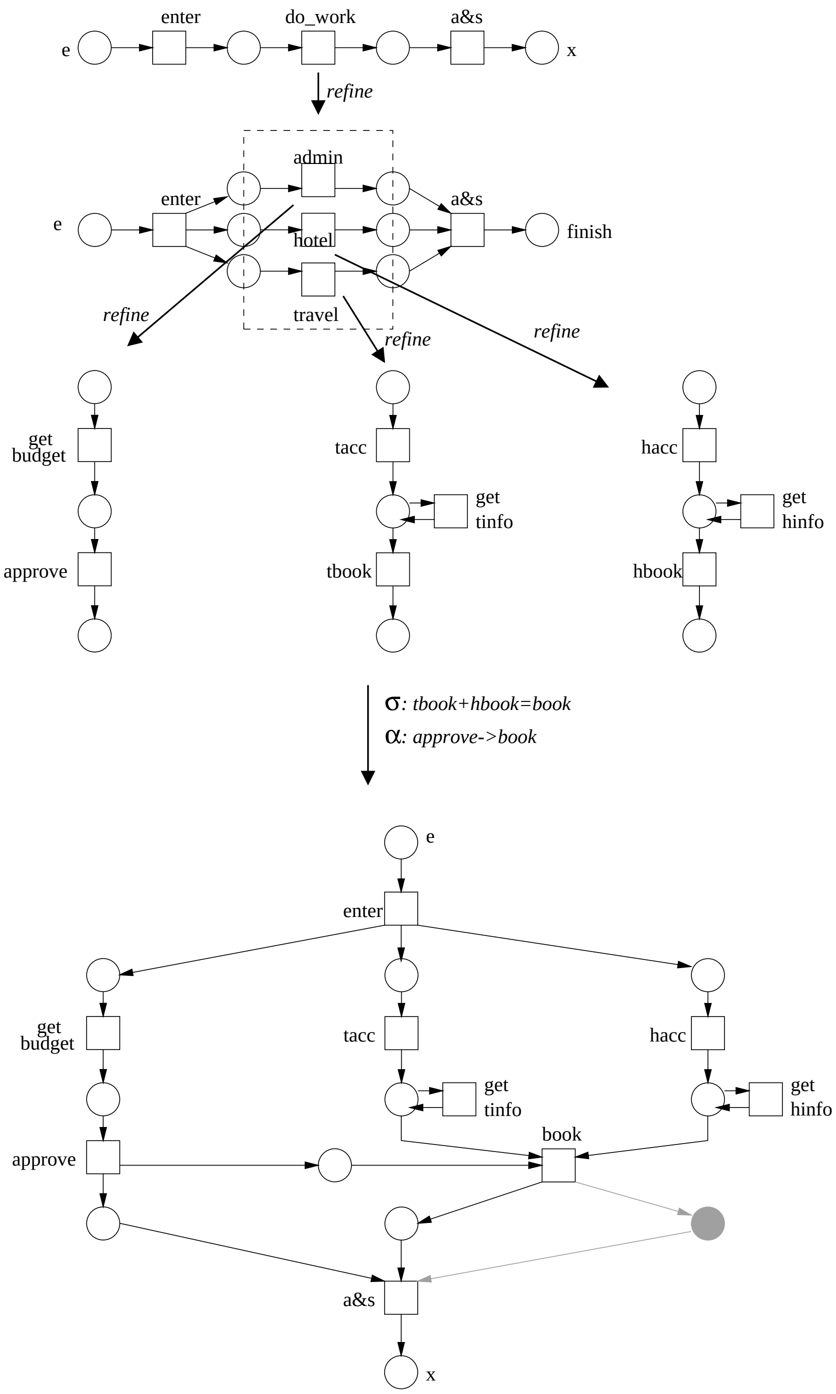


Figure 14: Top-down net composition

# References


[1] W.M.P. van der Aalst. Verification of Workflow Nets. In *Application and Theory of Petri Nets 1997, 18th. International Conference, Proceedings*, volume 1248 of *Lecture Notes in Computer Science*, Toulouse, France, 1997. Springer–Verlag, Berlin, Germany.

[2] J.C.M. Baeten and C. Verhoef. Concrete Process Algebra. In A. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 4, pages 149–268. Oxford University Press, Clarendon, UK, 1995.

[3] E. Best, R. Devillers, and J. Hall. The Petri Box Calculus: a New Causal Algebra with Multilabel Communication. In G. Rozenberg, editor, *Advances in Petri Nets 1992*, volume 609 of *Lecture Notes in Computer Science*, pages 21–69. Springer–Verlag, Berlin, Germany, 1992.

[4] R.J. van Glabbeek and U. Goltz. Equivalence Notions for Concurrent Systems and Refinement of Actions. In A. Kreczmar and G. Mirkowska, editors, *Mathematical Foundations of Computer Science 1989, 14th. International Symposium, Proceedings*, volume 379 of *Lecture Notes in Computer Science*, pages 237–248. Springer–Verlag, Berlin, Germany, 1989.

[5] M. Hennesy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[6] R. Milner. *Communication and Concurrency*. Prentice–Hall, London, UK, 1989.

[7] W. Reisig. *Petri Nets*. Springer–Verlag, Berlin, Germany, 1985.

[8] H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnosing Workflow Processes using Woflan. Computing Science Reports 99/02, Eindhoven University of Technology, 1999.

[9] W. Vogler. Bisimulation and Action Refinement. *Theoretical Computer Science*, 114(1):173–200, 1993.

[10] M. Voorhoeve. State Event Net Equivalence. Computing Science Reports 98/02, Eindhoven University of Technology, 1998. `http://wwwis.win.tue.nl/~wsinmarc/techreports.html`.