Verifying Workflows with Cancellation Regions and OR-joins: An Approach Based on Invariants

H.M.W. Verbeek[1], W.M.P. van der Aalst[1,2], and A.H.M. ter Hofstede[2]

[1] Faculty of Faculty of Technology Management,
Eindhoven University of Technology, The Netherlands,
[2] Faculty of Information Technology,
Queensland University of Technology, Australia

Abstract

The YAWL (Yet Another Workflow Language) workflow language supports the most frequent control-flow patterns found in the current workflow practice. As a result, most workflow languages can be mapped onto YAWL without loss of control-flow details, even languages allowing for advanced constructs such as cancellation regions and OR-joins. At the moment no analysis techniques are available for such languages, because both cancellation regions and OR-joins are "non-local" properties and therefore difficult to verify. Hence, a verification approach for YAWL is desirable, because such an approach could be used for any workflow language that can be mapped onto YAWL. This paper introduces a verification approach for YAWL that abstracts from the actual semantics of the OR-join. This approach is correct (errors reported are really errors), but not necessarily complete (not every error might get reported). This incompleteness is due to the fact that the approach approximates the OR-join semantics. Nevertheless, our approach can be used to successfully detect errors in YAWL models. Moreover, the approach can easily be transferred to other workflow languages allowing for advanced constructs such as cancellations and OR-joins.

1 Introduction

At the moment, dozens of workflow management systems are available on the market, examples are Staffware, COSA, WebSphere Workflow, Visual Workflo, SAP R/3 Workflow, Forté Conductor, Meteor, and Mobile. Unfortunately, these systems all use proprietary languages to specify workflows, each with different constructs, possibilities, and impossibilities.

This papers focuses on the verification of workflows, and in particular, on the control-flow aspect of these workflows. Basically, this control-flow aspect determines which tasks can be executed in which order. Typically, all available workflow management systems support the more basic control-flow patterns, like sequence, choice, and parallel flow. However, more advanced patterns exist [6] that are typically supported by some, but not all, of these systems.

The YAWL (Yet Another Workflow Language) workflow language [5] was originally conceived as a workflow language that would support all-but-one of the 20 most frequently used patterns found in existing workflow languages. As such, YAWL supports the multiple instance patterns, the OR-join pattern, and the cancellation patterns. The only pattern not supported by YAWL is the implicit termination pattern, and the authors of YAWL deliberately chose not to support this pattern.

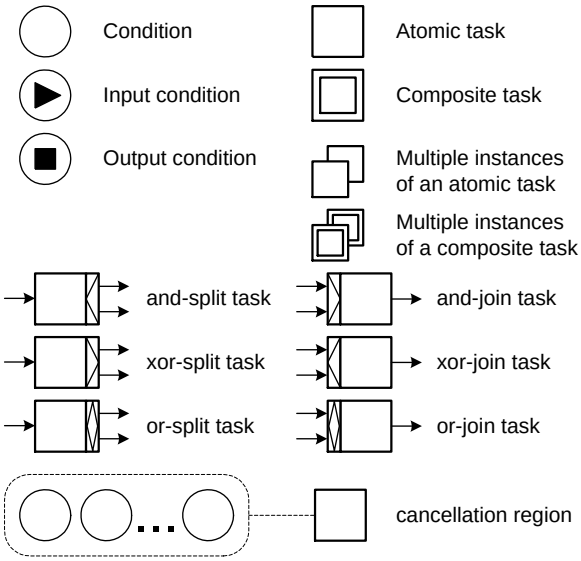


Figure 1: Symbols used in YAWL

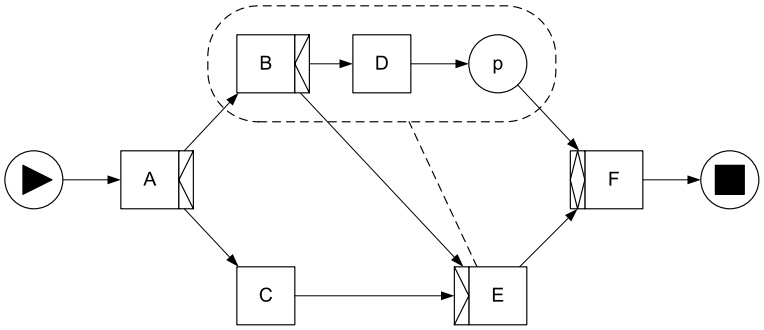


Figure 2: Example YAWL model

Figure 1 shows the symbols used by YAWL, and gives an indication of the patterns supported by YAWL.

Exactly because YAWL supports these most frequent patterns, it is positioned to be a kind of 'lingua franca' for the control-flow aspect of workflow languages. As such, it is a desirable language for verification purposes: If one can verify YAWL models, one can verify the most frequently occurring patterns and, hopefully, most of the existing workflow models in practice. However, exactly because YAWL supports a lot of advanced patterns, verification of YAWL models is not an easy journey. However, precisely due to YAWL's comprehensive support for advanced patterns, the verification of YAWL models is a challenging problem.

Figure 2 shows a YAWL model which will be used as a running example in the remainder of this paper. Execution of the YAWL model starts at the far left, at the input condition. This condition is reached as soon as an instance of the workflow is created. If the input condition has been reached, task A can be started. If task A completes, tasks B and C can be started. Task E can only be started after both task B and task C have been completed. If task E completes, the tasks B and D and the condition p are cancelled (that is, aborted or withdrawn). Task F acts as an OR-join, that is, after it is triggered via one of its input arcs, it waits if additional triggers may arrive. If condition p is reached, then task F is not to be started if task

E can still be completed. If task E has been completed, task F is not to be started as long as condition p may be reached. YAWL uses a kind of backwards reasoning technique to determine whether a task with OR-join behavior such as task F may be started or not [39].

Figures 1 and 2 illustrate the capabilities of YAWL. From a verification point of view concepts such as composite tasks, multiple instances, XOR/AND-joins, and XOR/OR/AND-joins are fairly standard and not complicating matters. The two constructs that are more difficult to tackle are the cancellation region and the OR-join. These are very useful constructs and more and more languages start to support them. Therefore, it is highly relevant to be able to verify YAWL models, that is, the results can be transferred to other contemporary languages ranging from BPML and UML to Staffware and BPEL. The complicating factor of both the cancellation region and the OR-join is that they make the semantics *non-local* as is discussed below.

Cancellation region. In Figure 2 the completion of task E results in the removal of all tokens/activities in the region consisting of B, D, and p. Clearly, the effect is non-local; besides relating inputs to outputs the task influences a region without being able to see the effect of the cancellation. Note that the task initiating the cancellation cannot tell whether something is actually cancelled. This corresponds to the ability of *reset nets* [13, 14, 16]. A reset net is a Petri net with special arcs (reset arcs) to empty a place independent of the number of tokens involved. This seemingly innocent extension of Petri nets has rather dramatic consequences. Simple questions such as reachability become undecidable. This shows that, although cancellation regions form a very useful modeling construct, they complicate matters. Note that several languages offer such a construct, for example, Staffware allows one step to withdraw another step, BPMN offers several ways to model cancellations [38], and BPEL offers constructs such as compensation and fault handlers that use cancellation-like behaviors. Hence, it is important to be able to analyze models with cancellations.

OR-join. Task F in Figure 2 is a so-called OR-join. Once the OR-join is triggered it will wait as long as additional triggers may arrive. This is also referred to as the "bus-driver semantics" [39], that is, the OR-join is like a bus driver that has to make a decision each time a passenger enters the bus. Should the bus start moving or not? The bus-driver semantics assumes that the bus driver has "perfect knowledge", that is, (s)he can see whether there are still potential passengers on their way to the bus. If there are no such passengers, the bus starts to drive, otherwise the bus will continue to wait. Since potential passengers may decide at any time not to take the bus, the bus may start to drive at a moment no new passengers are boarding, that is, only when it becomes clear that no more passengers will actually board the synchronization takes place. The bus-driver semantics is very appealing for people making workflow designs. Instead of using an explicit AND-join in case of parallel routing and an explicit XOR-join in case of alternative routing or even a small network of AND-joins and XOR-joins to deal with mixtures of parallel and alternative routing, the designer can always use an OR-join and let the system decide whether it needs to synchronize of not. Therefore, many languages support OR-join constructs having the bus-driver semantics, for example, BPMN, BPEL, EPCs, and a variety of workflow systems (Eastman, Domino Workflow, etc.) support some notion of an OR-join. Unfortunately, these languages are vague about the exact semantics or they impose syntactical requirements to make the interpretation easier. For example, in the context of EPCs the OR-join has been debated for several years [8, 21, 24] and it is even possible to create a paradox (the vicious circle [3, 23]). To avoid such problems many systems do not allow cycles in combination with OR-joins, for example, the various implementations of BPEL do not allow links to form a cycle. YAWL is the only system we know that supports

the OR-join without any restrictions. Clearly the OR-join has non-local semantics, the decision to wait or not does not only depend on its direct predecessors but also on parts of the model that may lead to future triggers (that is, "passengers").

This paper presents a verification approach that can deal with cancellation regions and the OR-joins. To make things tangible and to be able to implement and experiment with our approach, we use YAWL as a target language. However, we again would like to emphasize that the results are applicable to a large class of models and systems (as has just been motivated). We will use an approach based on T-invariants [9, 30]. The use of invariants allows us to abstract from the actual semantics of OR-joins. Our approach cannot give a definitive proof that the model at hand is sound: it can only indicate the presence of errors, not the absence. Pivotal to our approach is the concept of "good execution paths", which corresponds to the so-called relaxed soundness property. Basically, a part of a model which is not covered by good execution paths, must contain some kind of error.

The remainder of this paper is organized as follows. Section 2 discusses related work in the area of control-flow verification for workflow models. Section 3 provides the formal concepts we need for our approach, such as WF-nets, relaxed soundness, and T-invariants. Section 4 introduces the mapping from YAWL models onto WF-nets, the subclass of Petri nets on which our approach is based. Section 5 introduces our verification approach and its possibilities. Section 6 introduces the tool *WofYAWL*, which implements our verification approach. Section 7 introduces a case study with our tool, and Section 8 concludes the paper.

2 Related work

The workflow language YAWL has been introduced in [5]. The design of the language is based on the patterns presented in [6]. For detailed information on patterns (including animations and product evaluations), a website is available: `www.workflowpatterns.com`. Documentation on YAWL and the software can be downloaded from `www.yawl-system.com` (YAWL is an open source workflow management system).

From a verification point of view, the cancellation regions and the OR-joins are most challenging. The YAWL OR-join semantics has been discussed extensively in [39]. As far as we know, no publications exist on the verification of the control-flow aspect of YAWL models. In fact, we know of no analysis techniques that aim at workflow languages *supporting both cancellation regions and OR-joins*.

Many authors have been focusing on the verification of workflow models with less expressive power. An overview of verification problems for workflow models is given in [18]. An early example is FlowMake [31, 32], which aims at the verification of the control-flow aspect using graph reduction techniques. Although the authors use a fairly simple language (just XOR/AND-split/join nodes), their approach turned out to be flawed as shown in [4, 25]. Another example is the Woflan tool [35, 37], the workflow verification tool on which the WofYAWL tool (presented in this paper) is built. Woflan focuses on the soundness property for a subclass of Petri nets (WF-nets) [2].

This paper uses the notion of relaxed soundness. This notion was introduced in [11, 10], where it was used as a correctness criterion for EPCs [22]. Like YAWL, EPCs also include OR-joins, which significantly complicates the use of the traditional soundness property as defined in [2]. To fix this, the relaxed soundness property was introduced at the level of EPCs, and mappings were defined from relaxed sound EPCs to sound WF-nets.

Most other papers that deal with the verification of the control-flow aspect of workflow models use model checking techniques [20, 33, 26, 7, 17, 27]. These

techniques all require the construction of the state space, and typically deal with different verification questions than those addressed by this paper. For us, a combination of our tools with model checking techniques would be ideal: First we check with our tools whether a process model adheres to some minimal requirements that any process model should adhere to, second we check additional properties using model checking. Note that some of the model checking techniques [7, 17] are not limited to the control-flow aspect, but can also deal with the data aspect as well. However, the main difficulty of incorporating data is the requirement to truly model applications and humans. This is often not feasible and therefore analysis needs to abstract from data.

This paper heavily uses the fruits of more than 40 years of Petri net research. See [12, 30] for pointers. Particularly relevant is the work on invariants [9], Petri nets with inhibitors arcs, and reset nets [13, 14, 16].

3 Preliminaries

This section introduces the formal Petri-net and YAWL related definitions used in this paper. First of all, we introduce WF-nets [1], a subclass of Petri nets which we use to capture the essential part of the process. Next, we introduce the well-known concepts of soundness [1] and relaxed soundness [10] on WF-nets. Both these concepts are used to verify processes. A process is called sound if it can always complete properly no matter what, and it is called relaxed sound if all parts of the process can be involved in proper completion. However, both these concepts rely on the ability to generate the entire state space of the process. If this state space is too large to be generated within reasonable time, soundness and relaxed soundness might remain inconclusive. For this reason, we also introduce a new approach based on the well-known T-invariants. As we will show later on, this approach comes very close to relaxed soundness, but it does not rely on the construction of the state space. As indicated in Section 1, we focus on YAWL because of its expressiveness. Unlike existing approaches we allow for cancellation regions and the OR-joins. Instead of considering YAWL in detail, we introduce EWF-nets, which capture the essential behavior of YAWL processes. We will motivate why it is possible to abstract from the other parts of YAWL not contained in EWF-nets at the end of this section.

3.1 WF-nets

Basically, a WF-net is a Petri net which has one source place, usually denoted i, and one sink place, o, such that all nodes are covered by the directed paths from i to o. To be able to handle YAWL's cancellation regions, we include inhibitor arcs to our definition of nets. An inhibitor arc specifies that a transition is only enabled if a given place is empty.

Definition 1 *net*
A (Petri) net N is a tuple (P, T, F_i, F_o, I), where:

- *P is a set of places,*
- *T is a set of transitions such that $P \cap T = \emptyset$,*
- *$F_i \in T \rightarrow \mathbb{P}(P)$ maps every transition onto a set of input places,*
- *$F_o \in T \rightarrow \mathbb{P}(P)$ maps every transition onto a set of output places, and*
- *$I \in T \rightarrow \mathbb{P}(P)$ maps every transition onto a set of inhibitor places.*

Usually, $F_i(t)$ is denoted $\bullet t$, and $F_o(t)$ is denoted $t\bullet$. In a similar way, we denote $I(t)$ as $\circ t$. Furthermore, we extend these notations to places: $\bullet p = \{t \in T | p \in t\bullet\}$, $p\bullet = \{t \in T | p \in \bullet t\}$, and $p\circ = \{t \in T | p \in \circ t\}$.

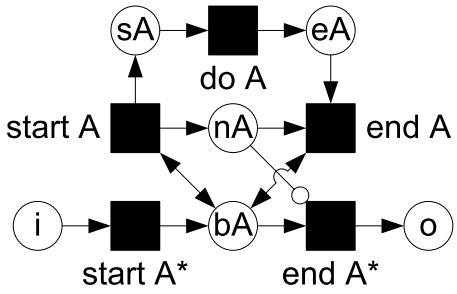


Figure 3: An example net with an inhibitor arc

Figure 3 shows an example of a Petri net with an inhibitor arc. As usual, transitions are visualized using rectangles and places are visualized using circles. There is one source place i ($\bullet$i $= \emptyset$), one sink place o (o$\bullet = \emptyset$), and four more places. There are five transitions. There is one inhibitor arc connecting place nA and transition end A*.

The state of a Petri net, also called marking, corresponds to a multiset of places, that is, $M \in P \rightarrow \mathbb{N}$. For any $p \in P$, $M(p)$ is the number of tokens residing in place p. We will use $[p]$ to denote the marking with just a token in p. A transition $t \in T$ is enabled in state M if and only if for all $p \in \bullet t$: $M(p) > 0$, and for all $p \in \circ t$: $M(p) = 0$. An enabled transition t can fire by removing tokens from the input places and producing tokens for the output places, that is, in the resulting marking $M'(p) = M(p) + 1$ if $p \in t \bullet \setminus \bullet t$, $M'(p) = M(p) - 1$ if $p \in \bullet t \setminus t\bullet$, and $M'(p) = M(p)$ in all other cases.

Consider the net shown in Figure 3. Assume that initially there is a token in place i, that is, the initial state is [i]. In this state start A* can fire. This will result in state [bA]. As long as there is a token in bA, transition start A can fire. If startA fires in [bA], the resulting state is [bA, sA, nA]. Transition startA can fire repeatedly, that is, states of the form [bA, sA^k, nA^k] for $k \in \mathbb{N}$ are reachable. As a result, doA can also fire repeatedly, resulting in states of the form [bA, sA^m, eA^n, nA^k] for $k, m, n \in \mathbb{N}$ and $k = m + n$. Transition endA can fire once for every firing of both start A and do A. Transition end A* can only fire if place bA contains a token and place nA is empty (note the inhibitor arc), that is, the top part of the net can be activated multiple times while the lower part can only complete if the top part is "finished". Note that behavior of the net shown in Figure 3 cannot be modelled using classical Petri nets (that is, a Petri net without inhibitor arcs).

In the remainder of this paper, the concept of a path is used regularly. To avoid confusion, we mention that I is ignored for paths, that is, only F_i and F_o are taken into account for paths. Thus, if $n_0 n_1 \ldots n_N$ is a path, then $n_{x+1} \in F_i(n_x) \cup F_o(n_x)$, for all $0 \leq x < N$.

Figure 3 is a so-called WorkFlow-net (WF-net) having a source place i, a sink place o, and all other nodes on a path from i to o.

Definition 2 *WF-net*
A WF-net [1] is a net (P, T, F_i, F_o, I) such that:

One source place *There is exactly one place $i \in P$ such that $\bullet i = \emptyset$.*

One sink place *There is exactly one place $o \in P$ such that $o\bullet = \emptyset$.*

Directed path *Every node $n \in P \cup T$ is on some directed path from i to o.*

The example net shown in Figure 4 is also a WF-net: the topmost place is its only source place, the bottommost place its only sink place, and every node is on some directed path from the topmost place to the bottommost place.

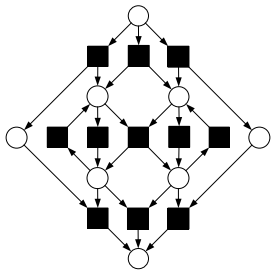


Figure 4: Another WF-net

3.2 Soundness and relaxed soundness

In the context of workflow, place i is the entry point for new cases, while place o is the exit point. Furthermore, ideally, every case that enters the WF-net (by adding a token to place i) should exit it exactly once (by removing a token from place o) while leaving no references to that case behind in the WF-net (no tokens should be left behind). Furthermore, every part of the process should be viable, that is, every transition in the corresponding WF-net should be executable. Together, these requirements constitute the soundness property [1].

Definition 3 *Soundness*
Let net $N = (P, T, F_i, F_o, I)$ be a WF-net with source place i and sink place o. Furthermore, let $[p]$ denote the state with exactly one token in place p (and no tokens in all other places). Net N is said to be sound [1] iff*:*

- *From every state reachable from $[i]$, the state $[o]$ is reachable (completion is always possible).*
- *If in some state s reachable from $[i]$ the place o is marked, then $s = [o]$ (completion is always proper).*
- *No transition is dead.*

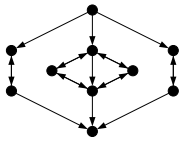


Figure 5: The state space of the example net

Figure 5 shows the state space of the example WF-net shown in Figure 4. The topmost state corresponds to the state $[i]$, whereas the bottommost state corresponds to the state $[o]$. From this state space, we can conclude that the example WF-net is sound: (1) from every state reachable from $[i]$, there exists a path to

$[o]$, (2) $[o]$ is the only reachable state marking place o, and (3) all transitions are present in Figure 5.[1]

Some verification techniques require the addition of an extra transition $*$ such that $\bullet * = \{o\}$ and $*\bullet = \{i\}$ to a WF-net N. We use $*N$ to denote this *short-circuited* WF-net. Figure 6 shows the short-circuited example net. Note a short-circuited net is not a WF-net. The short-circuited WF-net can be used to express soundness in terms of well-known Petri-net properties: A WF-net is sound if and only if its short-circuited net is live and bounded [1]. Recall that liveness and boundedness are two well-known properties supported by a variety of analysis tools and techniques [12, 28, 30].

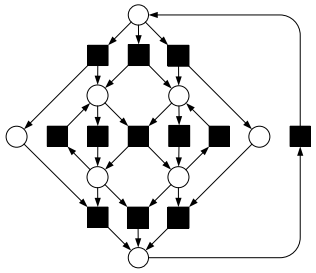


Figure 6: The short-circuited example net

In some circumstances, the soundness property is too restrictive. Usually, a designer of a process knows that certain situations will not occur. As a result, certain execution paths in the corresponding WF-net should be considered impossible. Thus, certain reachable states should be considered unreachable. Note that in the verification process we are often forced to abstract from data, applications, and human behavior. Note that it is typically impossible to model the behavior of humans and applications. However, by abstracting from these aspects typically more execution paths become possible in the model. In her thesis [10], Juliane Dehnert introduced the notion of relaxed soundness to cope with this phenomenon. A WF-net is called relaxed sound if every transition can contribute to proper completion.

Definition 4 *Relaxed soundness*
Let net $N = (P, T, F_i, F_o, I)$ be a WF-net with source place i and sink place o. A transition $t \in T$ is said to be relaxed sound [10] iff *there exists an execution sequence $\sigma = t_1 t_2 \ldots t_n$ such that:*

- *transition t is included, that is, $t = t_i$ for some $1 \leq i \leq n$, and*
- *the net effect of σ is moving the token from place i to place o.*

Net N is said to be relaxed sound iff all transitions $t \in T$ are relaxed sound.

As mentioned before, every case that enters a WF-net should exit it exactly once while leaving no references to that case behind in the WF-net (no tokens should be left behind). Thus, the ultimate goal of a WF-net is to move from place i to place o. The notion of relaxed soundness brings this goal down to the level of transitions: every transition occurs in at least one firing sequence moving a token from place i to place o. A transition that cannot aid in moving a token from place i to place o, cannot help the WF-net in achieving its goal. Hence, such a transition has to be erroneous.

[1]Note that the transition and place labels have been omitted throughout the paper since the mappings are obvious and explicit labels would only distract from the core ideas.

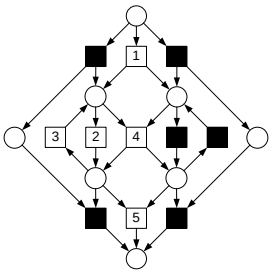


Figure 7: An execution path for the example WF-net

Figure 7 visualizes an execution path in the example net: First transition 1 is executed, then transition 2, and so on. It is straightforward to check that in the example net all transitions are covered by such execution paths.

3.3 T-invariants

An interesting observation[2] now is that an execution path that moves a token from place i to place o corresponds to a cyclic execution path in the short-circuited net: By executing the short-circuiting transition once, the token is back in place i. It is well-known that a cyclic execution path corresponds to a semi-positive transition invariant. A semi-positive transition invariant (or T-invariant for short) is a bag (multi set) of transitions such that the accumulated sets of input places equals the accumulated sets of output places (where accumulation yields bags, not sets). As a result, the net effect of executing every transition from the bag exactly once is zero.

Definition 5 *T-invariant*
Let net $N = (P, T, F_i, F_o, I)$ be a net and let $w \in T \to \mathbb{N}$ be a function assigning a non-negative weight to each of the transitions. Function w is a T-invariant of net N if and only if for all $p \in P : \sum_{t \in {}^\bullet p} w(t) = \sum_{t \in p^\bullet} w(t)$.

By definition, every relaxed sound transition is covered by some path from the initial marking $[i]$ to the final marking $[o]$. As a result, every relaxed sound transition is covered by some T-invariant in the *short-circuited* net. However, this does not work the other way around. T-invariants abstract from the state of the net. Therefore, it might be possible that the bag of transitions covered by some T-invariant cannot be executed (because some tokens are lacking). As a result, there may be a transition that is covered by some T-invariant in the short-circuited net, but that is not covered by any execution path from state $[i]$ to state $[o]$. Figure 8 visualizes a T-invariant for the short-circuited example WF-net which does not correspond to an execution path, where the numbers indicate transition weights and black transitions have weight zero. Note that the execution path would simply block on the transition in the middle.

Thus, if we are unable to generate the state space in reasonable time, then we can use T-invariants as an approximation. Note that for every execution path from state $[i]$ to state $[o]$ the short-circuiting transition only needs to be executed once to obtain a cyclic execution path. Furthermore, note that there may be cyclic execution paths present in the WF-net itself. For these two reasons, we restrict ourselves to T-invariants where the short-circuiting transition has either weight 0 (corresponds to a cycle in the WF-net itself) or 1 (corresponds to an execution path from $[i]$ to $[o]$).

[2]The same observation has also been used in, for example, [34, 36] to reduce computation time for deciding life-cycle inheritance.

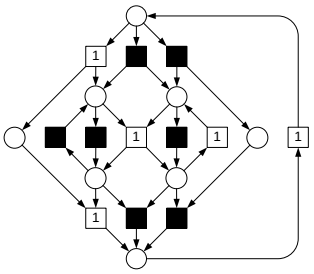


Figure 8: A T-invariant for the short-circuited example WF-net

For constructing a set of minimal (semi-positive) T-invariants, we will use the generic algorithms as described by Colom and Silva [9]. In the worst case, these algorithms are exponential space in the number of transitions, whereas the algorithm to construct a coverability graph is non-primitive recursive space. Thus, constructing a set of T-invariants has a better complexity than constructing a coverability graph. Nevertheless, it might be possible to improve the complexity even further, as we do not need a complete set of minimal T-invariants: We only require a subset of minimal T-invariants that *cover* all transitions that are covered by some minimal T-invariant. Although there is room for improvement, experiments show that our approach using T-invariants already outperforms state-space methods and is able to deal with complex workflows. The computation time is typically reduced from minutes (or even hours) to just a few seconds.

3.4 YAWL

In the introduction, we used figures 1 and 2 to illustrate the capabilities of YAWL. YAWL allows for the hierarchical decomposition of workflow models, that is, using composite tasks it is possible to decompose parts of a model. In Section 3.5 we will explain why we can abstract from this hierarchical decomposition and focus on a single *Extended WorkFlow net* (EWF-net). Figure 2 represents such an EWF-net. The next definition formalizes the notion of an EWF-net.

Definition 6 *EWF-net*
An EWF-net [5] N is a tuple $(C, i, o, T, F, s, j, r, n)$ such that:

- *C is a set of conditions,*
- *$i \in C$ is the input condition,*
- *$o \in C$ is the output condition,*
- *T is a set of tasks,*
- *$F \subseteq ((C \setminus \{o\}) \times T) \cup (T \times (C \setminus \{i\})) \cup (T \times T)$ is the flow relation,*
- *every node in the graph $(C \cup T, F)$ is on a directed path from i to o,*
- *$s \in T \rightarrow \{\wedge, \times, \vee\}$ specifies the split behavior of each task, where $\wedge$ corresponds to an AND-join, $\times$ to an XOR-join, and $\vee$ to an OR-join,*
- *$j \in T \rightarrow \{\wedge, \times, \vee\}$ specifies the split behavior of each task, where $\wedge$ corresponds to an AND-split, $\times$ to an XOR-split, and $\vee$ to an OR-split,*
- *$r \in T \not\rightarrow \mathbb{P}(T \cup C \setminus \{i, o\})$ specifies the additional tokens to be removed by emptying a part of the workflow, and*

- $n \in T \not\rightarrow \mathbb{N} \times \mathbb{N}^{inf} \times \mathbb{N}^{inf} \times \{dynamic, static\}$ *specifies the multiplicity of each task (minimum, maximum, threshold for continuation and dynamic/static creation of instances).*

An EWF-net resembles a WF-net to a large extent: a condition corresponds to a place, a unique input condition and a unique output condition exist, a task correspond to a transition, the flow relation corresponds to the input places and output places, and every node is on some path from the input condition to the output condition. Nevertheless, as the name suggests, EWF-nets contain extensions to WF-nets:

- First of all, conditions are not mandatory in between tasks: tasks can be directly connected to tasks. Basically, an arc from task t to task u (that is, $(t, u) \in F \cap (T \times T)$) is considered to be a placeholder for an *implicit* condition c such that $\bullet c = \{t\}$ and $t\bullet = \{u\}$.

- Second, every task has an associated join behavior, which can be either $\wedge$ (requires all inputs), $\times$ (requires one input), or $\vee$ (requires any non-empty set of inputs). Likewise, every task has an associated split behavior, which can also be either $\wedge$ (produces all outputs), $\times$ (produces one output), or $\vee$ (produces any non-empty set of outputs).

- Third, an EWF-net supports the concept of a cancellation region through function r. If a task $t \in \mathrm{dom}(r)$ is completed, then all nodes in $r(t)$ are cancelled (in Petri net terms: all tokens in the corresponding places would be removed).

- Fourth and last, an EWF-net also supports the concept of multiple task instances through function n. Using this function, it is possible to specify a lower bound and an upper bound for the number of instances created after initiating the task. Furthermore, it is possible to specify a threshold for the number of completed instances. If this threshold is reached, all remaining running instances are terminated and the task completes automatically. Finally, there is a fourth parameter indicating whether the number of instances is fixed after creating the initial instances. The value of the parameter is "static" if after creation no instances can be added and "dynamic" if it is possible to add additional instances while there are still instances being processed.

EWF-nets can be seen as an extension of WF-nets. Therefore, we adopt some of the notations for WF-nets, for example, for $x \in (T \cup C)$: $\bullet x = \{y \mid (y, x) \in F\}$ and $x\bullet = \{y \mid (x, y) \in F\}$.

3.5 Abstractions

A complete YAWL model is a non-empty set of EWF-nets with a special EWF-net N_{top}. Composite tasks are mapped onto EWF-nets such that the set of EWF-nets forms a tree-like structure with N_{top} as root node. Furthermore, a complete YAWL model contains a map. Tasks in the domain of this map are composite tasks which are mapped onto EWF-nets. Throughout this paper we will assume that there are *no name clashes*, for example, names of conditions differ from names of tasks and there is no overlap in names of conditions and tasks originating from different EWF-nets. If there are name clashes, tasks/conditions are simply renamed.

The goal of this paper is a verification approach for a complete YAWL model *based on relaxed soundness and T-invariants*. As such, our approach pivots on the *good execution paths* from the start to the end. As any EWF-net has a well-defined point of entry (its input condition) and a well-defined point of exit (its output

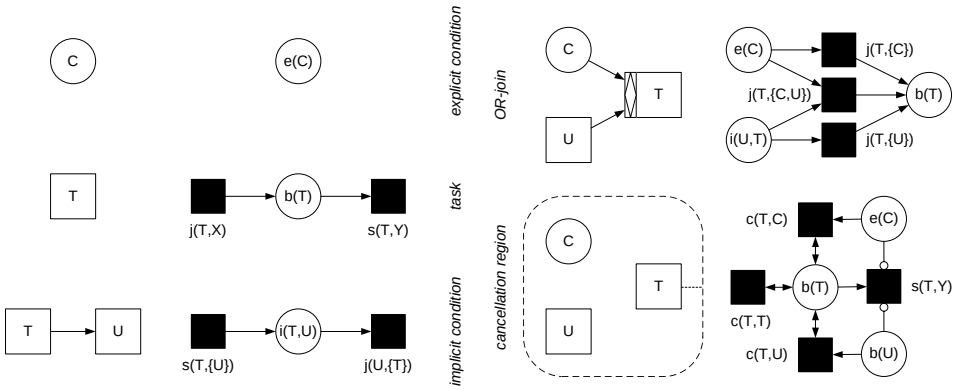


Figure 9: Mapping templates

condition), there is no need to replace a composite task by its underlying EWF-net when verifying the EWF-net that contains that composite task. We can simply verify that underlying EWF-net in isolation. As a result, we can *abstract from hierarchy*.

In a similar way, we can also *abstract from multiple instances* (function n). For the verification, we may assume that the YAWL engine is able to keep the multiple running instances from getting mixed (this is indeed the case). Thus, if we have verified an EWF-net for one instance in isolation, then we may assume that running multiple instances in parallel on the engine will not result in erroneous behavior.

4 Mapping

This section presents the mapping from YAWL models to WF-nets. As we have argued at the end of the previous section, for our approach, it suffices to verify the EWF-nets of the YAWL model in isolation, and we can also abstract from multiple instances (function n of the EWF-net). Furthermore, for our approach, we can also abstract from the actual YAWL semantics of the OR-joins (or-splits): We only want to know whether an OR-join (or-split) with a specific set of inputs (outputs) is viable, that is, whether it is covered by some good execution path.

To keep the join behavior separated from the split behavior, we map a task onto a *busy place*, a number of *join transitions*, and a number of *split transitions*. Conditions get mapped onto places, where explicit conditions are mapped onto *explicit places* and implicit conditions onto *implicit places*. A cancellation region is mapped onto a set of *cancel transitions*, using also inhibitor arcs. Figure 9 visualizes the mapping.

The remainder of this section presents the detailed mapping. For this mapping, assume that we have an EWF-net $(C, i, o, T, F, s, j, r, n)$, and that we want to map it onto a WF-net (P, U, F_i, F_o, I).

4.1 Places

The set of places P contains three types of places: explicit places, implicit places, and busy places. An explicit place $\mathsf{e}(c)$ corresponds to an explicit condition $c \in C$, an implicit condition $\mathsf{i}(t, u)$ corresponds to an implicit condition between tasks $t \in T$ and $u \in T$, and a busy place $\mathsf{b}(t)$ corresponds to a task $t \in T$.

$$\begin{array}{rcl} P & = & \{\mathsf{e}(c)|c \in C\} \\ & \cup & \{\mathsf{i}(t,u)|(t,u) \in F \cap (T \times T)\} \\ & \cup & \{\mathsf{b}(t)|t \in T\} \end{array} \tag{1}$$

4.2 Transitions

The set of transitions U contains three types of transitions: join transitions, split transitions, and cancel transitions. A join transition $\mathsf{j}(t, X)$ corresponds to starting task $t \in T$ given the input set $X \subseteq C \cup T$, a split transition $\mathsf{s}(t, X)$ corresponds to completing task $t \in T$ given the output set $X \subseteq C \cup T$, and a cancel transition $\mathsf{c}(t, x)$ corresponds to canceling a task or an explicit or implicit condition $x \in C \cup T \cup (F \cap (T \times T))$ because task $t \in T$ has completed. Note that the validity of the actual input set depends on the task's join behavior, that is, on $j(t)$, and that the validity of the actual output set depends on the task's split behavior.

$$\mathsf{valid_j}(X,t) = \begin{cases} |X| = |\bullet t| & \text{if } j(t) = \wedge \\ |X| = 1 & \text{if } j(t) = \times \\ |X| > 0 & \text{if } j(t) = \vee \end{cases} \tag{2}$$

$$\mathsf{valid_s}(X,t) = \begin{cases} |X| = |t \bullet| & \text{if } s(t) = \wedge \\ |X| = 1 & \text{if } s(t) = \times \\ |X| > 0 & \text{if } s(t) = \vee \end{cases} \tag{3}$$

Cancel transitions can either cancel a busy place (if the task cancels itself or another task), an explicit place (if the task cancels an explicit condition), or an implicit place (if the task cancels two tasks who have this implicit condition in between). If the task cancels another task, then all tokens from the corresponding busy place need to be removed. However, if a task cancels itself, then all but one token need to be removed as we need the last one to continue. Normally, this would be hard to model in a WF-net, if possible at all. However, because we are only interested in good execution paths (and simply ignore the bad ones), we can model this in a simple and elegant way: Any model that *could* remove all but one token and then continue will do. Figure 9 shows how we can model this: We add a cancel transition to the task, but do not add an inhibitor arc between its busy place and any split transition (as this would effectively block the split transitions).

$$\begin{array}{rcl} U & = & \{\mathsf{j}(t,X)|t \in T \wedge X \subseteq \bullet t \wedge \mathsf{valid_j}(X,t)\} \\ & \cup & \{\mathsf{s}(t,X)|t \in T \wedge X \subseteq t \bullet \wedge \mathsf{valid_s}(X,t)\} \\ & \cup & \{\mathsf{c}(t,x)|t \in \mathsf{dom}(r) \wedge x \in r(t)\} \\ & \cup & \{\mathsf{c}(t,(u,v))|t \in \mathsf{dom}(r) \wedge u,v \in r(t) \wedge (u,v) \in F \cap (T \times T)\} \end{array} \tag{4}$$

4.3 Input places

The set of input places depends on the transition type. Join transitions have only explicit or implicit places as input places, split transitions only busy places, and cancel transitions explicit, implicit, and busy places. A join transition $\mathsf{j}(t, X)$ has an explicit place $\mathsf{e}(c)$ as input *iff* $c \in X \cap C$ and has implicit place $\mathsf{i}(u, v)$ as input *iff* $u \in X \wedge v = t$.

$$\begin{array}{rcl} F_i(\mathsf{j}(t,X)) & = & \{\mathsf{e}(c)|c \in X \cap C\} \\ & \cup & \{\mathsf{i}(u,t)|u \in X \wedge (u,t) \in (F \cap (T \times T))\} \end{array} \tag{5}$$

A split transition $\mathsf{s}(t, X)$ only has busy place $\mathsf{b}(t)$ as input place.

$$F_i(\mathsf{s}(t,X)) = \{\mathsf{b}(t)\} \tag{6}$$

A cancel transition $\mathsf{c}(t,x)$ has an explicit place $\mathsf{e}(c)$ as input place *iff* $x = c$, has implicit place $\mathsf{i}(u,v)$ as input place *iff* $x = (u,v)$, and has busy place $\mathsf{b}(u)$ as input place *iff* $t = u$ (to check whether it may cancel, that is, whether it is active) or $x = u$ (to actually cancel task u). Note that a cancel transition $\mathsf{c}(t,x)$ has place $\mathsf{b}(t)$ as input. Later on, we will see that this transition has this place as output place as well. As a result, the token is only tested, but not removed.

$$F_i(\mathsf{c}(t,x)) = \{\mathsf{b}(t)\} \cup \begin{cases} \{\mathsf{e}(x)\} & \text{if } x \in C \\ \{\mathsf{i}(x)\} & \text{if } x \in T \times T \\ \{\mathsf{b}(x)\} & \text{if } x \in T \end{cases} \tag{7}$$

4.4 Output places

The set of output places also depends on the transition type. Join transitions have only busy places as output places, split transitions only explicit or implicit places, and cancel transitions only busy places. A join transition $\mathsf{j}(t,X)$ has busy place $\mathsf{b}(t)$ as output.

$$F_o(\mathsf{j}(t,X)) \quad = \quad \{\mathsf{b}(t)\} \tag{8}$$

A split transition $\mathsf{s}(t,X)$ has an explicit place $\mathsf{e}(c)$ as output *iff* $c \in X \cap C$ and has implicit place $\mathsf{i}(u,v)$ as output place *iff* $u = t \wedge v \in X$.

$$\begin{array}{rcl} F_o(\mathsf{s}(t,X)) & = & \{\mathsf{e}(c) | c \in X \cap C\} \\ & \cup & \{\mathsf{i}(t,v) | v \in X \wedge (t,v) \in (F \cap (T \times T))\} \end{array} \tag{9}$$

A cancel transition $\mathsf{c}(t,x)$ has busy place $\mathsf{b}(t)$ as output place (as mentioned before, we only want to test this token).

$$F_o(\mathsf{c}(t,X)) \quad = \quad \{\mathsf{b}(t)\} \tag{10}$$

4.5 Inhibitor places

A task may only complete if its cancellation region is empty, that is, if all tokens in the corresponding places (whether they be explicit, implicit, or busy places) have been removed. Thus, the transitions that model the completion of the task, that is, the split transitions, need to be inhibited by all these places. As a result, a split transition $\mathsf{s}(t,X)$ has an explicit place $\mathsf{e}(c)$ as inhibitor place *iff* $c \in r(t)$, has implicit place $\mathsf{i}(u,v)$ as inhibitor place *iff* $u,v \in r(t)$, and has busy place $\mathsf{b}(u)$ as inhibitor place *iff* $u \neq t \wedge u \in r(t)$. As mentioned earlier, a busy place of some task should not inhibit any of the task's split transitions, as this would effectively block the split transitions. Therefore, we require $u \neq t$. Join transitions and cancel transitions have no inhibitor places.

$$\begin{array}{rcl} I(\mathsf{j}(t,X)) & = & \emptyset \\ I(\mathsf{s}(t,X)) & = & \{\mathsf{e}(c) | c \in r(t) \cap C\} \\ & \cup & \{\mathsf{i}(u,v) | u,v \in r(t) \cap T \wedge (u,v) \in F\} \\ & \cup & \{\mathsf{b}(u) | u \neq t \wedge u \in r(t) \cap T\} \\ I(\mathsf{c}(t,x)) & = & \emptyset \end{array} \tag{11}$$

4.6 Example

Figure 10 shows the WF-net that results from applying the mapping to the example EWF-net from Figure 2.

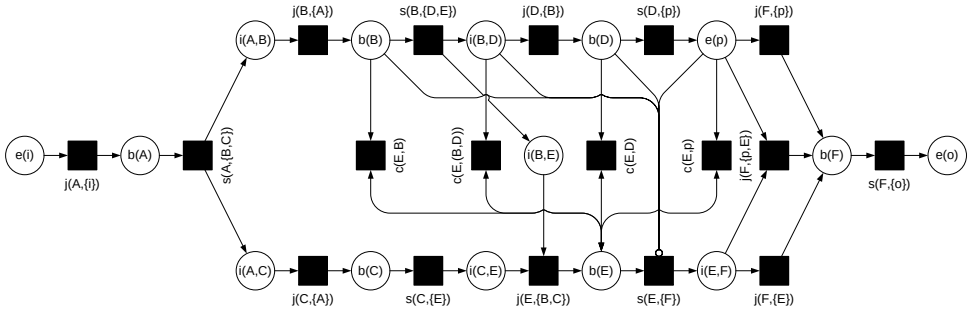


Figure 10: The example EWF-net mapped onto a WF-net

5 Verification

With the mapping in place, we can now turn our attention towards the verification of the YAWL models. As mentioned in Section 1, our goal is not a complete and exhaustive verification of a YAWL model, as such a verification would have to take the complex semantics of the OR-joins into account. Instead, we propose a much simpler form of verification that can simply abstract from this complex semantics.

5.1 Goal

Pivotal to our approach is the concept of good execution paths. A good execution path is a path that, if started from the initial state (the state where the instance has just been created, that is, the state where the input place contains one token), ends in the completed state (the state where the instance has been properly completed, that is, the state where only the output place contains one token). All other paths are considered bad execution paths. Clearly, any task should be viable, that is, should be covered by a good execution path. As a result, at least one of its corresponding join transitions and at least one of its corresponding split transitions should be on some good execution path. Furthermore, it could be the case that no good execution path exists in which a task cancels some node in its cancellation region. Thus, the entire cancellation region of a task should be covered as well by the good execution paths.

Definition 7 *Viability*
Let $N = (C, i, o, T, F, s, j, r, n)$ be an EWF-net, and let (P, U, F_I, F_o, I) be the WF-net EWF-net N is mapped onto. A transition $u \in U$ is called viable iff *it is covered by some good execution path (that is, a firing sequence starting in state $[i]$ and resulting in state $[o]$). The join behavior of a task $t \in T$ is called viable* iff *at least one of its join transitions is viable. Likewise, the split behavior of a task $t \in T$ is called viable* iff *at least one of its split transitions is viable. The cancel behavior of a task $t \in T$ is called viable* iff *all its cancel transitions are viable. A task $t \in T$ is called viable* iff *its join and split behavior are viable.*

5.1.1 Relaxed soundness

The definition of viability on the level of WF-nets corresponds to the definition of relaxed soundness: A transition is viable *iff* it is relaxed sound.

Theorem 1 *Let $N = (P, U, F_i, F_o, I)$ be a WF-net. A transition $t \in U$ is viable* iff *it is relaxed sound.*

Proof By definition, the set of good execution paths coincides with the execution sequences that move the token from the input place to the output place.

As a result, we can use the relaxed soundness property to compute the set of viable transitions. However, the relaxed soundness property requires the entire state space to be computed, and constructing that state space might not be an option. For instance, if the number of reachable states is unbounded, we simply cannot construct the state space. Furthermore, for Petri nets that include inhibitor arcs the reachability problem is known to be undecidable [15]. As a result (we could use a state space to decide reachability), computing the state space might also not be an option if inhibitor arcs are present. For these reasons, we also introduce a structural property that can be used to approximate the set of viable transitions: T-invariants.

5.1.2 T-invariants

By definition, every good execution path removes a token from the input place and adds a token to the output place. As a result, every good execution path corresponds to a T-invariant in the short-circuited net (see also Section 3). However, this does not hold in the other direction: T-invariants might exist that do not correspond to a good execution path. Some of these 'bad' T-invariants can be detected quite easily:

- A T-invariants should have weight 0 or 1 for the short-circuiting transition (we do not want to fire the short-circuiting transition more than once; note that if the weight is 0 then the T-invariant might correspond to an internal cycle).
- A T-invariant that includes a cancel transition for some task should also include a join transition for that task (a task can only cancel other nodes if it has been started).

Nevertheless, 'bad' T-invariants might remain, and the remaining set of T-invariants might still cover some non-viable transitions. As a result, the warnings obtained by our approach might not be complete, but they will be correct.

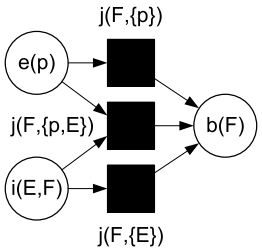
5.2 Viability

Using either the relaxed soundness property or the T-invariant property, we can obtain (an approximation of) the set of viable transitions. However, we still need to map the results back onto the level of the YAWL model.

5.2.1 Input and output nodes

Figure 11 shows how we can map the viability information from the WF-net level back to the EWF-net level, using the join behavior of task F (see also Figure 2 and Figure 10).

- If all corresponding join transitions are viable, then no errors are detected and no warnings are issued.
- If only transition $\mathsf{j}(\mathsf{F}, \{\mathsf{p}, \mathsf{E}\})$ is not viable (that is, only $\mathsf{j}(\mathsf{F}, \{\mathsf{p}\})$ and $\mathsf{j}(\mathsf{F}, \{\mathsf{E}\})$ are viable in Figure 11), then task F might as well have been an XOR-join, and a warning is issued.
- If only transition $\mathsf{j}(\mathsf{F}, \{\mathsf{p}\})$ is viable, then (apparently) task F cannot be executed successfully using the input from task E, and a warning is issued.



Viable	Not viable	Warning
$\mathsf{j}(\mathsf{F},\{\mathsf{p}\})$ $\mathsf{j}(\mathsf{F},\{\mathsf{p},\mathsf{E}\})$ $\mathsf{j}(\mathsf{F},\{\mathsf{E}\})$	none	None.
$\mathsf{j}(\mathsf{F},\{\mathsf{p}\})$ $\mathsf{j}(\mathsf{F},\{\mathsf{p},\mathsf{E}\})$	$\mathsf{j}(\mathsf{F},\{\mathsf{E}\})$	None.
$\mathsf{j}(\mathsf{F},\{\mathsf{p}\})$ $\mathsf{j}(\mathsf{F},\{\mathsf{E}\})$	$\mathsf{j}(\mathsf{F},\{\mathsf{p},\mathsf{E}\})$	Task F could have been an XOR-join.
$\mathsf{j}(\mathsf{F},\{\mathsf{p},\mathsf{E}\})$ $\mathsf{j}(\mathsf{F},\{\mathsf{E}\})$	$\mathsf{j}(\mathsf{F},\{\mathsf{p}\})$	None.
$\mathsf{j}(\mathsf{F},\{\mathsf{p}\})$	$\mathsf{j}(\mathsf{F},\{\mathsf{p},\mathsf{E}\})$ $\mathsf{j}(\mathsf{F},\{\mathsf{E}\})$	Task E is not viable for task F.
$\mathsf{j}(\mathsf{F},\{\mathsf{p},\mathsf{E}\})$	$\mathsf{j}(\mathsf{F},\{\mathsf{p}\})$ $\mathsf{j}(\mathsf{F},\{\mathsf{E}\})$	Task F could have been an AND-join.
$\mathsf{j}(\mathsf{F},\{\mathsf{E}\})$	$\mathsf{j}(\mathsf{F},\{\mathsf{p}\})$ $\mathsf{j}(\mathsf{F},\{\mathsf{p},\mathsf{E}\})$	Condition p is not viable for task F.
none	$\mathsf{j}(\mathsf{F},\{\mathsf{p}\})$ $\mathsf{j}(\mathsf{F},\{\mathsf{p},\mathsf{E}\})$ $\mathsf{j}(\mathsf{F},\{\mathsf{E}\})$	Task F is not viable.

Figure 11: Possibilities for an OR-join

- ...

Note that we have used a binary OR-join to explain our approach, but that other joins are covered as well by our approach. In general, if some input is not covered by any of the viable transitions, then a warning is issued that the uncovered inputs are not viable for this task; if only the transition with all inputs is viable, then a warning that the OR-join could have been an AND-join is issued; if only transitions with only one input are viable, then a warning that the OR-join could have been an XOR-join is issued; and if none of the transitions is viable, then a warning is issued that this task is not viable. Formally, let $t \in T$ be a task, and let its set of join transitions be

$$\{\mathsf{j}(t, X_1), \ldots, \mathsf{j}(t, X_k), \mathsf{j}(t, X_{k+1}), \ldots, \mathsf{j}(t, X_n)\}, \tag{12}$$

such that only the first k join transitions are viable. Then:

- task t is not viable if $k = 0$,
- node n is not viable for task t if $n \in (X_1 \cup \ldots \cup X_n) \setminus (X_1 \cup \ldots \cup X_k)$,
- task t could have been an AND-join if $j(t) = \vee \wedge k = 1 \wedge |X_1| = |\bullet t|$, and
- task t could have been an XOR-join if $j(t) = \vee \wedge \forall_{1 \leq i \leq k} |X_i| = 1$.

Mutatis mutandis, the same holds for output nodes and splits.

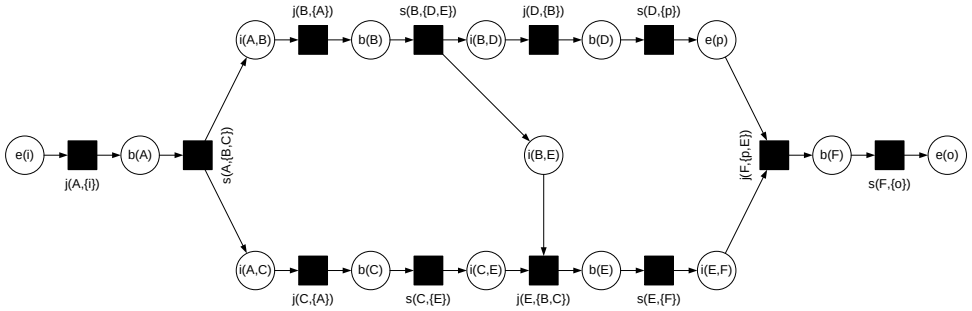


Figure 12: A T-invariant in the example WF-net

5.2.2 Cancel nodes

A cancellation region of task t is viable *iff* all cancel transitions for task t are viable. Only nodes x for which the cancel transitions $\mathsf{c}(t, x)$ is viable can be cancelled successfully by task t. As a result, if a cancel transition $\mathsf{c}(t, x)$ is not viable, then the cancellation of node x by task t is not viable.

5.3 Example

The transitions $\mathsf{j}(\mathsf{F}, \{\mathsf{p}\})$, $\mathsf{j}(\mathsf{F}, \{\mathsf{p}, \mathsf{E}\})$, and $\mathsf{c}(\mathsf{E}, \mathsf{B})$ in Figure 10 are not relaxed sound. As a result, these transitions are not viable, and the following warnings are issued:

- Condition p is not viable for task F.
- Task F could have been an XOR-join.
- The cancellation of task B by task E is not viable.

Figure 12 shows a fragment of the example WF-net that corresponds to a T-invariant. It is trivial to check that this fragment does not correspond to a good execution path (see also Figure 10), because transition $\mathsf{s}(\mathsf{E}, \{\mathsf{F}\})$ can only fire if the places $\mathsf{i}(\mathsf{B}, \mathsf{D})$, $\mathsf{b}(\mathsf{D})$, and $\mathsf{e}(\mathsf{p})$ are empty. Thus, the example EWF-net contains a T-invariant that does not correspond to a good execution sequence, and we might not detect all non-viable transitions. Indeed, we only detect transitions $\mathsf{j}(\mathsf{F}, \{\mathsf{p}\})$ and $\mathsf{c}(\mathsf{E}, \mathsf{B})$ to be not viable. As a result, using T-invariants, only the following warning is issued:

- The cancellation of task B by task E is not viable.

This example illustrates that without computing the state space we can issue useful warnings. However, these warnings are not necessarily complete.

6 Tool

Based on the mapping and the properties as described in the previous two sections, we can now present our tool, called WofYAWL. WofYAWL is a command-line utility that uses the core algorithms of the Woflan workflow verification tool.

6.1 Woflan

Woflan [35, 37] is a workflow verification tool that has been around now for almost ten years. It started as a soundness verification tool that uses the fact that soundness corresponds to the well-known boundedness and liveness properties. During the

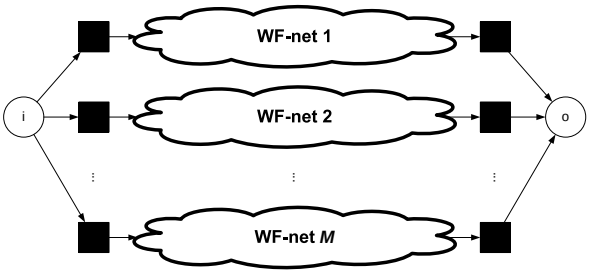


Figure 13: The resulting WF-net

years, several things have been added and/or changed. At the moment, Woflan can determine soundness for WF-nets, can provide diagnostic information if a WF-net is not sound, can check several inheritance relations between two WF-nets, can reduce WF-nets using boundedness and liveness preserving reduction rules [28], and can import for example PNML [19], Staffware, and BPEL files [29].

For the diagnostic information, Woflan uses algorithms for computing minimal sets of semi-positive invariants that are as efficient as possible [9]. For computing a state space, Woflan uses the algorithm to construct a coverability graph in combination with a balanced binary search tree. Unlike the state space, a coverability graph is always finite. Therefore, given sufficient time and space, a coverability graph can always be constructed. From the constructed coverability graph, we can deduce whether the state space is finite. Furthermore, if the state space is finite, then the coverability graph is identical to the state space.

6.2 WofYAWL

The command-line utility WofYAWL imports a YAWL model, maps all embedded EWF-nets to WF-nets, optionally reduces the resulting WF-nets using boundedness and liveness preserving reduction rules, and optionally generates a report using relaxed soundness and/or T-invariants.

6.2.1 Import

First, WofYAWL imports the provided YAWL model. For this import, we use the XML file that the YAWL editor exports for the YAWL engine. At the moment, the latest version of the corresponding XML Schema is version 4, and WofYAWL can import any file that adheres to this schema or previous versions of this schema.

6.2.2 Mapping

Second, WofYAWL maps every embedded EWF-net onto a WF-net, using the mapping as specified in Section 4. The resulting WF-nets are combined into one WF-net together with a new input place, a new output place, an input transition for every WF-net, and an output transition for every WF-net. Figure 13 visualizes this combining of WF-nets into one WF-net, assuming that the YAWL model embeds M EWF-nets. Note that every good execution in one of the 'sub' WF-nets is also a good execution path in the resulting WF-net. Note that for brute-force state-based methods it would not have been a good idea to merge the EWF-nets onto one big WF-net. However, since we use reductions and can always resort to the calculation of invariants, the performance is typically good even after merging the EWF-nets.

As the transitions that are added in this step are of no interest to the user, they will not be added to the report even if they are not viable.

6.2.3 Reduction

Third, WofYAWL optionally reduces the WF-net using boundedness and liveness preserving reduction rules [28]. Typically, a reduced WF-net will result in a smaller state space. Therefore, if WofYAWL has problems constructing the state space, it might be a good idea to have the WF-net reduced before the state space is constructed. Note, however, that the report will be based on a different WF-net, that is, on the reduced WF-net, and that this might complicate the interpretation of the report.

Fourth, WofYAWL optionally creates a report based on relaxed soundness and/or T-invariants.

6.2.4 Relaxed soundness

If we can construct a coverability graph within reasonable time, and if from this coverability graph we learn that the state space is finite, then we propose to use relaxed soundness as it provides a more complete report. If we fail to construct a coverability graph within reasonable time, we propose to construct a coverability graph for the reduced WF-net. If no errors are found for the reduced WF-net, then no errors will be found for the original WF-net. If the state space turns out to be infinite, then we could use the constructed coverability graph as an *approximation* for that infinite state space. However, the results obtained from this approximation might be incorrect. Recall that good execution paths are paths that start in the state with one token in the input place and end in the state with one token in the output place. In the coverability graph, this latter state may be obscured by other states, and it might not even be present at all. As a result, only a subset of the good execution paths might be found, which could result in incorrect results. Therefore we do not propose to use this approximative approach. Instead, we propose to use only the results based on the T-invariants.

6.2.5 T-invariants

If constructing the state space for the reduced WF-net is also a problem, then we propose to use T-invariants. Errors found using T-invariants will correspond to errors found using relaxed soundness, but possibly not all errors will be detected using T-invariants.

6.3 Example

Figure 14 shows a sample report for the example EWF-net (see Figure 2). For this report, no reductions were applied, and both a report based on relaxed soundness (see the `behavior` element in the report) and a report based on T-invariants (the `structure` element) were generated. Note that the names of tasks and conditions have been extended by an underscore and a number (for example, `F_3`, `p_2`). The YAWL editor (Version 1.4) used for the example generates these extensions when exporting to a YAWL engine file. From both reports, we learn that the condition `p` is not viable for task `F`, that task `F` could be and XOR-join instead of an OR-join, and that the cancellation of task `B` by task `E` is not viable.

7 Case study

As a case study we use a YAWL model describing the lifestyle of some famous artist shown in Figure 15. This example is one of the standard examples for the YAWL toolset and can be downloaded from `www.yawl-system.com` and executed using

```
<wofyawl version="0.6" status="released">
    <net file="example.xml">
        <structure>
            <uncovered task="t:example.ywl:example:F_3:join:p_2"/>
            <uncovered task="t:example.ywl:example:E_7:reset:*B_6"/>
            <warning specification="example.ywl"
                     decomposition="example"
                     task="E_7"
                     cancel="B_6"
            />
        </structure>
        <behavior>
            <uncovered task="t:example.ywl:example:F_3:join:p_2"/>
            <uncovered task="t:example.ywl:example:F_3:join:E_7*F_3:p_2"/>
            <uncovered task="t:example.ywl:example:E_7:reset:*B_6"/>
            <warning specification="example.ywl"
                     decomposition="example"
                     task="F_3"
                     input="p_2"
            />
            <warning specification="example.ywl"
                     decomposition="example"
                     task="F_3"
                     OR-join="XOR-join"
            />
            <warning specification="example.ywl"
                     decomposition="example"
                     task="E_7"
                     cancel="B_6"
            />
        </behavior>
    </net>
</wofyawl>
```

Figure 14: A report for the example EWF-net

the YAWL workflow engine. This particular example contains relevant control-flow patterns and is easy to explain, as it doesn't require much domain knowledge. Therefore, it is a nice example to demonstrate and test our verification approach.

Figure 15 shows this model using Version 1.4 of the YAWL Editor, after we have added several errors to it:

- The task "Do everything you are told" now cancels the tasks "Decide to make music", "Do audition", "Learn to play instrument", the conditions "Audition failed?" and "Audition passed", and the (unnamed) condition following the task "Learn to play instrument".
- The join behavior of the task "Choose songs" has been changed from an XOR-join to an AND-join.
- The split behavior of the task "Initial solo performance" has been changed from an XOR-split to an AND-split.

Appendix A shows the resulting YAWL file.

Appendix B shows the initial report. From this report, we learn that the state space could not be generated (the YAWL net is reported to be unbounded). Therefore, we restrict ourselves to the results obtained using the T-invariants:

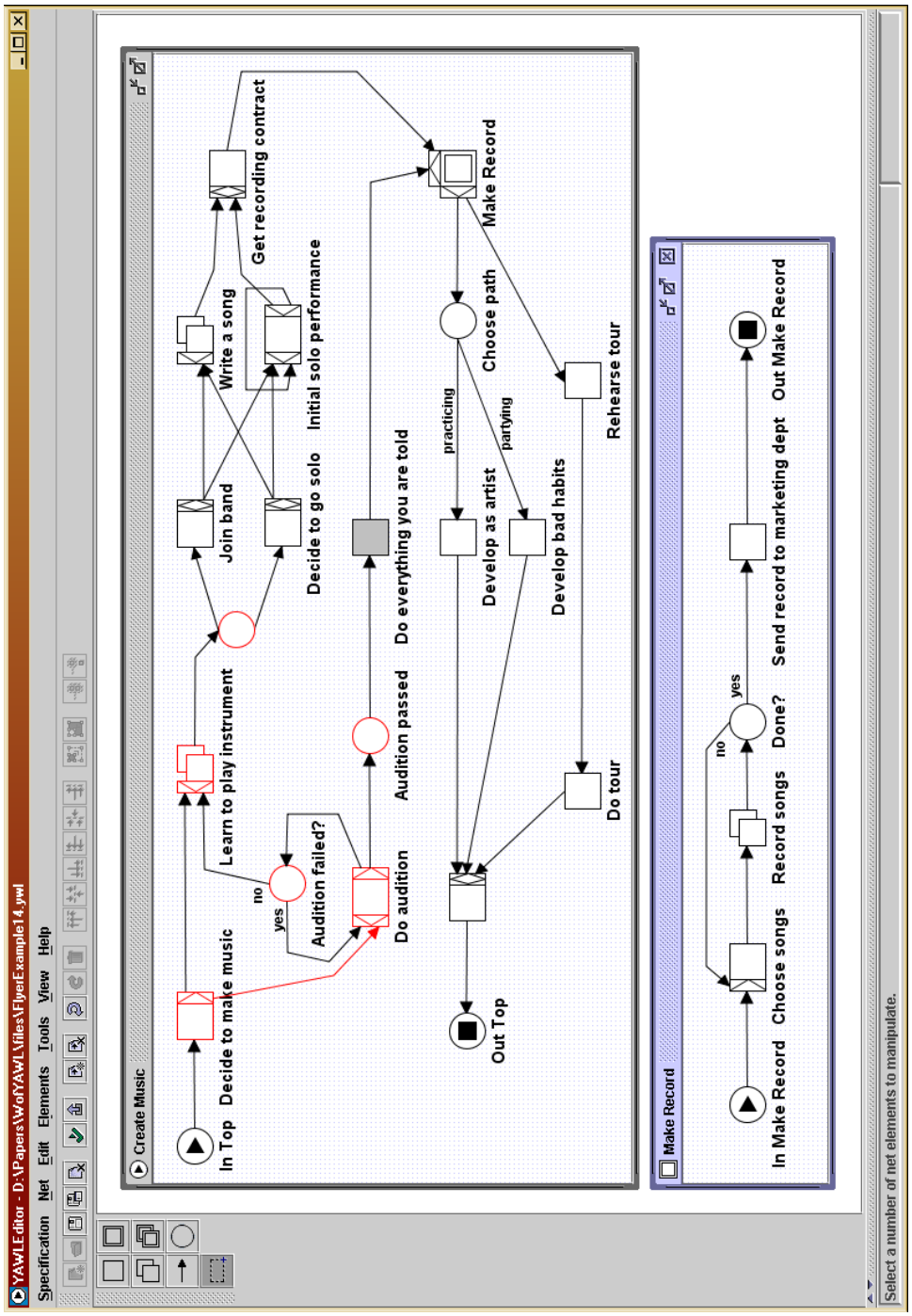


Figure 15: The lifestyle model

1. The task "Decide to go solo" is not viable for the task "initial solo performance" (as any path that goes through task "Decide to go solo" and that starts task "initial solo performance" cannot complete properly).
2. The task "Join band" is not viable for the task "initial solo performance".
3. The condition "Done?" is not viable for the task "Send record to marketing dept".
4. The task "Send record to marketing dept" is not viable.
5. The task "initial solo performance" is not viable for the task "Decide to go solo".
6. The task "Decide to go solo" could be an XOR-split instead of an OR-split.
7. The task "initial solo performance" is not viable for the task "Join band".
8. The task "Join band" could be an XOR-split instead of an OR-split.

The warnings 1, 2, 5, and 7 clearly indicate that something is wrong with the task "initial solo performance". Warnings 1 and 5 state that the arc from task "Decide to go solo" to task "initial solo performance" cannot successfully be taken, warnings 2 and 7 state the same for the arc from task "Join band" to task "initial solo performance". But apparently, nothing is wrong with the alternative task, task "Write a song". These warnings should be sufficient for the designer to have a closer look at the "initial solo performance" task, and to reconsider its split behavior.

The warnings 6 and 8 are a direct result of the previous error. As task "initial solo performance" is not viable, both preceding or-splits should choose to do only the task "Write a song". As a result, both could have been XOR-splits instead of OR-splits.

The warnings 3 and 4 indicate that something is wrong with the entire "Make record" process, as the arc from condition "Done?" to task "Send record to marketing dept" is not viable, which makes the entire process not viable. In such a case, it usually pays off to do a sample execution for this process: The process is not viable, hence no execution can lead to proper completion, thus, every execution should go wrong somewhere. Using a sample execution, a designer should have no problems at all to detect that the task "Choose songs" should be an XOR-join instead of an AND-join.

After having repaired both errors, we generate a new report. From this report, we learn that the state space is finite (and could be constructed within reasonable time). As a result, we use the results obtained using relaxed soundness:

1. Cancellation of task "Decide to make Music" by task "Do everything you are told" is not viable.
2. Cancellation of task "Do audition" by task "Do everything you are told" is not viable.
3. Cancellation of task "Learn to play instrument" by task "Do everything you are told" is not viable.
4. Cancellation of condition "" by task "Do everything you are told" is not viable.
5. Cancellation of condition "Audition failed" by task "Do everything you are told" is not viable.
6. Cancellation of condition "Audition passed" by task "Do everything you are told" is not viable.

7. Cancellation of the implicit condition between task "Decide to make Music" and task "Do audition" by task "Do everything you are told" is not viable.

8. Cancellation of the implicit condition between task "Decide to make Music" and task "Learn to play instrument" by task "Do everything you are told" is not viable.

Clearly, these warnings correspond to the first error we introduced. After having repaired this error as well, we obtain a report containing no warnings and the resulting model is indeed correct.

8 Conclusion

This paper presented a verification approach for the control-flow aspect of YAWL models. This verification approach is based on two properties that are known in the Petri-net literature: relaxed soundness and T-invariants. First, the YAWL model is mapped onto a WF-net, which is a subclass of Petri nets especially tailored towards workflow verification. Second, using the relaxed soundness property and/or the T-invariants property, a report with warnings is generated. If the state space of the WF-net can be constructed within reasonable time, the relaxed soundness property can be used, which yields a more complete report. Otherwise, the T-invariants property can be used, as T-invariants do not require this state space to be constructed. However, using T-invariants we possibly obtain less warnings (that is, a correct but possibly incomplete error report).

Our verification approach is not complete; in the sense that errors may exist that remain undetected by the approach. For a complete approach, we would have to take the complex OR-join semantics into account. Note that our verification approach can abstract from this semantics precisely because we did not require it to be complete. As a result, we believe that, at the moment, our verification is a fair trade off between a complete verification approach and no verification approach at all. In the near future, we hope to be able to also introduce a complete verification approach, which takes the OR-join semantics into account. However, as mentioned, this might be very hard, as this requires a formalism that makes many (but not all) relevant properties undecidable [39]. Note that besides the OR-join, the cancellation region is complicating matters. Using just cancellation region we get the expressive power of reset nets and it is known that reachability is undecidable for reset nets [13, 14, 16].

Being able to verify YAWL models, and given the fact that YAWL models support the most frequently occurring patterns found in existing workflow models today, our verification approach can also be applied to many existing workflow models found today. That is, its application is not limited to YAWL. Our verification approach could immediately be applied to any proprietary workflow language for which a mapping to YAWL exists. For example, our verification approach can be applied directly to other languages, like EPCs and BPEL [29].

Another interesting feature of our approach, that was left unaddressed in this paper, is the possibility to rule out unviable OR-join behavior. Given the tasks that have been executed for some running case, we can determine the set of good execution paths that are still open for this case. If these good execution paths do not contain some join transition that corresponds to an OR-join, than that join transition should not be executed, and that OR-join has to wait for additional tokens. In general, only join transitions that are covered by the good execution paths should be enabled and considered for execution.

Acknowledgments The authors would like to thank the people working on YAWL. Special thanks go to Lachlan Aldred and Lindsay Bradford for developing and editing the lifestyle example.

References

[1] W.M.P. van der Aalst. Verification of workflow nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426, Toulouse, France, June 1997. Springer, Berlin, Germany.

[2] W.M.P. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[3] W.M.P. van der Aalst, J. Desel, and E. Kindler. On the semantics of EPCs: A vicious circle. In M. Nüttgens and F.J. Rump, editors, *Proceedings of the EPK 2002: Business Process Management using EPCs*, pages 71–80. Gesellschaft für Informatik, Bonn, 2002.

[4] W.M.P. van der Aalst, A. Hirnschall, and H.M.W. Verbeek. An alternative way to analyze workflow graphs. In A. Banks-Pidduck, J. Mylopoulos, C.C. Woo, and M.T. Ozsu, editors, *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE'02)*, volume 2348 of *Lecture Notes in Computer Science*, pages 535–552. Springer, Berlin, Germany, 2002.

[5] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[6] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced workflow patterns. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer, Berlin, Germany, 2000.

[7] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, and J.C. van de Pol. mCRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer, Berlin, Germany, 2001.

[8] R. Chen and A.W. Scheer. Modellierung von Processketten mittels Petri-Netz Theorie. Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 107 (in German), University of Saarland, Saarbrücken, 1994.

[9] J.-M. Colom and M. Silva. Convex geometry and semiflows in P/T nets: A comparative study of algorithms for computation of minimal P-semiflows. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 79–112. Springer, Berlin, Germany, 1990.

[10] J. Dehnert. *A Methodology for Workflow Modelling: from Business Process Modelling towards Sound Workflow Specification*. PhD thesis, Technische Universität Berlin, Berlin, Germany, August 2003.

[11] J. Dehnert and W.M.P. van der Aalst. Bridging the gap between business models and workflow specifications. *International Journal of Cooperative Information Systems*, 13(3):289–332, 2004.

[12] J. Desel and J. Esparza. *Free Choice Petri Nets*, volume 40 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1995.

[13] C. Dufourd, A. Finkel, and Ph. Schnoebelen. Reset nets between decidability and undecidability. In K. Larsen, S. Skyum, and G. Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 103–115, Aalborg, Denmark, July 1998. Springer, Berlin, Germany.

[14] C. Dufourd, P. Jančar, and Ph. Schnoebelen. Boundedness of Reset P/T Nets. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *Lectures on Concurrency and Petri Nets*, volume 1644 of *Lecture Notes in Computer Science*, pages 301–310, Prague, Czech Republic, July 1999. Springer-Verlag.

[15] J. Esparza and M. Nielsen. Decidability issues for Petri nets - a survey. *Journal of Information Processing and Cybernetics*, 30:143–160, 1994.

[16] A. Finkel and Ph. Schnoebelen. Well-structured Transition Systems everywhere! *Theoretical Computer Science*, 256(1–2):63–92, April 2001.

[17] J.F. Groote and M.A. Reniers. *Algebraic Process Verification*, chapter 7, pages 1151–1208. Handbook of Process Algebra. Elsevier Science B.V., Amsterdam, The Netherlands, 2001.

[18] A.H.M. ter Hofstede, M.E. Orlowska, and J. Rajapakse. Verification problems in conceptual workflow specifications. *Data and Knowledge Engineering*, 24(3):239–256, 1998.

[19] M. Jungel, E. Kindler, and M. Weber. The Petri net markup language. In S. Philippi, editor, *Proceedings of AWPN 2000 - 7thWorkshop Algorithmen und Werkzeuge für Petrinetze*, pages 47–52. Research Report 7/2000, Institute for Computer Science, University of Koblenz, Germany, 2000.

[20] C. Karamanolis, D. Giannakopoulou, J. Magee, and S.M. Wheater. Formal verification of workflow schemas.

[21] G. Keller, M. Nüttgens, and A.W. Scheer. Semantische Processmodellierung auf der Grundlage Ereignisgesteuerter Processketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89 (in German), University of Saarland, Saarbrücken, 1992.

[22] G. Keller, M. Nüttgens, and A.W. Scheer. Semantische processmodellierung auf der grundlage ereignisgesteuerter processketten (epk). Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89 (in German), University of Saarland, Saarbrücken, 1992.

[23] E. Kindler. On the semantics of EPCs: A framework for resolving the vicious circle. In J. Desel, B. Pernici, and M. Weske, editors, *International Conference on Business Process Management (BPM 2004)*, volume 3080 of *Lecture Notes in Computer Science*, pages 82–97. Springer, Berlin, Germany, 2004.

[24] E. Kindler. On the Semantics of EPCs: A Framework for Resolving the Vicious Circle. *Data and Knowledge Engineering*, 56(1):23–40, 2006.

[25] H. Lin, Z. Zhao, H. Li, and Z. Chen. A novel graph reduction algorithm to identify structural conflicts. In *Proceedings of the Thirty-Fourth Annual Hawaii International Conference on System Science (HICSS-35)*, pages 3778–3787. IEEE Computer Society Press, 2002.

[26] T. Madhusudan. A model-checking approach to workflow design and verification. In *Fourth International Conference on Electronic Commerce Research (ICECR-4)*, 2001.

[27] P. Matousek. *Verification of Business Process Models*. PhD thesis, Technical University of Ostrava, Ostrava-Poruba, Czech Republic, 2003.

[28] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[29] C. Ouyang, W.M.P. van der Aalst, S. Breutel, M. Dumas, A.H.M. ter Hofstede, and H.M.W. Verbeek. WofBPEL: A tool for automated analysis of BPEL processes. In *ICSOC 2005 proceedings*, volume (to appear) of *Lecture Notes in Computer Science*. Springer, Berlin, Germany, 2005. Accepted as tool demo.

[30] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science. Advances in Petri Nets*. Springer, Berlin, Germany, 1998.

[31] W. Sadiq and M.E. Orlowska. Applying graph reduction techniques for identifying structural conflicts in process models. In M. Jarke and A. Oberweis, editors, *Advanced Information Systems Engineering, 11th. International Conference, CAiSE'99, Proceedings*, volume 1626 of *Lecture Notes in Computer Science*, pages 195–209, Heidelberg, Germany, June 1999. Springer, Berlin, Germany, 1999.

[32] W. Sadiq and M.E. Orlowska. Analyzing process models using graph reduction techniques. *Information Systems*, 25(2):117–134, 2000.

[33] M. Schroeder. Verification of business processes for a correspondence handling center using CCS. In *EUROVAV*, pages 253–264, 1999.

[34] H.M.W. Verbeek. *Verification of WF-nets*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, June 2004. BETA dissertation series D65.

[35] H.M.W. Verbeek and W.M.P. van der Aalst. Woflan 2.0: A Petri-net-based workflow diagnosis tool. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 475–484. Springer, Berlin, Germany, 2000.

[36] H.M.W. Verbeek and T. Basten. Deciding life-cycle inheritance on Petri nets. In W.M.P. van der Aalst and E. Best, editors, *24th International Conference on Application and Theory of Petri Nets (ICATPN 2003)*, volume 2679 of *Lecture Notes in Computer Science*, pages 44–63, Eindhoven, The Netherlands, June 2003. Springer, Berlin, Germany.

[37] H.M.W. Verbeek, T. Basten, and W.M.P. van der Aalst. Diagnozing workflow processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001.

[38] P. Wohed, W.M.P. van der Aalst, M. Dumas, A.H.M. ter Hofstede, and N. Russell. Pattern-based Analysis of BPMN - An extensive evaluation of the Control-flow, the Data and the Resource Perspectives. BPM Center Report BPM-05-26, BPMcenter.org, 2005.

[39] M.T. Wynn, D. Edmond, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Achieving a general, formal and decidable approach to the OR-join in workflow using reset nets. In G. Ciardo and P. Darondeau, editors, *Applications and Theory of Petri Nets 2005*, volume 3536 of *Lecture Notes in Computer Science*, pages 423–443. Springer, Berlin, Germany, 2005.

A The erroneous lifestyle example

```
<?xml version="1.0" encoding="UTF-8"?>
<specificationSet
    xmlns="http://www.citi.qut.edu.au/yawl"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    version="Beta 6"
    xsi:schemaLocation="http://www.citi.qut.edu.au/yawl/schema/YAWL_SchemaBeta6.xsd"
>
    <specification uri="FlyerExampleWithCancellations1.4">
        <metaData>
            <title>The YAWL Flyer Example</title>
            <creator>Eric Verbeek</creator>
            <description>The erroneous flyer example.</description>
            <version>1.0</version>
        </metaData>
        <schema xmlns="http://www.w3.org/2001/XMLSchema"/>
        <decomposition id="Create_Music" isRootNet="true" xsi:type="NetFactsType">
            <processControlElements>
                <inputCondition id="InputCondition_8">
                    <flowsInto>
                        <nextElementRef id="Decide_to_make_music_24"/>
                    </flowsInto>
                </inputCondition>
                <task id="Decide_to_make_music_24">
                    <flowsInto>
                        <nextElementRef id="Learn_to_play_instrument_22"/>
                        <isDefaultFlow/>
                    </flowsInto>
                    <flowsInto>
                        <nextElementRef id="Do_audition_17"/>
                        <predicate ordering="0">true()</predicate>
                    </flowsInto>
                    <join code="xor"/>
                    <split code="xor"/>
                    <decomposesTo id="Decide_to_make_music"/>
                </task>
                <task id="Learn_to_play_instrument_22">
                    <flowsInto>
                        <nextElementRef id="_11"/>
                    </flowsInto>
                    <join code="xor"/>
                    <split code="and"/>
                    <startingMappings>
                        <mapping>
                            <expression query=""/>
                            <mapsTo>null</mapsTo>
                        </mapping>
                    </startingMappings>
                    <completedMappings>
                        <mapping>
```

```
            <expression query=""/>
            <mapsTo>null</mapsTo>
        </mapping>
    </completedMappings>
    <decomposesTo id="Learn_to_play_instrument"/>
</task>
<task id="Do_audition_17">
    <flowsInto>
        <nextElementRef id="Audition_failed_13"/>
        <isDefaultFlow/>
    </flowsInto>
    <flowsInto>
        <nextElementRef id="Audition_passed_10"/>
        <predicate ordering="0">true()</predicate>
    </flowsInto>
    <join code="xor"/>
    <split code="xor"/>
    <decomposesTo id="Do_audition"/>
</task>
<condition id="Audition_failed_13">
    <flowsInto>
        <nextElementRef id="Do_audition_17"/>
    </flowsInto>
    <flowsInto>
        <nextElementRef id="Learn_to_play_instrument_22"/>
    </flowsInto>
</condition>
<condition id="_11">
    <flowsInto>
        <nextElementRef id="Join_band_20"/>
    </flowsInto>
    <flowsInto>
        <nextElementRef id="Decide_to_go_solo_21"/>
    </flowsInto>
</condition>
<condition id="Audition_passed_10">
    <flowsInto>
        <nextElementRef id="Do_everything_you_are_told_25"/>
    </flowsInto>
</condition>
<task id="Do_everything_you_are_told_25">
    <flowsInto>
        <nextElementRef id="Make_Record_28"/>
    </flowsInto>
    <join code="xor"/>
    <split code="and"/>
    <removesTokens id="Audition_failed_13"/>
    <removesTokens id="Audition_passed_10"/>
    <removesTokens id="Decide_to_make_music_24"/>
    <removesTokens id="Do_audition_17"/>
    <removesTokens id="Learn_to_play_instrument_22"/>
    <removesTokens id="_11"/>
    <removesTokensFromFlow>
        <flowSource id="Decide_to_make_music_24"/>
        <flowDestination id="Do_audition_17"/>
    </removesTokensFromFlow>
    <decomposesTo id="Do_everything_you_are_told"/>
</task>
```

```
<task id="Decide_to_go_solo_21">
    <flowsInto>
        <nextElementRef id="Write_a_song_26"/>
        <predicate>true()</predicate>
    </flowsInto>
    <flowsInto>
        <nextElementRef id="Initial_solo_performance_27"/>
        <predicate>true()</predicate>
        <isDefaultFlow/>
    </flowsInto>
    <join code="xor"/>
    <split code="or"/>
    <decomposesTo id="Decide_to_go_solo"/>
</task>
<task id="Join_band_20">
    <flowsInto>
        <nextElementRef id="Write_a_song_26"/>
        <predicate>true()</predicate>
    </flowsInto>
    <flowsInto>
        <nextElementRef id="Initial_solo_performance_27"/>
        <predicate>true()</predicate>
        <isDefaultFlow/>
    </flowsInto>
    <join code="xor"/>
    <split code="or"/>
    <decomposesTo id="Join_band"/>
</task>
<task id="Make_Record_28">
    <flowsInto>
        <nextElementRef id="Rehearse_tour_19"/>
        <isDefaultFlow/>
    </flowsInto>
    <flowsInto>
        <nextElementRef id="Choose_path_12"/>
        <predicate ordering="0">true()</predicate>
    </flowsInto>
    <join code="xor"/>
    <split code="xor"/>
    <decomposesTo id="Make_Record"/>
</task>
<task id="Initial_solo_performance_27">
    <flowsInto>
        <nextElementRef id="Initial_solo_performance_27"/>
    </flowsInto>
    <flowsInto>
        <nextElementRef id="Get_recording_contract_18"/>
    </flowsInto>
    <join code="xor"/>
    <split code="and"/>
    <decomposesTo id="Initial_solo_performance"/>
</task>
<task id="Write_a_song_26">
    <flowsInto>
        <nextElementRef id="Get_recording_contract_18"/>
    </flowsInto>
    <join code="xor"/>
    <split code="and"/>
```

```
    <startingMappings>
        <mapping>
            <expression query=""/>
            <mapsTo>null</mapsTo>
        </mapping>
    </startingMappings>
    <completedMappings>
        <mapping>
            <expression query=""/>
            <mapsTo>null</mapsTo>
        </mapping>
    </completedMappings>
    <decomposesTo id="Write_a_song"/>
</task>
<condition id="Choose_path_12">
    <flowsInto>
        <nextElementRef id="Develop_bad_habits_23"/>
    </flowsInto>
    <flowsInto>
        <nextElementRef id="Develop_as_artist_14"/>
    </flowsInto>
</condition>
<task id="Develop_bad_habits_23">
    <flowsInto>
        <nextElementRef id="_15"/>
    </flowsInto>
    <join code="xor"/>
    <split code="and"/>
    <decomposesTo id="Develop_bad_habits"/>
</task>
<task id="Rehearse_tour_19">
    <flowsInto>
        <nextElementRef id="Do_tour_16"/>
    </flowsInto>
    <join code="xor"/>
    <split code="and"/>
    <decomposesTo id="Rehearse_tour"/>
</task>
<task id="Get_recording_contract_18">
    <flowsInto>
        <nextElementRef id="Make_Record_28"/>
    </flowsInto>
    <join code="or"/>
    <split code="and"/>
    <decomposesTo id="Get_recording_contract"/>
</task>
<task id="Develop_as_artist_14">
    <flowsInto>
        <nextElementRef id="_15"/>
    </flowsInto>
    <join code="xor"/>
    <split code="and"/>
    <decomposesTo id="Develop_as_artist"/>
</task>
<task id="Do_tour_16">
    <flowsInto>
        <nextElementRef id="_15"/>
    </flowsInto>
```

```
            <join code="xor"/>
            <split code="and"/>
            <decomposesTo id="Do_tour"/>
        </task>
        <task id="_15">
            <flowsInto>
                <nextElementRef id="OutputCondition_9"/>
            </flowsInto>
            <join code="or"/>
            <split code="and"/>
        </task>
        <outputCondition id="OutputCondition_9"/>
    </processControlElements>
</decomposition>
<decomposition id="Do_tour"
               xsi:type="WebServiceGatewayFactsType"/>
<decomposition id="Send_record_to_marketing_dept"
               xsi:type="WebServiceGatewayFactsType"/>
<decomposition id="Develop_as_artist"
               xsi:type="WebServiceGatewayFactsType"/>
<decomposition id="Decide_to_go_solo"
               xsi:type="WebServiceGatewayFactsType"/>
<decomposition id="Do_everything_you_are_told"
               xsi:type="WebServiceGatewayFactsType"/>
<decomposition id="Join_band"
               xsi:type="WebServiceGatewayFactsType"/>
<decomposition id="Write_a_song"
               xsi:type="WebServiceGatewayFactsType"/>
<decomposition id="Choose_songs"
               xsi:type="WebServiceGatewayFactsType"/>
<decomposition id="Develop_bad_habits"
               xsi:type="WebServiceGatewayFactsType"/>
<decomposition id="Learn_to_play_instrument"
               xsi:type="WebServiceGatewayFactsType"/>
<decomposition id="Do_audition"
               xsi:type="WebServiceGatewayFactsType"/>
<decomposition id="Get_recording_contract"
               xsi:type="WebServiceGatewayFactsType"/>
<decomposition id="Initial_solo_performance"
               xsi:type="WebServiceGatewayFactsType"/>
<decomposition id="Decide_to_make_music"
               xsi:type="WebServiceGatewayFactsType"/>
<decomposition id="Make_Record" xsi:type="NetFactsType">
    <processControlElements>
        <inputCondition id="InputCondition_1">
            <flowsInto>
                <nextElementRef id="Choose_songs_6"/>
            </flowsInto>
        </inputCondition>
        <task id="Choose_songs_6">
            <flowsInto>
                <nextElementRef id="Record_songs_5"/>
            </flowsInto>
            <join code="and"/>
            <split code="and"/>
            <decomposesTo id="Choose_songs"/>
        </task>
        <task id="Record_songs_5">
```

```
                    <flowsInto>
                        <nextElementRef id="Done_3"/>
                    </flowsInto>
                    <join code="xor"/>
                    <split code="and"/>
                    <startingMappings>
                        <mapping>
                            <expression query=""/>
                            <mapsTo>null</mapsTo>
                        </mapping>
                    </startingMappings>
                    <completedMappings>
                        <mapping>
                            <expression query=""/>
                            <mapsTo>null</mapsTo>
                        </mapping>
                    </completedMappings>
                    <decomposesTo id="Record_songs"/>
                </task>
                <condition id="Done_3">
                    <flowsInto>
                        <nextElementRef id="Choose_songs_6"/>
                    </flowsInto>
                    <flowsInto>
                        <nextElementRef id="Send_record_to_marketing_dept_4"/>
                    </flowsInto>
                </condition>
                <task id="Send_record_to_marketing_dept_4">
                    <flowsInto>
                        <nextElementRef id="OutputCondition_2"/>
                    </flowsInto>
                    <join code="xor"/>
                    <split code="and"/>
                    <decomposesTo id="Send_record_to_marketing_dept"/>
                </task>
                <outputCondition id="OutputCondition_2"/>
            </processControlElements>
        </decomposition>
        <decomposition id="Record_songs"
                       xsi:type="WebServiceGatewayFactsType"/>
        <decomposition id="Rehearse_tour"
                       xsi:type="WebServiceGatewayFactsType"/>
    </specification>
</specificationSet>
```

B Report on the lifestyle example

```
<wofyawl version="0.6" status="released">
    <net file="FlyerExample14.xml">
        <structure>
            <warning specification="FlyerExampleWithCancellations1.4"
                     decomposition="Create_Music"
                     task="Initial_solo_performance_27"
                     input="Decide_to_go_solo_21"/>
            <warning specification="FlyerExampleWithCancellations1.4"
                     decomposition="Create_Music"
                     task="Initial_solo_performance_27"
```

```
            input="Join_band_20"/>
    <warning specification="FlyerExampleWithCancellations1.4"
             decomposition="Make_Record"
             task="Send_record_to_marketing_dept_4"
             input="Done_3"/>
    <warning specification="FlyerExampleWithCancellations1.4"
             decomposition="Make_Record"
             task="Send_record_to_marketing_dept_4"/>
    <warning specification="FlyerExampleWithCancellations1.4"
             decomposition="Create_Music"
             task="Decide_to_go_solo_21"
             output="Initial_solo_performance_27"/>
    <warning specification="FlyerExampleWithCancellations1.4"
             decomposition="Create_Music"
             task="Decide_to_go_solo_21"
             or-split="xor-split"/>
    <warning specification="FlyerExampleWithCancellations1.4"
             decomposition="Create_Music"
             task="Join_band_20"
             output="Initial_solo_performance_27"/>
    <warning specification="FlyerExampleWithCancellations1.4"
             decomposition="Create_Music"
             task="Join_band_20"
             or-split="xor-split"/>
</structure>
<warning description="YAWL net is unbounded"/>
<behavior>
    <warning specification="FlyerExampleWithCancellations1.4"
             decomposition="Create_Music"
             task="Initial_solo_performance_27"
             input="Decide_to_go_solo_21"/>
    <warning specification="FlyerExampleWithCancellations1.4"
             decomposition="Create_Music"
             task="Initial_solo_performance_27"
             input="Join_band_20"/>
    <warning specification="FlyerExampleWithCancellations1.4"
             decomposition="Create_Music"
             task="Initial_solo_performance_27"
             input="Initial_solo_performance_27"/>
    <warning specification="FlyerExampleWithCancellations1.4"
             decomposition="Create_Music"
             task="Initial_solo_performance_27"/>
    <warning specification="FlyerExampleWithCancellations1.4"
             decomposition="Create_Music"
             task="Get_recording_contract_18"
             input="Initial_solo_performance_27"/>
    <warning specification="FlyerExampleWithCancellations1.4"
             decomposition="Create_Music"
             task="Get_recording_contract_18"
             OR-join="xor-join"/>
    <warning specification="FlyerExampleWithCancellations1.4"
             decomposition="Create_Music"
             task="_15"
             OR-join="xor-join"/>
    <warning specification="FlyerExampleWithCancellations1.4"
             decomposition="Make_Record"
             task="Choose_songs_6"
             input="InputCondition_1"/>
```

```
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Make_Record"
         task="Choose_songs_6"
         input="Done_3"/>
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Make_Record"
         task="Choose_songs_6"/>
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Make_Record"
         task="Send_record_to_marketing_dept_4"
         input="Done_3"/>
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Make_Record"
         task="Send_record_to_marketing_dept_4"/>
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Create_Music"
         task="Decide_to_go_solo_21"
         output="Initial_solo_performance_27"/>
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Create_Music"
         task="Decide_to_go_solo_21"
         or-split="xor-split"/>
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Create_Music"
         task="Join_band_20"
         output="Initial_solo_performance_27"/>
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Create_Music"
         task="Join_band_20"
         or-split="xor-split"/>
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Create_Music"
         task="Do_everything_you_are_told_25"
         cancel="Decide_to_make_music_24"/>
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Create_Music"
         task="Do_everything_you_are_told_25"
         cancel="Learn_to_play_instrument_22"/>
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Create_Music"
         task="Do_everything_you_are_told_25"
         cancel="Do_audition_17"/>
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Create_Music"
         task="Do_everything_you_are_told_25"
         cancel="Audition_failed_13"/>
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Create_Music"
         task="Do_everything_you_are_told_25"
         cancel="_11"/>
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Create_Music"
         task="Do_everything_you_are_told_25"
         cancel="Audition_passed_10"/>
<warning specification="FlyerExampleWithCancellations1.4"
         decomposition="Create_Music"
         task="Do_everything_you_are_told_25"
         cancel="Decide_to_make_music_24*Learn_to_play_instrument_22"/>
```

```
            <warning specification="FlyerExampleWithCancellations1.4"
                     decomposition="Create_Music"
                     task="Do_everything_you_are_told_25"
                     cancel="Decide_to_make_music_24*Do_audition_17"/>
        </behavior>
    </net>
</wofyawl>
```