Analysis in YAWL

Zhong Xi Lu

Department of Mathematics
and Computer Science
University of Antwerp, Belgium
zhong-xi.lu@student.uantwerpen.be

Abstract

It's no surprise that nowadays, we want to model as much as possible whereas modelling can help us in various aspects. YAWL in particular models the workflow of a process from start to end. However, after designing such a flow, we also want to be able to derive certain properties from a given model. This paper will go deeper on this analysis step and how it's done in YAWL.

Keywords: yawl, workflow, petri-nets, workflow verification

1. Introduction

YAWL or *Yet Another Workflow Language* is a language to describe **workflows**. It's often used in a business management environment, but can in theory be used for almost every process, like a cashier system where a customer wants to pay for his/her goods. This is then described through a workflow with different tasks. Furthermore, YAWL also allows the user to do some kind of analysis, for example if deadlocks may occur or if the process will ever end.

This paper will give an overview of how we can deduct these varies properties, such a deadlock, liveness, soundness, ... In section 2, a brief introduction about the language is given, i.e. the syntax with its semantics. Note that this paper assumes that the reader already has basic knowledge about Petri-Nets as YAWL is based on them. In section 3, a method will be discussed on how we can find these properties and what properties are to be desired. Finally, in section 4, a small conclusion is given.

2. YAWL

2.1. Basic syntax and semantics

As earlier mentioned, YAWL is mostly based on Petri-Nets or it basically extends it with many advanced concepts. Therefor, there are two main components: **places** (or conditions) and **transitions** (or usually just called tasks or activities). These components can then be connected with an edge; places can only be connected to other transitions. A small difference with Petri-Nets is that tasks can directly linked to other ones, this essentially means that when a tasks finished, the next one is instantly started. In figure 1 is a basic workflow shown; we have three tasks, namely A, B and C. Notice that we have a startpoint and an endpoint (indicated with respectively a green triangle and red square inside a place). The whole process starts by either executing task A and B which is then followed up by the final task C. After this, the whole main process ends.

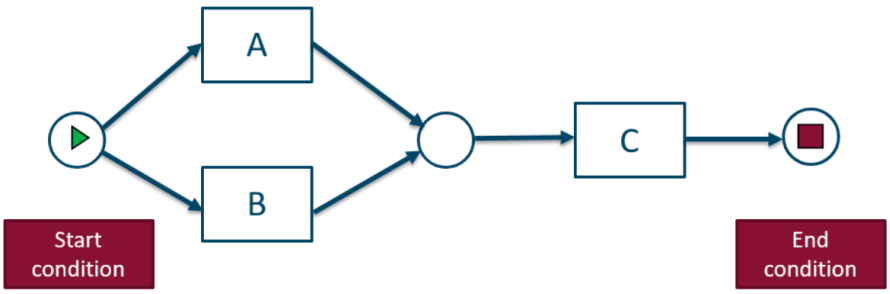

Figure 1: Basic workflow

For the semantics part, it's mostly the same as in Petri-Nets, a transition can be fired if all the input places have a marking value equal to the weight of their edge. In other words, we can start a task when all its dependencies are all finished.

2.2. Advanced concepts

The central component in YAWL are the tasks, YAWL offers a few representations for these task:

- **Atomic** task: a task that needs to be executed only once.
- **Composite** task: similar to the atomic task, but consists of more subtasks. This way we can isolate the details of such a task, creating modularity/hierarchy in a way.
- Lastly, we can represent a task by **multiple instances**, i.e. we can run the tasks multiple times (concurrently). This can be both for an atomic and a composite task.

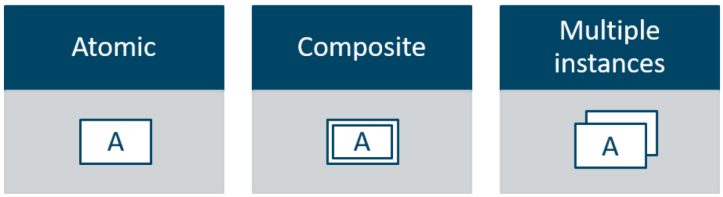

Figure 2: The different representations of a task

Another difference with Petri-Nets is that these tasks usually have some kind of delay, meaning that they need time to finish as in the real world. Moreover, YAWL also provides some "syntactic sugar" when it comes down to splits and joins:

- **AND-split**s: when we want to create working flows that go in parallel.
- **AND-join**s: when we have multiple dependencies of a task which all need to be finished. This implicitly also synchronizes the workflow.
- **XOR-split**s: when we only want to choose exactly one workflow from the outgoing edges.

- **XOR-join**s: when only one of the dependencies has finished, we can instantly start the task.
- **OR-split**s: here we can choose any number greater than 0 outgoing workflows.
- **OR-join**s: this one is a bit tricky, the task waits until all the previous tasks have finished or it knows that some tasks cannot finish. Basically, the dependencies all have the chance to finish their task, but if they cannot, the main task will just carry on. Otherwise, it waits for every task to complete. As a side note, OR-joins are to be avoided [1] since they will introduce complexity as it will be explained later in this paper.

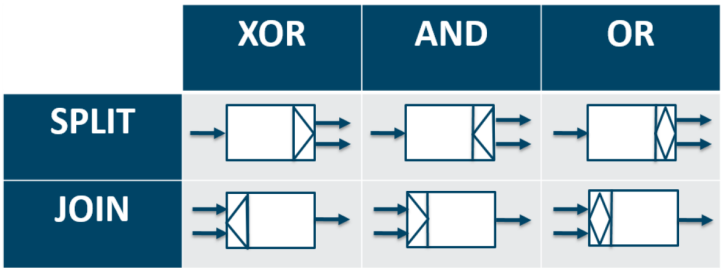

Figure 3: The concrete syntax of splits/joins combined with XOR, AND, OR

Finally, there are of course still many other concepts to describe, such as **cancellation regions** for when we want to cancel a task while it's running, **timers** to specify timeouts on tasks, **data storage** to store data across multiple tasks, ..., but that's outside the scope of this paper.

3. Analysis

Having built a valid workflow, we also want to be able to do some kind of analysis to know in advance if a "dangerous" condition might occur so we can adapt our initial model. For example, we may want to know how long such a process takes or if there's a possibility that a deadlock (or starvation) might happen.

3.1. Verification

In most cases, we can just directly translate our YAWL model to a **petri-net** through ways of model transformations. On this newly created petri-net model, we can then perform some analysis to find more about deadlock, liveness, invariants, reachability, ... The good thing about these petri-nets is that there is already a good foundation of tools that are supported to analyze a model, therefore it is not needed to re-implement those tools again in the YAWL editor.

However, there are a few edge cases we must keep in mind. To be more specific, cancellation regions and OR-joins form somewhat a problem when doing this analysis. For this, we cannot just simply map a workflow onto petri-nets. If the workflow contains some cancellation regions, but no OR-joins, we can map it to **reset-nets** and do the same analysis as before. Reset-nets extend the basic petri-nets with reset arcs; simply put, when those arcs are traversed, the tokens from the place it came from are all reset (marking set to 0). This is incredibly useful since we deal with "resets" when we have to cancel a task.

As explained before, OR-joins behave kind of "smart" in a sense that they have to know if a task will finish eventually. Normal joins and splits have some locality, they can only look at their inputs and outputs, but this OR-join does need some information about the other tasks. Again, we map a workflow to a reset-net, but this time we do some small modifications a priori to deal with this. Basically, we can transform all the OR-joins to XOR-joins as it was proposed in this paper [2] about how to verify a YAWL net.

3.2. Properties

Obviously, since we can map our YAWL nets onto petri-nets, we can easily deduct properties as deadlocks with the already existing tools. Either way, in this section a small set of properties are given which are desirable to have for a workflow.

3.2.1. Soundness

For a net to be sound, it needs to satisfy three conditions:

1. **Option to complete**: the process needs to be able to finish in all possible cases. This implicitly means that there cannot be a deadlock in any scenario.
2. **Proper completion**: when the process finishes, all other tasks must already have ended. In other words, there cannot still be a task running while the whole process ends.
3. **No dead transitions**: we don't allow any dead transitions in our net. A transition is dead if the transition can never fire in all possible cases.

Finally, there was also a weaker version of this property proposed [2], namely **weak soundness**. In this version the first condition is weakened; the task needs to be able to finish in some cases instead of all cases.

3.2.2. Immutable OR-joins

As OR-joins introduce some complexity, we might want to reduce the amount of OR-joins as much as we can. Therefore, if possible, we want to convert an OR-join into XOR-join if there can never be more than one input condition marked and convert it to an AND-join if all the input conditions are always marked in all cases. This property holds, if we cannot find any OR-join that can be converted following these principles.

3.2.3. Irreducible cancellation regions

Cancellation regions allows us to define regions where tasks can be canceled, even if they're still being executed. However, one must be careful choosing this region. When an element cannot be canceled while the task is running, then this element is called **reducible**. If all the elements in every region are irreducible, then the net satisfies the irreducible cancellation regions property.

4. Conclusion

As a conclusion, YAWL can be a very strong language when it's put to good use. It not only aids in modelling a precise workflow with lots of different

constructs available, but there's also a good amount of verification tools that will further help the user understand the workflow. Ultimately with this verification, the user can try to create the optimal flow, whether that be the fastest way to finish a process, minimizing the resources, ..., it can all be done. We can even opt to transform this YAWL model to another model, such as petri-nets for more analysis. Either way, YAWL provides a good basis as a language and is especially useful in the context of business/development processes.

References

[1] L. Bradford, M. Dumas, Getting Started with YAWL (May 2007).

[2] M. T. Wynn, W. M. P. van der Aalst, A. H. M. ter Hofstede, D. Edmond, Verifying Workflows with Cancellation Regions and OR-joins: An Approach Based on Reset Nets and Reachability Analysis.

[3] W. M. P. van der Aalst, A. H. M. ter Hofstede, YAWL: Yet Another Workflow Language (Revised version).

[4] H. M. W. Verbeek, M. T. Wynn, W. M. P. van der Aalst, A. H. M. ter Hofstede, Reduction Rules for Reset/Inhibitor Nets (May 2009).

[5] Verification in yawl, `http://www.yawlfoundation.org/pages/research/verification.html`.