Chapter 20
Verification

Eric Verbeek and Moe Wynn

20.1 Introduction

Chapter 2 introduced the soundness property on a special class of Petri nets called WF-nets (WorkFlow nets). To reiterate, a WF-net is sound if and only if the following requirements are met:

- Any executing instance of the WF-net must eventually terminate
- At the moment of termination, there must be precisely one token in the end place and all other places are empty
- No dead tasks

Van der Aalst has shown that soundness of a WF-net corresponds to boundedness and liveness of an extension of that WF-net. As boundedness and liveness of Petri nets are both decidable, soundness of WF-nets is also decidable. Based on this observation, the Woflan tool was built, which uses standard Petri-net techniques to decide soundness.

However, as Chap. 1 has already mentioned, the class of WF-nets is not sufficient to reason about YAWL nets. For this, we need to extend the WF-nets with the concept of reset arcs, yielding RWF-nets (Reset WorkFlow nets). Using reset arcs, the cancelation features of YAWL can be captured in a natural way: if some task or condition is canceled by some task, then the corresponding place is reset by the corresponding transition.

Unfortunately, soundness of an RWF-net does not correspond to boundedness and liveness of an extension of this net, as the example RWF-net in Fig. 20.1 shows. Although this RWF-net is sound, it is unbounded as the place p may contain an arbitrary number of tokens. Furthermore, the reachability problem is undecidable for reset nets. As the liveness property corresponds to a reachability problem, the liveness property cannot be decided for arbitrary reset nets. On top of this, YAWL also includes the OR-join construct, which has been neglected so far in this chapter.

E. Verbeek (✉)
Eindhoven University of Technology, Eindhoven, the Netherlands,
e-mail: h.m.w.verbeek@tue.nl

A.H.M. ter Hofstede et al. (eds.), *Modern Business Process Automation*,
DOI 10.1007/978-3-642-03121-2_20,

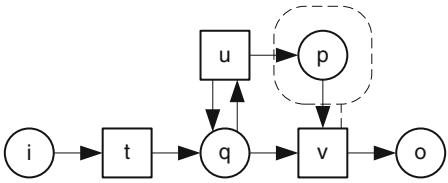

Fig. 20.1 Soundness of an RWF-net does not require boundedness

Based on these observations, answering the question whether some YAWL net is sound is a challenging problem. This chapter tries to tackle this problem using advanced techniques. As mentioned, the question is undecidable in general. Therefore, we allow ourselves to simplify the question a bit whenever we see no other way. For this reason, this chapter:

- Introduces a number of soundness-preserving reduction rules for YAWL nets and RWF-nets. Applying these rules simplifies the net to verify, which simplifies the soundness problem
- Introduces some weaker notions of soundness that might help the YAWL developer in cases where soundness itself cannot be decided

The remainder of this chapter is organized as follows. First, we answer the question of when we consider a YAWL net to be sound. Second, we show how soundness can be decided for many nets by constructing their state spaces. Third, we introduce a number of reduction rules (both for RWF-nets and for YAWL nets) that simplify the soundness problem. Fourth, for those cases where the simplified problem is still too complex, we propose the use of structural invariant properties. Although these structural properties are necessary, they are not sufficient. As a result, using them, we can only show that an RWF-net is not sound, but not that it is sound.

This chapter does not provide any formal definition, formal theorem, or formal proof. Rather than going into these details, this chapter keeps at a high level. The reader interested in these details is referred to the Chapter Notes.

20.2 Preliminaries

As mentioned in Chap. 2, YAWL has been inspired by a class of Petri nets called WF-nets. For these WF-nets, a soundness criterion has been defined (recall from Chap. 2), which can be transferred to YAWL nets with ease: a YAWL net is sound if (and only if) it satisfies the following three requirements:

- *Option to complete:* Any executing instance of the YAWL net must eventually terminate
- *Proper completion:* At the moment of termination there must be precisely one token in the end condition and all other conditions are empty
- *No dead tasks:* Any task can be executed for some case

However, possibly, these requirements are too strong. As an example, consider a YAWL net that includes two decision points (say, two XOR splits) that are controlled by the same case attribute. As a result, if we go left at the first decision point, we also go left at the second, and vice versa. However, as we abstract from the data perspective when considering soundness, the fact that both decisions are synchronized is lost, and the possibility that we go left at one and right at the other is also taken into account. As a result, the YAWL net may be considered not sound, while in fact it is.

The *relaxed soundness* property takes these synchronized decision points into account. A YAWL net is relaxed sound if (and only if) all its tasks are relaxed sound, and a task is relaxed sound if some proper completing case exists for which the task is executed. Obviously, the intent of a YAWL net is to forward a running case towards its completion. If some task cannot help in completing any case, then there has to be an error in the model. This is what the relaxed soundness property is about.

Another advantage of the relaxed soundness property is that it allows for a simplistic view on the OR-join. When trying to decide soundness, we need to check for every OR-join over and over again whether it is enabled or not, which makes it extremely hard, if possible at all, to capture this by a mere reset net. Relaxed soundness, however, will give us the input combinations that are covered by some proper completing execution paths. Therefore, we can initially allow all possible input combinations, and the relaxed soundness property will filter out those combinations that are hopeless, and for which the OR-join should be disabled.

Still, the relaxed soundness property requires the entire state space to be constructed, which is known to be impossible in general. For many YAWL nets, it will be possible to construct the entire state space, but YAWL nets exist for which this is not possible. Therefore, our approach so far (soundness and relaxed soundness) is not complete. To make the approach more complete, we introduce a third notion of soundness, called *weak soundness*. A YAWL net is weak sound if (and only if) the following three requirements are satisfied:

- *Weak option to complete:* It is possible to complete a case
- *Proper completion:* Once a case has been completed, no references to that case are left behind
- *No dead tasks:* Any task can be executed for some case

Note that only the first requirement differs from the soundness requirements.

As Chaps. 1 and 2 have explained, YAWL is grounded in reset nets to provide support for cancelation regions. Recall that a reset net is a Petri net that allows transitions to reset places, that is, to remove all tokens from these places. YAWL's cancelation regions can be modeled by reset nets in a natural way: If some task cancels a region, then the corresponding transition resets all places in the corresponding region. For example, Fig. 20.2 shows a first version[1] of the *Carrier Appointment* decomposition, where the part between the *Prepare Transportation Quote* and the

[1] This version predates the version throughout the rest of this book, and contains a subtle error that was overlooked by the YAWL experts who designed the entire model.

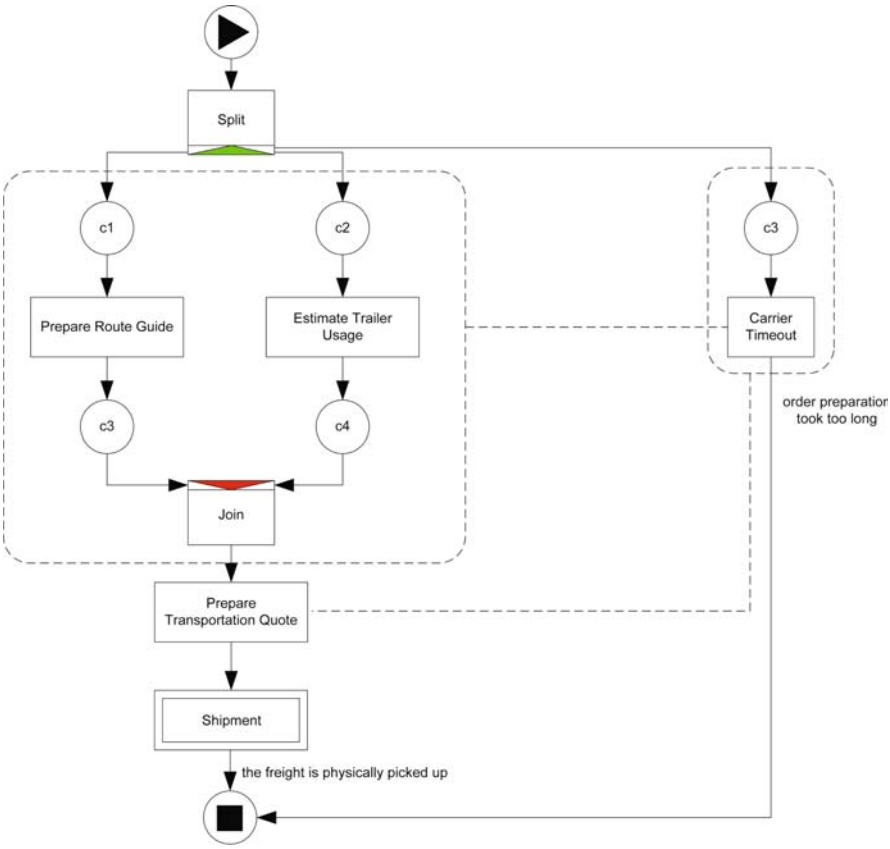

Fig. 20.2 The simplified carrier appointment decomposition

output condition has been replaced by a single composition task named *Shipment*. For ease of reference, we have labeled the other conditions as well. This decomposition contains two cancelation regions, and Fig. 20.3 shows how this fragment can be captured by an RWF-net. Conceptually, every task is captured by a *Busy* place, a *Start* transition for every possible input combination, and a *Complete* transition for every possible output combination, and conditions (including implicit conditions) are captured by places.

Furthermore, YAWL's OR-joins also benefit from these reset nets, as their semantics use reset nets to determine when they are enabled. Fortunately, the question whether some state is *coverable* from a given state is decidable for reset nets, which is sufficient for this purpose. A state is coverable if some state is reachable that contains the given state (see also Chap. 2). This also explains why the weak soundness property is decidable, as the weak soundness property can be expressed in terms of coverability:

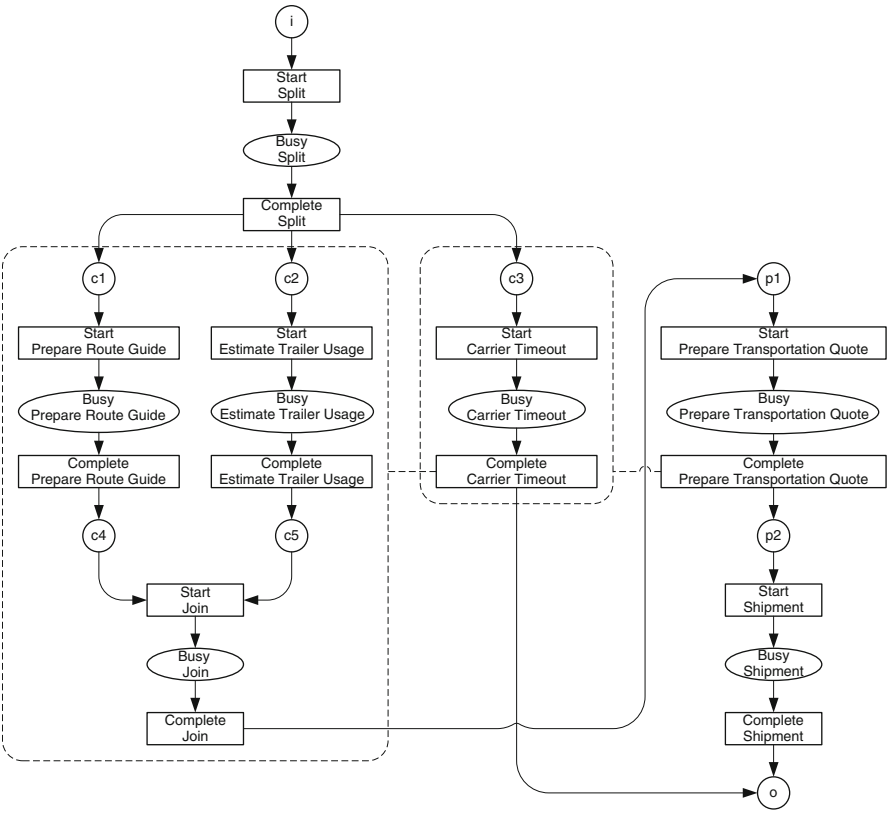

Fig. 20.3 The decomposition of Fig. 20.2 as an RWF-net

- *Weak option to complete:* The completion state is coverable from the initial state
- *Proper completion:* No state exceeding the proper completion state is coverable from the initial state
- *No dead tasks:* For every task a state is coverable that enables the task

In contrast, the first requirement of the soundness property requires reachability, as it requires all reachable states.

20.3 Soundness of YAWL Models

Unfortunately, the question whether some state is reachable from a given state is not decidable. In other words, it is not possible to construct an algorithm that can tell for any YAWL net whether completion is always reachable from the initial state. Nevertheless, for specific YAWL nets, this may still be possible. As an example, we take the RWF-net from Fig. 20.3.

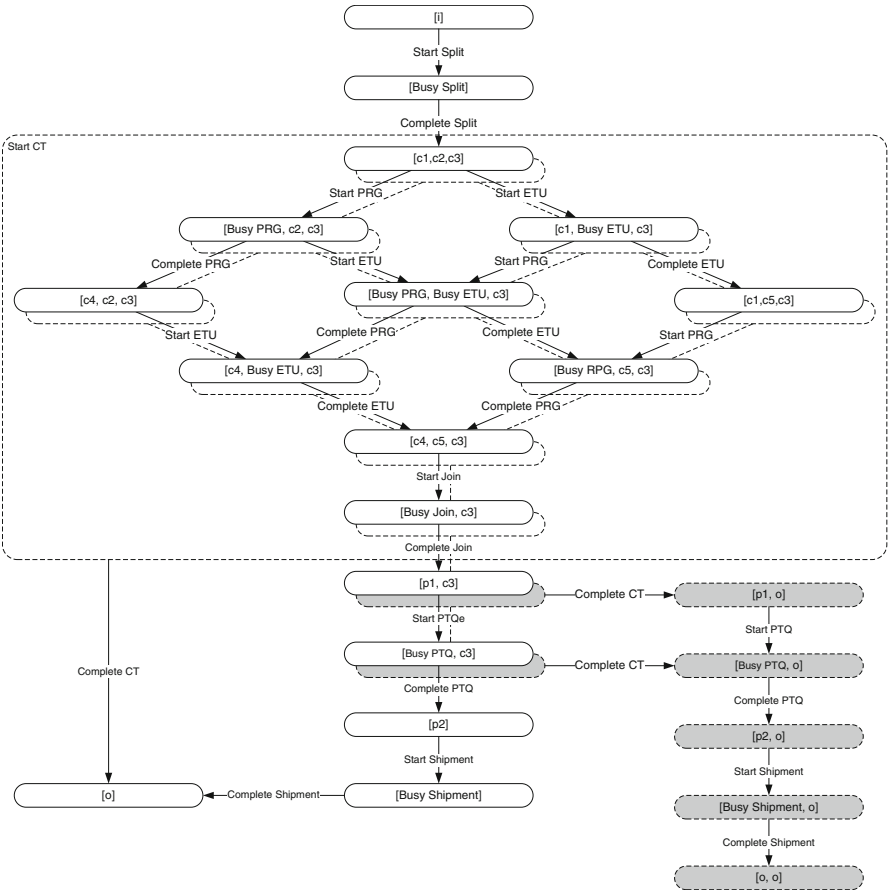

Fig. 20.4 The state space of the RWF-net shown in Fig. 20.3

First, we construct the state space of the RWF-net, which contains 34 states and is shown (in a compact way) in Fig. 20.4. For sake of completeness, we mention that we have used acronyms instead of long task names, and that the dotted (shadow) states can be reached by executing the *Start Carrier Timeout* (*Start CT*) transition. The reason why some of the shadow states are filled will become clear further on; please ignore these fills for the time being. Second, we check the three requirements for soundness on this state space.

- *Option to complete:* It is always possible to mark the sink place o, hence it is always possible to complete a case
- *Proper completion:* Several states exist that exceed the state $[o]$. Examples are $[p1, o]$ and $[o, o]$. As a result, completion may be improper
- *No dead tasks:* For any transition, an edge exists in the state space. Hence, no task is dead

We conclude that the simplified *Carrier Appointment* decomposition is not sound. It is straightforward to check that this erroneous behavior is also present in the original *Carrier Appointment* decomposition, hence the entire Order Fulfillment model is not sound.

This error is caused by the fact that the *Carrier Timeout* task does not cancel the *Prepare Transportation Quote* task nor its immediately preceding (implicit) condition. As a result, the latter task may be enabled or in progress, which cannot be canceled by the former.

As mentioned before, perhaps this error is due to some case attribute. Perhaps some attribute exists that prevents the completion of the *Carrier Timeout* task as soon as the *Prepare Transportation Quote* task has been enabled. As we take only the control-flow of the YAWL net into account for verification, we abstracted from all such attributes. Hence, it might be interesting to check whether the decomposition is relaxed sound.

To check relaxed soundness, we restrict the state space to that part that is covered by proper completing execution paths. In other words, we strip all states from the state space from which proper completion is not possible anymore. In Fig. 20.4, all filled states are to be stripped. For relaxed soundness, we now need to check whether all tasks are covered by this state space, that is, do all tasks occur on some edge in this state space? It is straightforward to check that this is indeed the case. Hence, the decomposition is relaxed sound, and provided that all other decompositions in the entire model are also relaxed sound, the entire model is also relaxed sound.

Until now, we have a kind of overlooked the possible problem that a YAWL task is not a transition and vice versa. In the example above, every task was captured by a single *Start* transition and a single *Complete* transition, but this need not be the case. A second fragment from the *Carrier Appointment* decomposition clarifies this. Figure 20.5 shows this fragment, which includes the *Create Bill of Lading* task that has an OR-join semantics. When converting such an OR-join to a reset net, there are no alternatives but to introduce a transition for every possible input combination, which is shown by Fig. 20.6. However, from the seven possible input combinations, only two are covered by proper completing execution paths (the other five have fills in Fig. 20.6).

Now, for verification it is important to note that all inputs are covered by the relaxed sound transitions. As a result, all inputs to this task are viable. If some inputs (outputs) of some tasks are not covered by relaxed sound transitions, then a warning can be issued that some incoming (outgoing) edge of the corresponding task cannot be successfully traversed; if such an edge would be traversed, proper completion would be impossible. When taking relaxed soundness into account, we can disregard possible input combinations that cannot lead to proper completion, which enhances the effectiveness (more warnings can be issued).

In a similar way, we can use relaxed soundness and the state space to detect whether some tasks or conditions can be removed from a certain cancelation region. If a certain task cannot cancel a given task or condition, or if canceling this given task or condition prohibits proper completion, then a warning can be issued that the cancelation region can be simplified by removing from it the given task or condition. However, these warnings can also be achieved by using the coverability property:

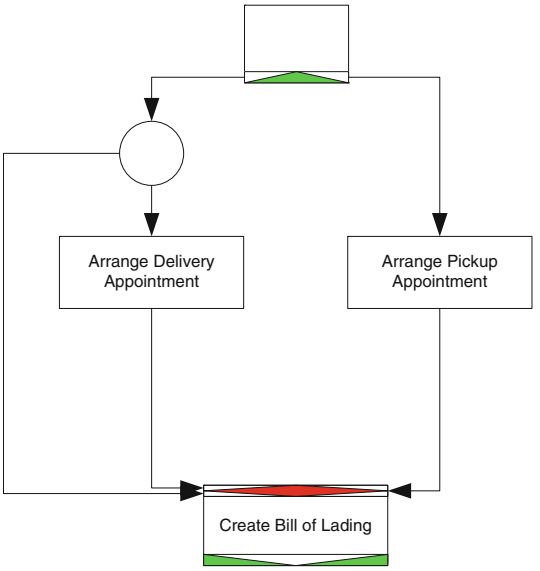

Fig. 20.5 OR-join construct in the Carrier Appointment decomposition

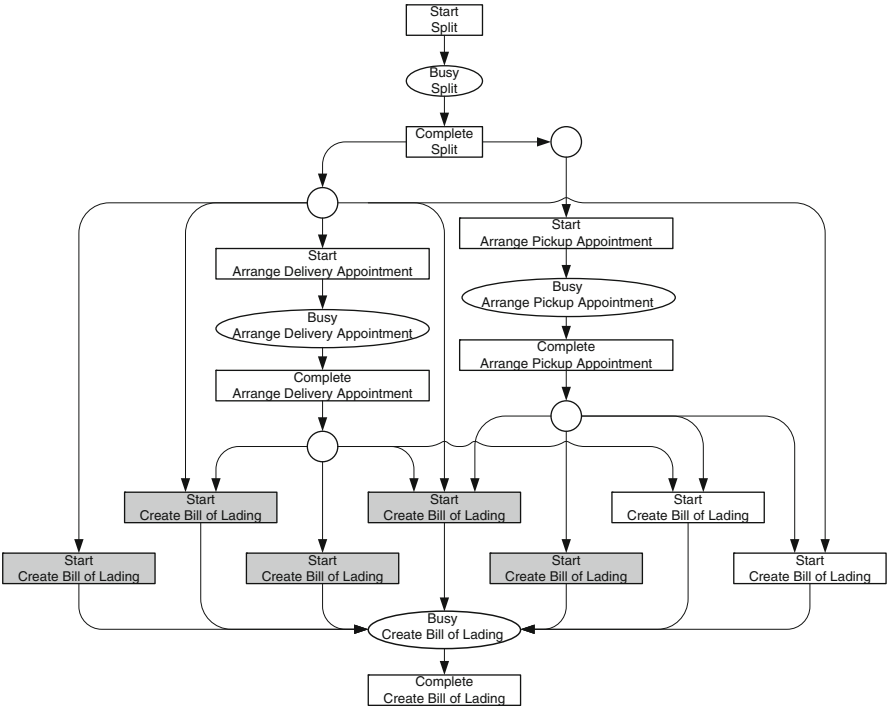

Fig. 20.6 RWF-net for OR-join construct

- A condition can be removed from a task's cancelation region if the state where
  - the condition is satisfied and
  - the task is enabled

  cannot be covered from the initial state.
- A task can be removed from another task's cancelation region if the state where both tasks are enabled is not coverable from the initial state.

Thus, in contrast to the OR-join (OR-split) warnings, the cancelation region warning can be complete.

Although for relaxed soundness it is possible to model an OR-join by a transition for every possible input combination, for soundness this is, in general, not possible. Take, for example, the RWF-net fragment as shown by Fig. 20.6. Clearly, the two places in the *Delivery* branch are mutually exclusive. However, the two top filled transitions require both places to be marked simultaneously. Hence, these transitions are dead, which violates the soundness property. Furthermore, both the *Delivery* and the *Pickup* branch will contain a token. However, the three bottom-filled transitions only remove one of these tokens, which violates the completion from being proper.

Because of the non-local semantics of an OR-join, a YAWL net with one or more OR-joins cannot be converted to an RWF-net *without some approximation*. As a result, we perform the soundness analysis for YAWL nets with OR-joins directly on the YAWL level and not on the RWF-net level. That is, a YAWL net is *not* converted to an RWF-net before performing the state space analysis to determine the soundness property. If the state space of a given YAWL net is finite, then it is possible to decide whether a particular YAWL net is sound.

If the soundness property of a YAWL net with OR-join cannot be determined, we can also attempt to use coverability analysis on the RWF-net level to decide whether the net is weak sound. In this case, we transform all OR-joins in the YAWL net into XOR-joins first, and then translate this resulting YAWL net into an RWF-net. Because of this potential semantic loss, it is only possible to determine whether a YAWL net with OR-joins is weak sound in some cases.

Similarly, because of the expensive nature of OR-joins, we propose to provide users with warnings when unnecessary OR-joins are present in the net. This is made possible by looking at the entire state space of the YAWL net with OR-joins, if available. In case we have an OR-join task that is enabled only if all inputs are available, then a warning can be issued that this task can be an AND-join, and if we have an OR-join task that is enabled only if exactly one input is available, then a warning can be issued that this task can be an XOR-join.

20.4 Soundness-Preserving Reduction Rules

The success of our verification approach hinges on the ability to generate a state space. If we can construct a state space, then we can decide soundness, and we can provide a complete diagnosis report. Otherwise, we have to revert to using either

relaxed soundness or weak soundness (or structural invariant properties), in which cases the results might be incomplete. Although it is known that construction of a state space is infeasible for some YAWL nets, we want to improve our chances of success in case it is feasible. This means that we try to avoid a state space explosion as much as possible. A popular way to do so is to reduce the size of the model beforehand, that is before generating its state space.

Of course, if we reduce the model, we change it in some way. This makes diagnosing the possible errors in the model a very complex task, as we need to be aware of this change and we need to be able to transfer errors in the reduced model to errors in the original model. A fine point to note is that the reduction should be such that any errors present in the original model are also present in the reduced model: The reduction should preserve the sought-after qualities, which in our case is the soundness property.

Reductions can be applied on either the YAWL net or on the underlying RWF-net. For the latter, it is important to note that (ideally) there should be no OR-joins present in the YAWL net, as these do not transfer well onto reset net constructs. Therefore, we first assume that no OR-joins are present, and introduce a number of reset net-based reduction rules. Later on, we will transfer these reduction rules to the level of the YAWL net and allow for OR-joins as well.

20.4.1 Reduction Rules for Reset Nets

We propose a number of reduction rules for reset nets to keep the size of the corresponding state space at bay. These rules are extensions of a number of well-known reduction rules for Petri nets. As the presence of reset arcs could invalidate these rules, we extend the rules with additional requirements on these reset arcs.

For the rules proposed, we do assume a YAWL context. As a result, only the place that corresponds to the input condition (also known as the source place) is marked initially. The other places are empty, which allows us to abstract from the initial marking. As a side-effect, some rules are by definition not applicable. As an example, we mention the *elimination of self-loop place* rule. This rule allows to reduce some place, provided that it is marked initially and provided that it satisfies some additional requirements. As only the source place is marked initially, and as this place does not satisfy these additional requirements, there is no reason to extend this rule.

Figure 20.7 wraps up the remainder of this section on reduction rules for reset nets by giving a visual representation for all rules. Every rule potentially removes (or replaces by an arc) the filled places and transitions.

20.4.1.1 Fusion of Series Places (FSP)

Recall that our aim is to reduce the size of the state space, that is, the number of reachable states. Typically, the state explosion problem is caused by a number of

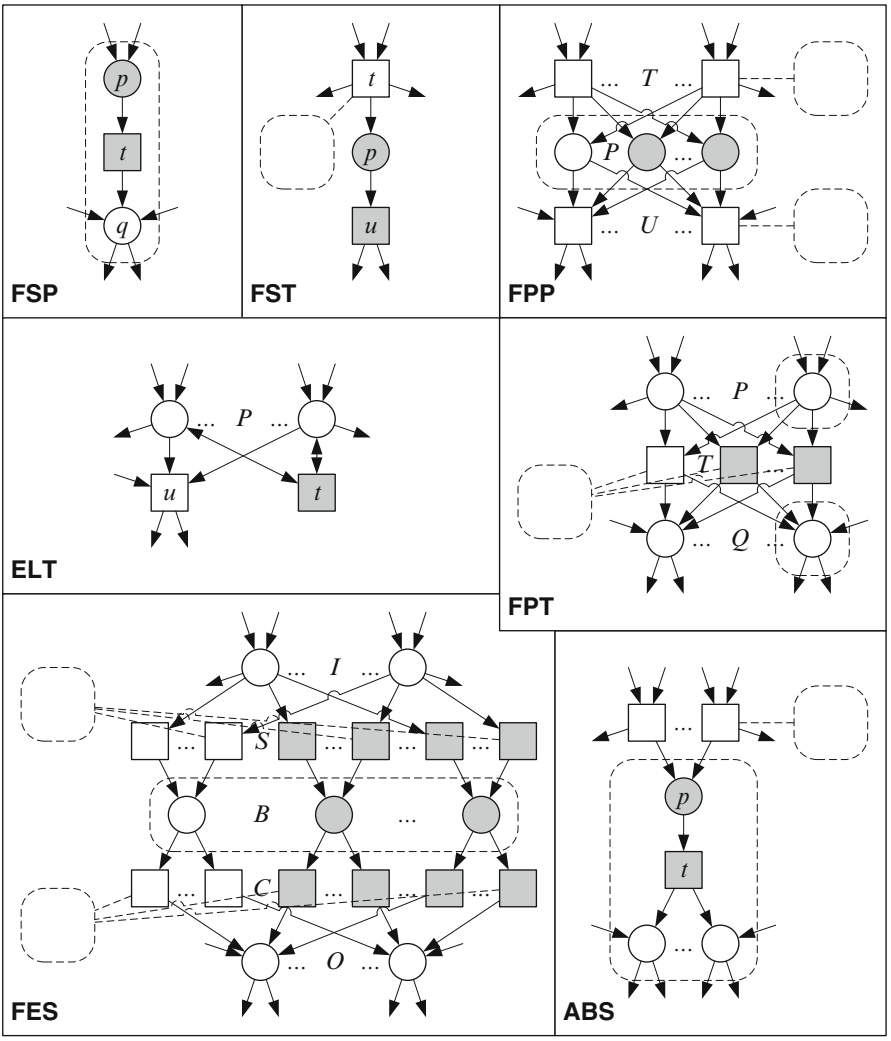

Fig. 20.7 A visual representation of all reduction rules for reset nets

parallel paths, which each contains a number of places. If these paths are truly parallel, then the number of reachable states would equal the product of the number of places of each path. Therefore, if we would be able to reduce the number of places on such paths, and this could have a profound effect on the state space size.

The first rule, called the *fusion of series places (FSP)* rule, attempts to reduce two places, say p and q, which are connected by a single transition, say t. Provided that:

- p is the only input and q the only output of t and
- t is the only output of p,

then we can either:

- Transfer all input arcs from p to q (we have to introduce arc weights here if such an arc to q already exists) and
- Remove both p and t together with their (incoming or outgoing) arcs

or

- Transfer all input and output arcs from q to p (we have to introduce arc weights here if such an arc from p already exists) and
- Remove both q and t together with their (incoming or outgoing) arcs.

The question now is, what requirements do we need to pose on reset arcs such that the rule is still applicable?

Transition t should not reset any place

As t has only p as input and only q as output, the remainder of the net cannot influence the moment t is being executed. In other words, transition t is some kind of loose cannon. However, the moment t executes should not matter, as after it has been removed it cannot matter anymore. Therefore, we cannot allow t to reset any place, *unless* we can guarantee that the effect of this reset is always identical. In general, this requires a state space search, which is not a good idea as we are still trying to reduce the net prior to constructing a state space.

Places p and q should be reset by the same set of transitions

To see this, it helps to consider the tokens in place p to be present in some way in place q as well. After all, any token from p can be transferred to q by executing transition t at any moment in time. Therefore, any transition that resets p should reset q as well, as in the mean time the token it was about to reset may have been transferred by t to q.

Together, these requirements are sufficient to ensure that the result of this rule preserves the soundness property. Figure 20.8 shows the result of applying this rule to the reset net introduced earlier: filled transitions and places have been removed by the rule. The state space of the resulting net contains only eight (was: 34) states (three states if transition *Complete Carrier Timeout* is executed in an improper way and five if it is not executed or executed in a proper way).

20.4.1.2 Fusion of Series Transitions (FST)

A second rule that allows for removal of places is the *fusion of series transitions (FST)* rule. While the previous rule looked for a series containing two places and one transition, this rule looks for a series containing two transitions and one place. Furthermore, while the previous rule allowed the second place to have additional input, this rule allows for the first transition to have additional output.

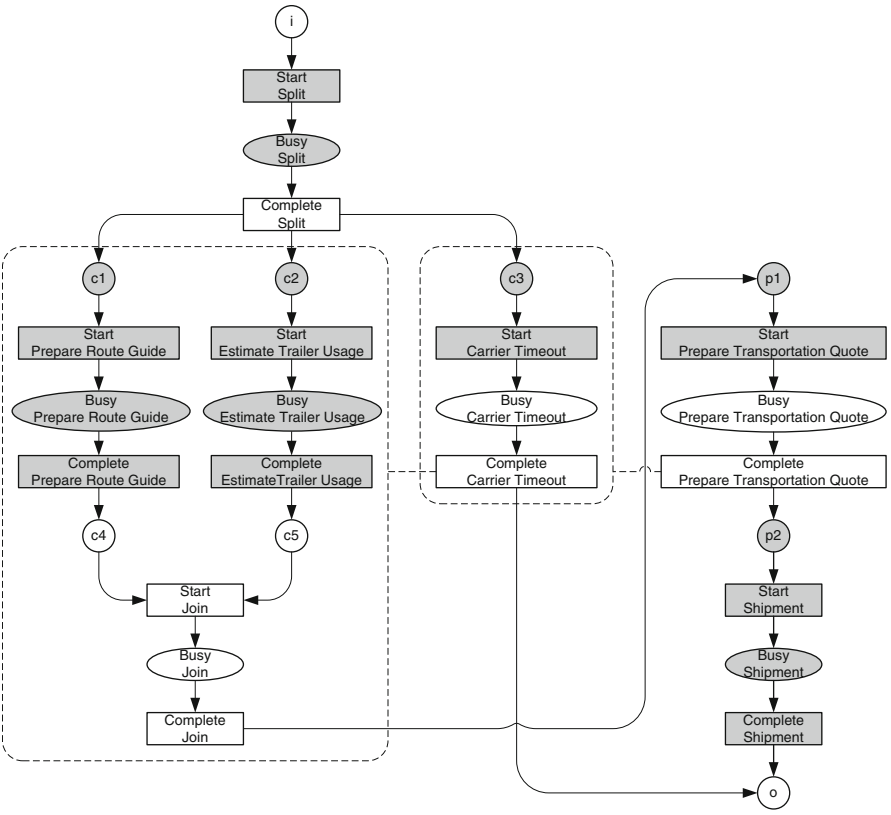

Fig. 20.8 The result of applying the *fusion of series places* rule to the RWF-net of Fig. 20.3

Given two transitions, say t and u, and one place, say p, this rule can remove a place and a transition. Provided that:

- t is the only input and u the only output of p and
- p is the only input of u,

then we can either:

- Transfer all output arcs from u to t and
- Remove both p and u together with their incident arcs

or

- Transfer all inputs and outputs from t to u and
- Remove both p and t together with their incident arcs

Again, the question is what additional requirements we should pose for possible reset arcs.

Place p is reset by the same set of transitions as any output of transition u is

Analogous to the previous rule, we can argue that the execution of transition u cannot be influenced. Hence, the effect of the execution of any transition should not be influenced by an intermediate execution of u. All outputs of u should agree on the set of transitions that reset them, and that p should be reset by the same set of transitions as well.

Transition u does not reset

If u resets some place, then any transition that requires that place as input is influenced by the execution of u, which is undesirable. As with the previous rule, we could construct a state space to determine whether u actually resets a place (resetting an empty place is excluded from this), in which case the reset arc could be allowed, but that constructing a state space at this point defeats the purpose of the rule.

Together, these requirements are sufficient to ensure that this rule preserves soundness.

20.4.1.3 Fusion of Parallel Places (FPP)

The two previous rules reduce the size of the state space by removing places from the net prior to constructing the state space. The *fusion of parallel places (FPP)* rule also removes a place, but this does not lead to a direct reduction in the size of the state space. However, after this rule has been applied, one of the other rules could then be applied, which might reduce the state space size. As such, the aim of this rule is not to reduce the state space per se, but to enable other rules to do so.

Given a set of places, say P, and two sets of transitions, say T and U, this rule can remove a number of places. Provided that:

- Every place from P has the first set of transitions (T) as inputs and
- Every place from P has the second set of transition (U) has outputs,

then we can remove all-but-one places from P.

This rule has only one additional requirement for reset arcs.

All places in P are being reset by the same set of transitions

If none of the places are reset places, then it is obvious that the rule holds. Assume that one of the places from P can be reset by some transition. The only way to guarantee that soundness is preserved by the reduction is to have the remaining place being reset by the same transition as well, which, in turn, requires the other places from P to be reset by this transition.

As a result of this rule, place *p5* can be filled as well in Fig. 20.8, which allows either the *fusion of series places* rule or the *fusion of series transitions* rule to remove both *c4* and *Start Join*. Note that *Busy Join* cannot be removed by either of these

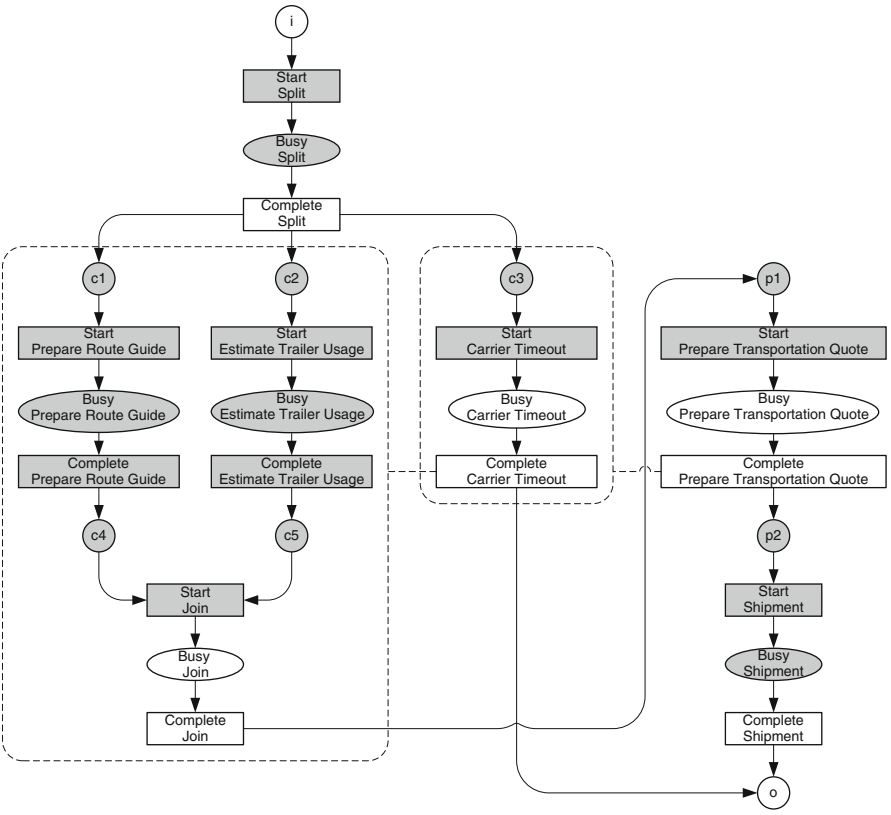

Fig. 20.9 The result of applying the three rules known so far to the RWF-net of Fig. 20.3

rules, as it is being reset by *Complete Carrier Timeout*, while place *Busy Prepare Transportation Quote* is not. Figure 20.9 shows the result.

20.4.1.4 Fusion of Parallel Transitions (FPT)

Similar to the previous rule, the *fusion of parallel transitions (FPT)* rule aims to enable any other rule, but while the previous rule attempts to do so by removing parallel places, this rule attempts to do so by removing parallel transitions.

Given a set of transitions, say T, and two sets of places, say P and Q, this rule can remove all-but-one transitions from T. Provided that:

- Every transition from T has the first set of places (P) as inputs, and
- Every transition from T has the second set of places (Q) as outputs,

then we can remove all-but-one transitions from T.

All transitions in T reset the same set of places

The argument for this is similar to the argument for the *fusion of parallel places* rule: all transitions should have an identical effect.

20.4.1.5 Eliminate Self-Loop Transitions (ELT)

A third rule that aims to enable other rules is the rule to remove an elementary cycle. If executing a transition does not change the current state, then we can ignore (or remove) that transition, provided a state exists that enables the transition. If the transition is never enabled, then removing it is not allowed as this might not preserve the third soundness requirement. Therefore, we restrict ourselves to transitions for which it is known that another transition is enabled as soon as it is enabled, that is, its inputs should be a subset of that other transition.

Given two transitions, say t and u, and a set of places, say P, this rule can remove the transition t. Provided that:

- Transition t has the set of places P as inputs and as outputs, and
- Transition u has at least the set of places P as inputs,

then we can remove transition t.

Transition t resets no place

Transition t may only reset empty places, as the effect of executing t should be identical to the effect of not executing it. As mentioned before, constructing a state space to determine whether t can only reset empty places defeats the purpose of the rule. Therefore, we require that t does not reset any place.

20.4.1.6 Abstraction (ABS)

The next rule is based on the *abstraction* rule from Desel and Esparza. As for the first two rules, this rule allows for the removal of a place and a transition. As such, it aims to reduce the size of the state space in a direct way.

Given a transition, say t, and a place, say p, this rule can remove both t and p. Provided that:

- t is the only output of p and
- p is the only input of t,

then we can:

- Add arcs from any input of p to any output of t and
- Remove both t and p.

As for both *fusion of series* rules, we may have to introduce arc weights if arcs already exist from some inputs of p to some outputs of t. Also, because place p

may have multiple inputs, the *fusion of series transitions* rule may not be enabled, and that because transition t may have multiple outputs, the *fusion of series places* rule may not be enabled.

The additional requirements of this rule match the requirements of both *fusion of series* rules and for similar reasons.

Place p is reset by the same set of transitions as any output of transition t is

Transition t does not reset any place

20.4.1.7 Fusion of Equivalent Subnets (FES)

The last rule for reducing reset nets is a rule specifically tailored towards the reduction of alternative XOR-join-XOR-split tasks in YAWL nets. Recall that in a reset net a YAWL task is captured by a *busy* place, a set of *Start* transitions, and a set of *Complete* transitions. Now assume that we have several tasks that meet the following requirements:

- The inputs of all tasks are identical
- The outputs of all tasks are identical
- The cancelation regions of all tasks are identical
- All tasks have XOR-join and XOR-split decorators

then, we should be able to remove all-but-one of these tasks. Unfortunately, none of the rules presented so far is enabled on the resulting reset net fragment. Therefore, we include a specific rule for this construct. This rule *allows to remove* the subnets (or fragments) corresponding to all-but-one of the corresponding YAWL tasks. As this rule is directly aimed at reset nets, we present the additional requirements for reset arcs right away.

Given a set of *Busy* places B, a set of *input* places I, a set of *output* places O, a set of *Start* transitions S, and a set of *Complete* transitions C, and provided that:

- For every combination of an input place and a *Busy* place, there exists a *Start* transition such that the input place is its only input and the *Busy place* is its only output
- For every combination of a *Busy* place and an output place, there exists a *Complete* transition such that the *Busy* place is its only input and the output place is its only output
- All transitions are covered by the previous two items
- All transitions sharing the same input place as input (i.e., which correspond to the XOR-join of the same YAWL task) share the same set of reset places
- All transitions sharing the same output place as output (i.e., which correspond to the XOR-split of the same YAWL task) share the same set of reset places
- All busy places are reset by the same set of transitions,

then we can remove:

- All-but-one *Busy* places and
- All *Start* and *Complete* transitions not connected to the remaining *Busy* place.

20.4.2 Reduction Rules for YAWL Nets

On the basis of the reduction rules for reset nets, we can construct a number of reduction rules for YAWL nets, as long as no OR-joins are present. As we discussed the rules for reset nets in some length, we only briefly discuss the rules for YAWL nets. Figure 20.10 visualizes these rules:

- The *fusion of series conditions (FSC)* rule *allows to remove* a condition and a task, provided that the task does not reset and the place is reset by the same set of transitions as the output condition of the transition is.
- The *fusion of series tasks (FST)* rule *allows to remove* a condition and a task, provided that the task does not reset and the place is not being reset.
- The *fusion of parallel tasks (FPT)* rule *allows to remove* all-but-one parallel AND-join–AND-split tasks, while the *fusion of alternative tasks (FAT)* rule *allows to remove* alternative XOR-join–XOR-split tasks.
- The *fusion of parallel conditions (FPC)* rule *allows to remove* all-but-one parallel conditions, provided that all input tasks are XOR-splits and all input transitions are XOR-joins, while the *fusion of alternative conditions (FAC)* rule *allows to remove* all-but-one alternative conditions, provided that all input tasks are XOR-splits and all input transitions are XOR-joins.
- The *elimination of self-loop tasks (EST)* rule *allows to remove* an AND-join–AND-split task for which the set of input conditions is identical to the set of output conditions, provided that some other AND-join task contains at least the set of output conditions.
- The *fusion of AND-join–AND-split tasks (FAAT)* rule *allows to merge* an AND-split task and an AND-join task, provided that the output conditions of the former match the input conditions of the latter, while the *fusion of XOR-join-XOR-split tasks (FXXT)* rule *allows this* for an XOR-split task and a XOR-join task.

In some rules, some join-split decorators are kept, while in others the decorators are removed as well. The *fusion of series tasks* rule is an example of the former, while the *fusion of AND-join–AND-split task* is an example of the latter. The latter is visualized by filling the join-split constructs, while for the former these constructs are left as they were.

Also, some tasks have multiple inputs (outputs) while they do not have a join (split) decorator. This should be read as that for such a task the join (split) decorator is of no importance, it can either be an AND decorator, an XOR decorator, an OR decorator, or simply no decorator at all.

Apart from these rules, there are also two YAWL reduction rules that do take the OR-join into account. As one of these rules is a specialization of the other, we start with the more general rule: the *fusion of incoming edges to an OR-join (FOR1)* rule. This rule is quite similar to the *fusion of parallel conditions* rule and the *fusion of alternative conditions* rule, but

- Requires only one split task and one join task, while the other two rules allow for multiple splits and joins, and

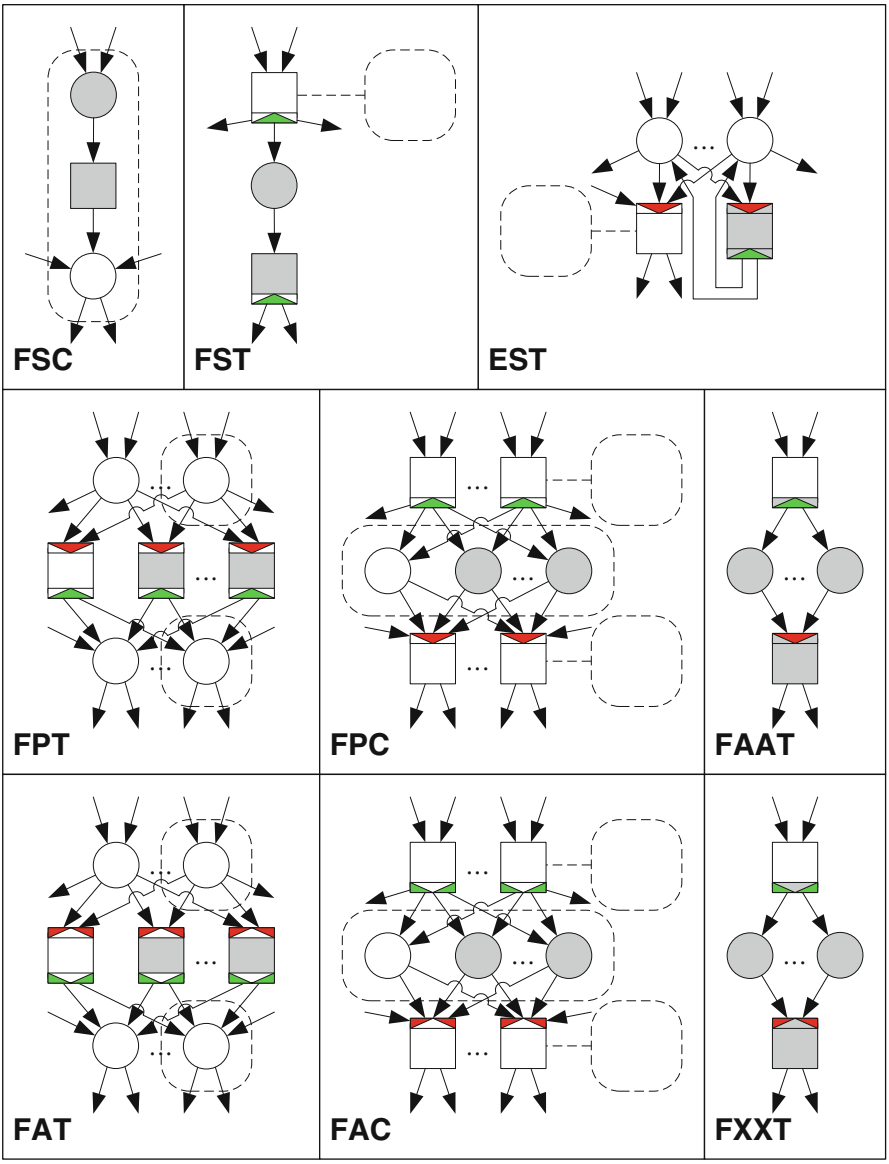

Fig. 20.10 A visual representation of all reduction rules for YAWL without OR-joins

- Allows for the join construct to be an OR-join, while the *parallel* rule requires an AND-join and the *alternative* rule an XOR-join.

Given a set of conditions, which contains at least two conditions, an OR-join task, and an arbitrary split task, this rule can remove all-but-one of the conditions. Provided that:

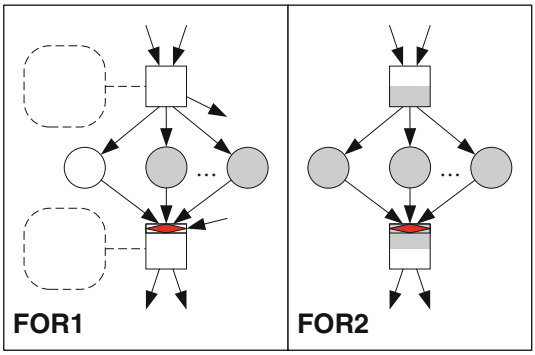

Fig. 20.11 A visual representation of both OR-join rules

- The conditions are being canceled by the same set of tasks
- The OR-join task is the output of every condition from the set
- The other task is the input of every condition from the set
- The OR-join task is not on some cycle with any other OR-join task,

then we can remove all-but-one of these conditions. The OR-join task is allowed to have additional inputs, while the other task is allowed to have additional outputs.

The OR-join is indifferent to the number of conditions in-between any preceding task and itself, as long as there is at least one. Therefore, we can simply remove all-but-one of them.

The second rule is a specialization of this rule, as it:

- Forbids the OR-join task to have additional inputs,
- Forbids the other task to have additional outputs,
- Forbids both tasks to have a (nonempty) cancelation region, and
- Forbids the OR-join to be on some cycle with some other OR-join task.

Using the *fusion of incoming edges to an OR-join (FOR2)* rule, we can now first remove all-but-one conditions in-between both tasks, which leaves only one input place for the OR-join. As a result, this OR-join decorator can be removed, and the *fusion of series tasks* rule can then be applied to reduce the condition and one of the tasks. However, because of the middle step (removing an OR-join decorator), this rule is not just a combination of both other rules. Figure 20.11 visualizes both rules.

After having applied all these soundness-preserving reduction rules over and over again, the resulting state space could be considerably smaller than the original state space. This could have a positive effect on the verification of the soundness properties, which all are based on this state space. Nevertheless, there is no guarantee that the resulting state-space is even finite in size, let alone be small enough to make the verification feasible. In such situations, we can use structural transition invariants that can be used to *approximate* the relaxed soundness property and diagnose the YAWL net based on this approximation.

20.5 Structural Invariant Properties

Recall that relaxed soundness uses the proper completing execution paths in a YAWL net, that is, those execution paths that start with a case in the input condition and end with the case in the output condition. As an example, take the YAWL net shown by Fig. 20.12, and ignore the dotted task *t* for now. Possible execution paths in this net include the following:

- *n c s*
- *n c e ce s*
- *nw cw w cw sw*

Now, if we would add the additional task *t* that transfers a case from the output condition to the input condition, then all these proper execution paths correspond to cyclic execution paths that include this additional task. As a result, all relaxed sound tasks would be covered by cyclic execution paths.

It is a known fact that in a Petri net any cyclic execution path corresponds to a semi-positive transition invariant. Basically, a transition invariant is a weighted sum of the transitions such that the current state would not change if all these transitions would be executed simultaneously according to their weight (a negative weight for some transition means that it is executed in the backward direction: it produces tokens for its input places and consumes tokens from its output places),

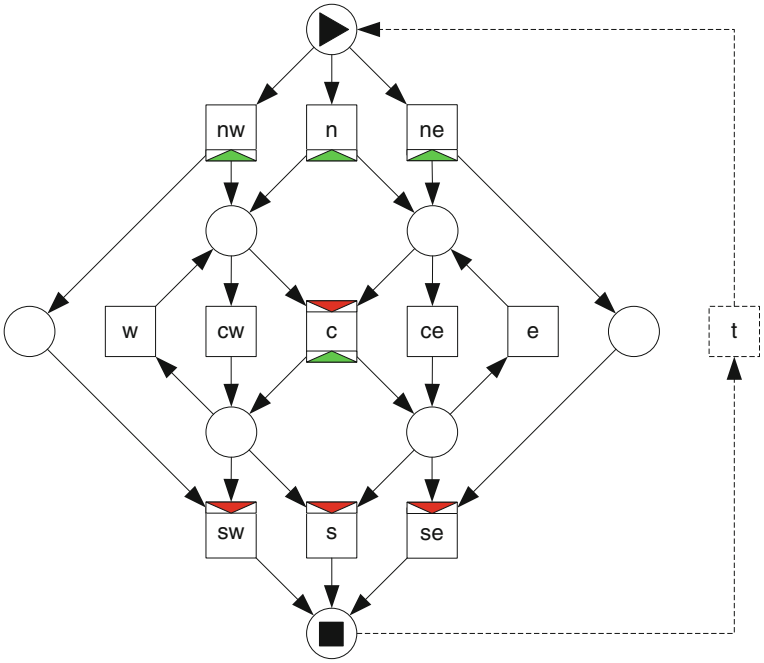

Fig. 20.12 The semi-positive transition invariant $c + e + nw + sw + t$ is not realizable

while a semi-positive transition invariant is a transition invariant where none of the weights are negative. As any cyclic execution paths lead back to the same state, the weighted set of executed transitions has to be a semi-positive transition invariant. In the example as shown by Fig. 20.12, possible invariants include the following:

- $c + n + s + t$
- $ce + e$
- $c + e + nw + sw + t$
- $c - ce - cw$

Note that only the first three invariants are semi-positive invariants, and that only the first and third include the additional task t.

So far, we have obtained the following:

1. Any relaxed sound transition is on some proper execution path.
2. Any proper execution path corresponds to a cyclic execution path after having added task t.
3. Any corresponding cyclic execution path corresponds to a semi-positive transition invariant that covers t.

Combining these three results, we obtain the fact that any relaxed sound transition is covered by some semi-positive transition invariant that covers t. Thus, if some transition is not covered by these invariants, then it cannot be relaxed sound.

Unfortunately, as Fig. 20.12 shows, this does not work the other way: examples exist for which some invariants do not correspond to some proper completion path. Take, for example, the third invariant mentioned above: $c+e+nw+sw+t$. Although this invariant satisfies the requirements (it is semi-positive and it covers t), it does not correspond to a proper execution path as the path blocks after having executed nw. The root cause of this is that not every invariant may be *realizable* in the model. For an invariant to be realizable, we need to be able to order the transitions in such a way that the transitions could be executed by the model in that order. For some invariants, this may not be possible. As a result, we may not be able to locate all relaxed transitions by this technique, as some not-relaxed-sound transitions may be covered by invariants that are not realizable.

Transition invariants are structural properties, which means that they can be decided using only the structure (and not the behavior, i.e., the state space) of the net. Thus, if constructing the state space is no option, we could still construct transition invariants and use these invariants for diagnostic purposes.

20.6 Tools

The techniques mentioned in this chapter have been implemented in two tools: The YAWL Editor and WofYAWL. While the Editor is part of the entire YAWL suite, the WofYAWL tool is a stand-alone tool that works on YAWL Engine files.

The Editor contains algorithms to construct a state space, if possible. To help in constructing a state space, it can reduce the YAWL net as much as possible while preserving the soundness property. As mentioned before, if a YAWL net contains OR-joins, the YAWL semantics are used to construct a state space. Otherwise, a YAWL net is converted to an RWF-net and the reset net semantics are used to construct the state space. Given that a state space can be constructed, the Editor can decide soundness and decide which OR-joins can be simplified. If the construction of the state space is not feasible, the Editor can still decide weak soundness for YAWL nets without OR-joins and for some YAWL nets with OR-joins and decide which tasks and conditions could be removed from certain cancelation regions.

Figure 20.13 shows the verification result on the unsound *Carrier Appointment* decomposition described earlier. The first warning states that this example may complete while the task *Prepare Transportation Quote* and its immediately preceding implicit condition may still be enabled. For sake of readability, we include the first four messages on this decomposition here:

1. ResetNet Analysis Warning: Tokens could be left in the following condition(s) when the net has completed: [[Atomic Task:Prepare_Transportation_Quote_4, Condition:c{null_41_Prepare_Transportation_Quote_4}]]
2. ResetNet Analysis Warning: The net Carrier_Appointment does not satisfy the weak soundness property.
3. ResetNet Analysis Warning: The net Carrier_Appointment does not have proper completion. A marking 2OutputCondition_2 larger than the final marking is reachable.
4. ResetNet Analysis Warning: The net Carrier_Appointment does not satisfy the soundness property.

As a result (see also the fourth warning), this model is not sound.

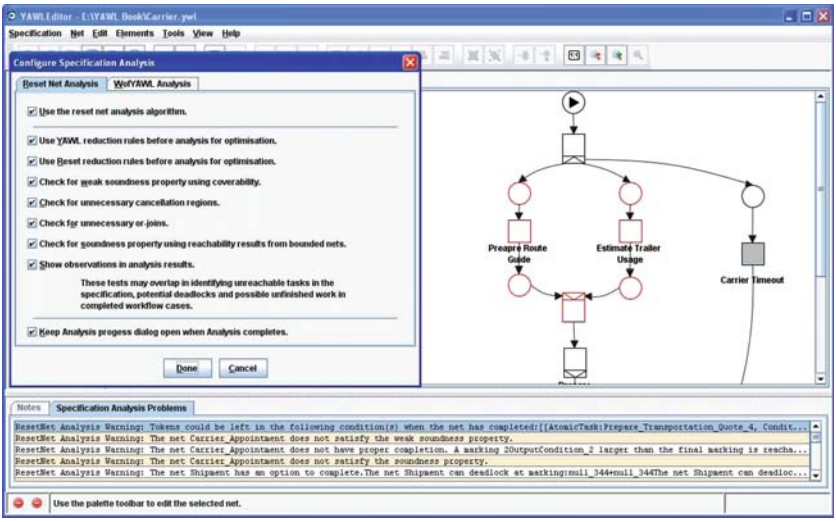

Fig. 20.13 The Carrier Appointment example in the YAWL editor

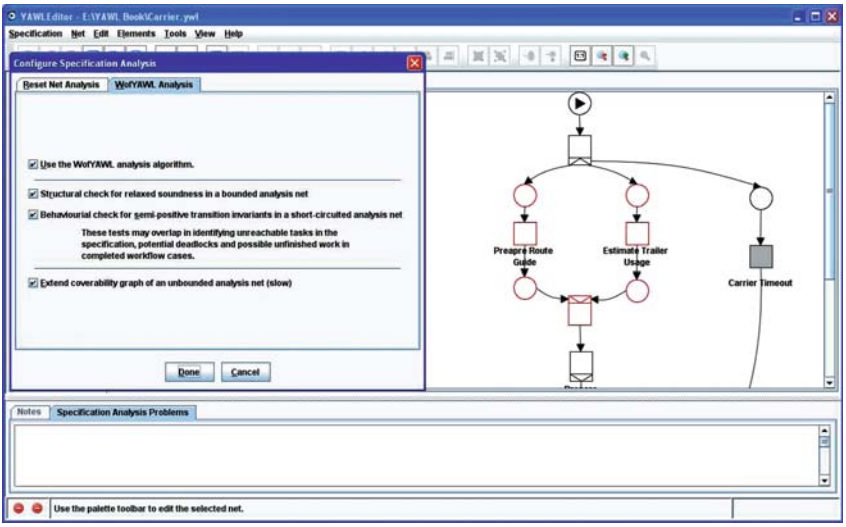

Fig. 20.14 The Carrier Appointment example in the YAWL editor, using WofYAWL

WofYAWL is a spin-off of the workflow verification tool Woflan that is especially tailored towards verification of YAWL nets. Based on a YAWL Engine file, it can also try to construct the entire state space, but, unlike the Editor, it uses only the reset net-based reduction rules when reducing the model before constructing the necessary state space. Given a state space, WofYAWL can decide relaxed soundness or use transition invariants to approximate relaxed soundness.

The Editor and WofYAWL have been loosely integrated in the sense that the Editor can start WofYAWL on the YAWL net at hand and can display WofYAWL's messages in the Editor pane. Figure 20.14 shows the results of WofYAWL's diagnosis of the unsound *Carrier Appointment* decomposition. As this decomposition is relaxed sound, WofYAWL reports that no problems were discovered.

20.7 Concluding Remarks

This chapter has introduced a number of desirable properties for YAWL nets, which include the following:

- *Soundness:* Can every case complete in a proper way and is every task viable?
- *Weak soundness:* Can a new case complete, is completion always proper, and is every task viable?
- *Relaxed soundness:* Can any task contribute to the proper completion of a case?

The weak soundness property is decidable, but the soundness property and the relaxed soundness property are not. This means that for certain example of YAWL

nets, a definitive answer on whether the YAWL net is (relaxed) sound can not be given. Nevertheless, many YAWL nets may exist for which this answer can be given.

To help in the computation of these properties, two sets of reduction rules have been introduced. By first applying these reduction rules, the YAWL net becomes smaller, which might alleviate the problem of computing the property itself. The first set of reduction rules operates on the reset net that underlies a YAWL net, while the second set operates on the YAWL net itself. As the OR-join construct cannot be captured successfully by reset nets, the first set of rules can be used only in the absence of this construct.

In case the computation of the properties is too hard to compute even after the reductions have been applied, this chapter has shown how the relaxed soundness property can be approximated using only structural properties of the underlying reset net. These properties are structural as they do not require the current state, and hence do not require the entire state space to be constructed.

The YAWL Editor has been extended to enable a YAWL process designer to check the above mentioned properties for a YAWL net at hand. For the non-structural properties, the Editor provides native support, while it relies on the WofYAWL tool for the structural properties. Based on these techniques, the Editor can also check whether:

- Some OR-joins can be simplified to either an XOR-join or an AND-join
- Some tasks or conditions can be removed from some cancelation region

These advanced verification techniques help the developer of a YAWL net to correct possible errors in the YAWL net at hand, which is shown nicely by the *Carrier Appointment* decomposition. The original *Carrier Appointment* decomposition was developed by two YAWL experts. Nevertheless, it contained a flaw that allowed for improper completion, which shows the value of verification support for all developers.

Exercises

Exercise 1. Determine whether the *Carrier Appointment* decomposition (shown in Fig. 20.2) is sound, relaxed sound, and/or weak sound.

Exercise 2. Consider the OR-join construct shown by Fig. 20.5. Assume that the input condition directly precedes the anonymous AND-split task, and that the output condition directly succeeds the *Create Bill of Lading* task. Construct a state space (use the YAWL OR-join semantics) for this YAWL net, and determine whether it is sound, relaxed sound, and/or weak sound. Also determine whether the OR-join can be simplified to either an AND-join or a XOR-join.

Exercise 3. As Fig. 20.3 shows, the *fusion of series places* cannot reduce the *Busy Join* place and the *Complete Join* transition. Determine why not.

Exercise 4 (Advanced). The three soundness properties all abstract from the ordering between tasks. Show that the second option of the *fusion of series transitions* rule (where inputs and outputs are transferred from t to u) does not necessarily preserve the ordering between tasks.

Exercise 5. Determine which places and transition could be reduced from the net of Fig. 20.3 if it would only use the *fusion of series transitions* rule, and to what extent this differs from the result as shown by Fig. 20.8.

Exercise 6. Show that the rules as shown by Fig. 20.10 are correct using the rules as shown by Fig. 20.7. To do so, first convert a YAWL fragment onto a reset net fragment, apply reduction rules on the reset net level, and convert the resulting reset net fragment back to a YAWL net fragment.

Exercise 7 (Advanced). Show how transition invariants can be computed in the presence of reset arcs, which, by nature, disturb these invariants. Hint: transition invariants are trivial to compute if every reset arc could be replaced by an *inhibitor* arc. An inhibitor arc from a place to a transition prevents execution of the transition as long as the place is marked.

Chapter Notes

Petri Nets

In his overview paper on Petri nets, Murata [179] has provided a number of reduction rules that preserve both liveness and boundedness. In their book on free-choice Petri nets, Desel and Esparza [77] have given three different reduction rules that also preserve liveness and boundedness. In contrast with the rules provided by Murata, the Desel and Esparza rules are complete: any live and bounded nets can be reduced to a simple net containing only one place, one transition, and two arcs. However, the decision whether some rule is applicable is easy for the Murata rules, while it is complex for two of the three Desel and Esparza rules.

Workflow Nets

Workflow nets have been introduced by van der Aalst [2] as a subclass of Petri nets to capture the essential behavior of workflow processes.

Soundness

In the same paper, van der Aalst [2] has introduced the soundness property on workflow nets, which is considered as the minimal requirement any workflow process

should satisfy. Based on this soundness property, the Woflan tool has been build by Verbeek and others [250]. The WofYAWL [249] tool is a spin-off of the Woflan tool. As this soundness property abstracts from data, which might influence the flow of control, the notion of relaxed soundness has been introduced by Dehnert [75]. A workflow net that is sound is also relaxed sound, but not vice versa. Both soundness and relaxed soundness are not decidable in general when taking reset nets into account, as reachability is undecidable [12, 13, 80, 272]. Therefore, Wynn [272, 274] has introduced the notion of weak soundness,[2] which is decidable even for reset nets.

Reset Nets

More information on reset nets can be found in [72, 80, 81, 95, 96]. The fact that coverability is decidable for reset nets has been shown by Dufourd and others [80], which was used by Wynn [272, 274] to show that weak soundness is decidable for reset nets. The soundness preserving reduction rules have been introduced by Wynn and others [251, 272, 277] and implemented in the YAWL Editor.

YAWL Nets

The OR-join semantics of YAWL nets has been discussed by Wynn and others [273]. The simplistic view on the OR-join semantics which is allowed by the relaxed soundness property has been introduced by Verbeek and others [249]. Finally, the set of YAWL net reduction rules has been introduced by Wynn and others [272, 276].

[2] The term "weak soundness" used here has a different meaning to the one introduced by Martens [158].