Modelling and Analyzing YAWL in AToMPM

Zhong Xi Lu

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

Abstract

YAWL or *Yet Another Workflow Language* [1] is a language to model different kind of workflows, but what lies behind that is basically petri-nets. Therefore, if we want to do some analysis on the YAWL net we can simply map them to a petri-net. This paper will go deeper on how we can first model this formalism in the tool AToMPM (*A Tool for Multi-Paradigm Modelling* [2]) and then check the model for certain properties.

Keywords: yawl, petri-net, deadlock, soundness

1. Introduction

This paper will describe in detail how we can create a modelling environment for the language YAWL or at least a small subset of it. To be specific, the tool AToMPM will be used to model the language and on top of that, also the different kind of model transformations that will be needed to map them to petri-nets. This mapping allows us to directly perform an analysis on the net and retrieve the results, which we can somehow indicate on our initial model, so the user is able to modify the model to maybe avoid some potential problems. The tool that will be used for the analysis is LoLA (*a Low Level Petri net Analyzer* [3]), but a more detailed description of this tool is later given.

First, we will explicitly model both the abstract and concrete syntax in section 2. Next, we will define a mapping from a YAWL net to a petri net with transformation rules in section 3. After we have successfully created an

equivalent petri-net, we can easily simulate the model (section 4) and of course, we can run an analysis to find some properties (section 5). Finally, a brief conclusion is given in the last section 6. Some examples are also discussed in the appendix (section 7) for more clarification.

2. Abstract and Concrete Syntax

Of course, when we want to model a language, we need to precisely define its syntax. AToMPM allows us to create meta-models which then can be instantiated to create a model, which the user eventually wants.

2.1. Abstract Syntax

The abstract syntax will describe all the different entities together with their attributes that can be instantiated in the formalism. It will also specify all the possible relations between the entities. Note that it's also possible to define constraints.

Specifically, in YAWL, there are two main components represent, namely tasks and conditions. Therefore we have two "base" entities `Task` and `Condition` that respectively represent an atomic task and a condition. Conditions also have a property `tokens` that represents the amount of tokens it has. But naturally, there are also some other concepts available, like split and joins. These concepts are explicitly modelled here, i.e. they have an entity themselves. As a consequence, it is not possible to combine splits and joins in this specific case. As for the conditions, the start and end condition are also explicitly modelled to make things easier for later.

For the relations, there are three types of relations; `ConditionToTask`, `TaskToCondition` and `TaskToTask`. These relations allows the user to create a "link" between entities, e.g. the `ConditionToTask` relation allows linking a condition to a task in this specific direction. For simplicity reasons, an atomic task can only be connected to and from one task. On the other hand, a split can be connected from multiple tasks and join to multiple, this is done by overriding the multiplicity constraint of the `TaskToTask` relation

To clarify, since only a subset of YAWL is being modelled, advanced concepts, such as cancellation regions, timers, data storage, ... are not modelled here.

The complete abstract syntax can be seen in figure 1.

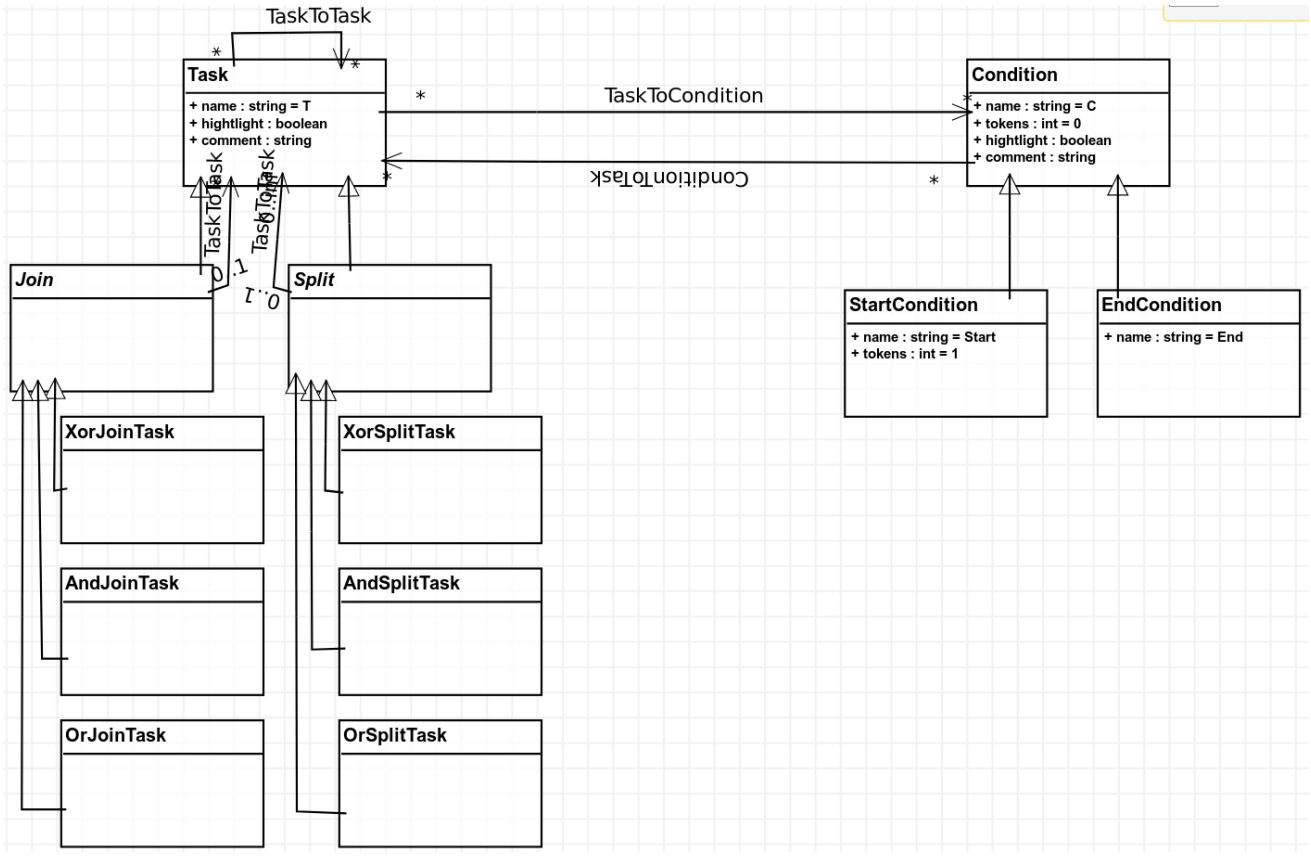

Figure 1: Abstract syntax of YAWL in AToMPM

2.2. Concrete Syntax

Next to the abstract syntax, it is also possible to define the concrete syntax. This latter syntax defines how all the modelled entities are visually rendered. Here also, we can define visual constraints.

All the non-abstract entities (that were defined in the abstract syntax) need a visual representation. The notations that are used are based on the official

notations of YAWL [4]. As said before, the splits and joins are explicitly modelled, meaning they also explicitly need some representation. Note that it might be possible to add some kind of enumeration and dynamically update the representation of a task. This then also allows us to combine split and joins, but again for simplicity reasons, this was not done.

The complete concrete syntax can be seen in figure 2.

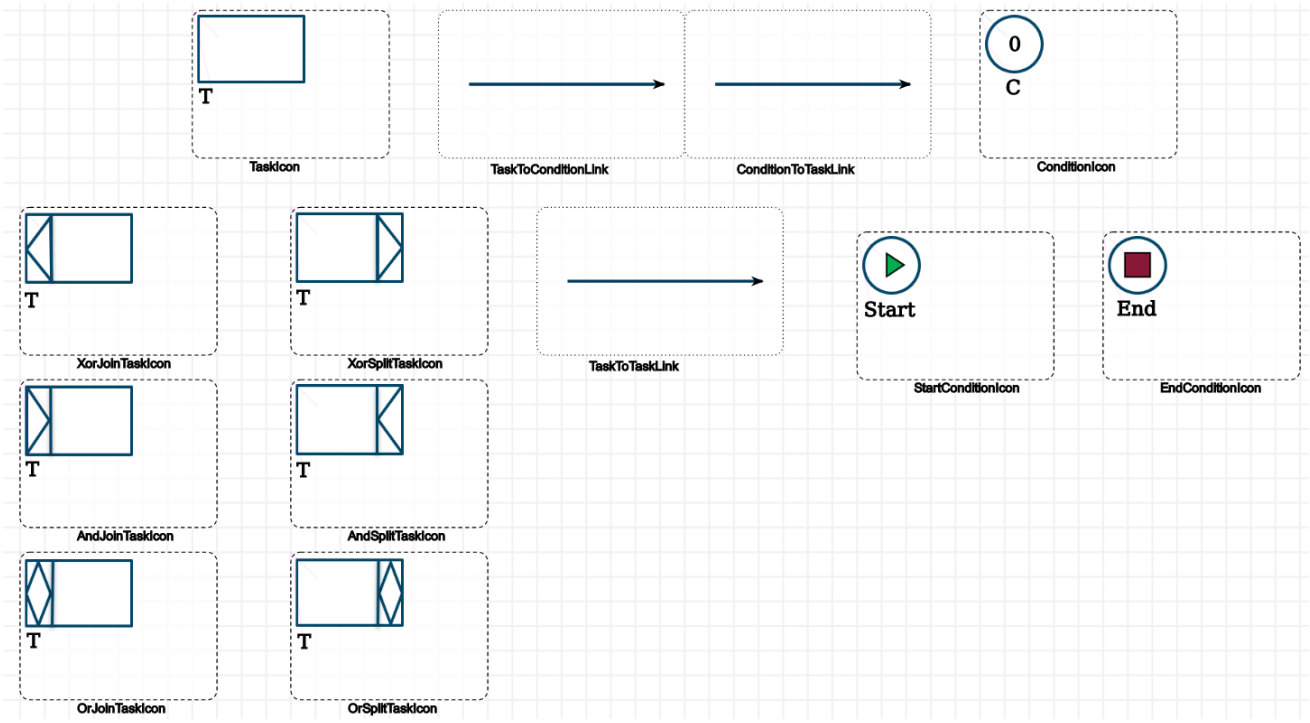

Figure 2: Concrete syntax of YAWL in AToMPM

3. Mapping to Petri-nets

Since YAWL is a language that is based on petri-nets, it is pretty straightforward we should map them to a petri-net. Luckily, those two formalisms are not so far apart in terms of semantics; tasks are transitions in petri-nets and conditions are places. The behaviour is also mostly similar, only that YAWL has

some extra concepts that need to be explicitly transformed to a petri-net-like structure.

The way this mapping is achieved, is through model transformations. In AToMPM, we can easily define transformation rules that will incrementally build a model in the target formalism. A more detailed explanation of these rules can be found in the documentation of AToMPM [2].

As earlier mentioned, we can directly map (atomic) tasks to transitions and conditions to places. Note that for the start condition, we want to create a place with a marking value of 1 and for the end condition we want to make it that it cannot end up in a dead state. This is simply done by adding an extra transition that will consume and produce from the end place (see figure 3), so it potentially runs forever if it reaches the end.

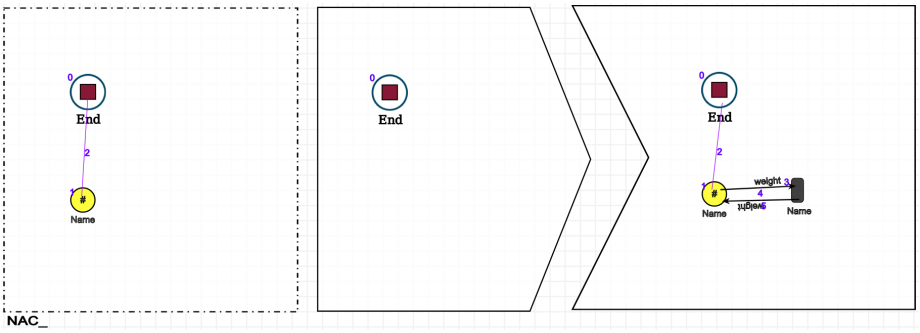

Figure 3: Transformation rule for an end condition

The different links can also easily be mapped to links from the petri-net formalism. The `TaskToTask` link, however, does need an intermediate transformation that will put a basic condition in between.

The non-trivial part of the whole mapping lies in these splits and joins, which need to be somehow translated to petri-net structures. Again, OR-joins are not implemented here due to their *complexity* [5],

YAWL can also be compared to *Business Process Model and Notation* (BPMN) [6]. As such, those mappings can be based on some of the already-existing trans-

formations [7] from BPMN to petri-nets. To be precise, we can always translate a split or join to a petri-net-like structure. These translations are shown for AND/OR joins/splits in figures 4, 5, 6, 7 and 8. Note that these are not the final rules implemented in AToMPM, these are just modelled in a rule for illustration purposes. The actual transformation consist of two steps that will create this complete transformation.

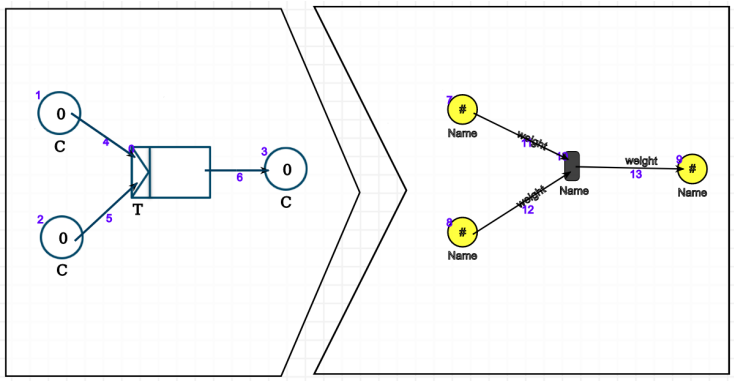

Figure 4: Mapping AND-join to equivalent petri-net structure

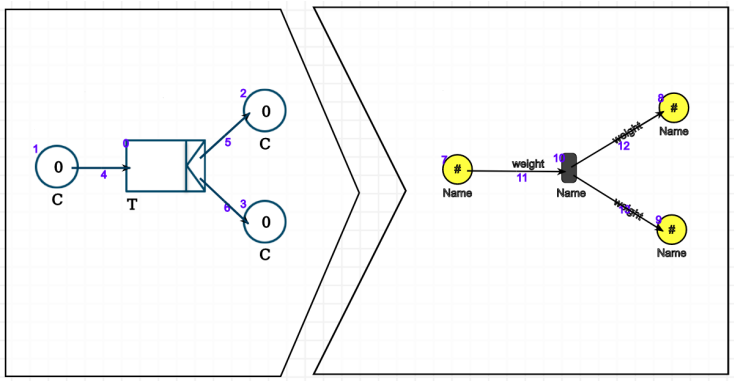

Figure 5: Mapping AND-split to equivalent petri-net structure

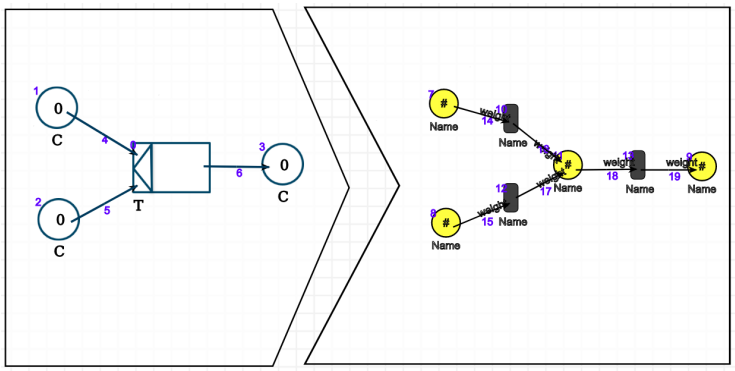

Figure 6: Mapping XOR-join to equivalent petri-net structure

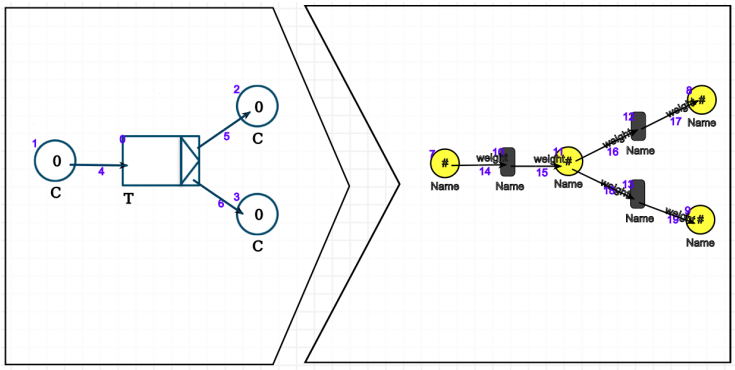

Figure 7: Mapping XOR-split to equivalent petri-net structure

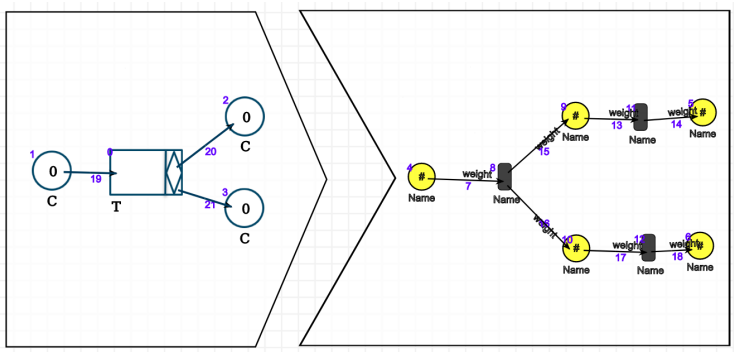

Figure 8: Mapping OR-split to equivalent petri-net structure

4. Simulation

Now that we have a valid mapping to a petri-net, we can easily use this petri-net to also simulate our workflow with its semantics. This way, we maintain the behaviour as well and we get the same results. To be precise, we map our YAWL net onto a petri-net and then call the simulator for the petri-net. The schedule (in MoTif) for the simulation can be found in figure 9. In other words, we define the semantics of YAWL by mapping it to petri-nets.

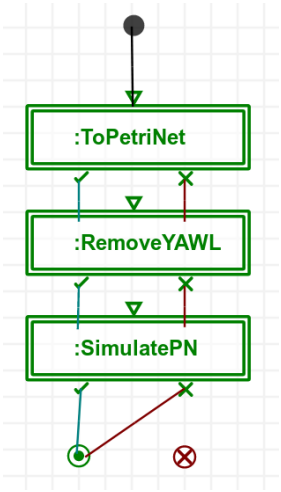

Figure 9: The MoTif schedule for the simulation

5. Analysis

Not only do we want to simulate our model, but most likely we want to verify our model, i.e. whether it satisfies certain properties. For this specific case, two main properties are implemented that are checked on the model, namely deadlock and soundness.

5.1. Mapping to petri-net

First, we of course map the YAWL model onto a petri-net. Then an intermediate step is taken; we create a `.net` file [3] based on our newly created petri-net. This is done through a transformation rule, but it will not modify

the model, it will simply have some action code on the RHS of the rule. This code will then generate this `.net` file which will be the input for the following LoLA executions.

5.2. Model Checking

The reason LoLA is used, is because it is a low level petri-net analyzer and it can be called via commands, which is what we ultimately want. On top of that, it allows the user to specify custom "formulas" that need to be true when checking the model. These formulas are written in *Computation Tree Logic* (CTL), more details on the notation can be partially found here [3].

5.2.1. Deadlock

The definition of deadlock (generated by `info lola`) is given by:

> *"A 'deadlock marking' is a marking where no transition is enabled."*

Basically, we want to check if our model can reach a certain deadlock state. If it can, then we most likely want to change our initial model, otherwise the whole process might never end when it reaches this deadlock state. In LoLA this can be easily checked by simply running a command on our petri-net and check whether the test concluded in a true or false. The formula that is used is as following: `EF DEADLOCK` and means that there exists (`E`) a deadlock state that is finally (`F`) or eventually reachable from the initial marking. It will true if a deadlock state can be reached.

5.2.2. Soundness

Another very strong property is soundness and its definition is given by (generated by `info lola`):

> *"A workflow net is 'sound' if the final marking is reachable from all reachable markings, and the net has no dead transitions. This amounts to liveness of a state predicate that describes the final marking [...] Again, we recommend to split the soundness check into many individual runs of LoLA."*

To verify soundness, we can split it in two formulas; one that checks that there are no dead transitions and one that will verify it the "final" marking can be reached from the initial state. This final marking will be the state where there's one token in the end condition (option to complete) and zero in all the other conditions (proper completion).

Checking if there are no dead transitions can be done by individually checking for each transition whether that transition is dead or not. The definition for "dead" is given by (generated by `info lola`):

> *"A transition is 'dead' in a given marking m if it is not enabled in any marking reachable from m."*

What we're essentially checking is if our petri-net satisfies the quasi-liveness property, i.e. if none of the transitions are dead. The CTL formula to check if a single transition is dead, is: `AG NOT FIREABLE(T)` with `T` the transition. This essentially means that for all (`A`) the possible paths from the initial marking `T` can never be fired globally (`G`). This formula is basically the negation of this property, i.e. it will return true if `T` is a dead transition. We then apply this formula for each transition and if all the tests return false, then the petri-net has no dead transitions.

For a net to be sound, it also needs the option to complete and have a proper completion, meaning that it is possible to reach the end condition and when it does, there cannot still be another task running. As described earlier, this comes down to checking if the final marking is reachable from the initial marking. In this final marking all the conditions have a marking value of 0 which represents that all the tasks have been finished, except for the end condition, which will have a marking value of 1, that will indicate that the whole process has come to an end. An example of such a state can be $p_1 = 0 \wedge p_2 = 0 \wedge ... \wedge p_n = 0 \wedge p_{end} = 1$ where p_i represents a place in the petri-net and p_{end} the end condition. The CTL formula is then given by: `AGEF(final_marking)` where `final_marking` corresponds to the final marking. What this formula will do is it will check for all paths (`A`) globally (`G`) if there exists (`E`) one path that will finally (`F`) lead

to this final marking. Here, it will also return true if it satisfies the option to complete and proper completion property.

5.3. Updating model

Having run these verification tests, we can then import the results and indicate somehow on our model these details, e.g. showing the path (sequence of tasks executed) that led to a deadlock state. Once more, we update our model through a model transformation, so that we can change the appearance of an icon or maybe add a comment to describe the details. In the appendix (section 7) a couple examples are shown on how the model is updated when a "dangerous condition" occurred.

5.4. Schedule

The full MoTif schedule for the verification can be found in figure 10. It will take a YAWL net as input, create an equivalent petri-net (with the same semantics), analyze this petri-net on certain properties, remove the petri-net (since it was only needed to do the analysis) and finally update the initial model, so the user can easily see the results.

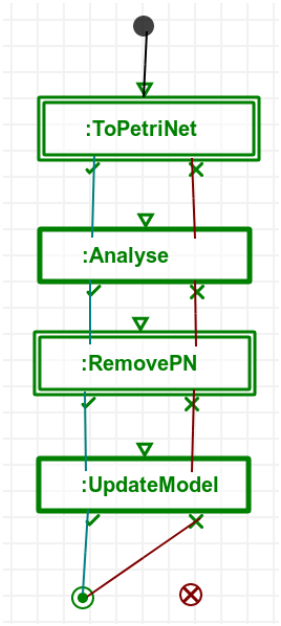

Figure 10: The MoTif schedule for the analyzer

6. Conclusion

To conclude, we've seen all the different aspects for modelling YAWL in AToMPM, from defining the meta-models to verifying it. In most cases, we needed to map our YAWL net to a petri-net since those behaviours are very much alike. In this specific implementation, the model is checked for deadlocks and for soundness, but this can of course be extended with more properties and easily be formulated with LoLA, allowing for a better verification.

References

[1] Yawl, `http://www.yawlfoundation.org/`.

[2] Atompm documentation, `https://msdl.uantwerpen.be/documentation/AToMPM/index.html`.

[3] K. Schmidt, LoLA: a Low Level Petri net Analyzer (September 2000).

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

[5] 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.

[6] Business process model and notation, `http://www.bpmn.org/`.

[7] N. Lohmann, E. Verbeek, R. Dijkman, Petri Net Transformations for Business Processes - A Survey (2009).

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

[9] 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).

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

[11] D. Fahland, C. Favre, J. Koehler, N. Lohmann, H. Völzer, K. Wolf, Analysis on Demand: Instantaneous Soundness Checking of Industrial Business Process Models (March 2011).

7. Appendix: Examples

Example of deadlock

In figure 11 a model is shown where a deadlock will occur. After task `T1` has been finished, it will only select exactly one of the outgoing tasks (XOR-split), namely `T2` or `T3`, but since `T4` is an AND-join, it requires both `T2` and `T3` to finish, which will never happen and lead to a deadlock state. The intermediate petri-net that is needed for the analysis is pictured in figure 12 and the final results in figure 13. Note that the sequence of executed tasks that led to the deadlock is also given by the #.

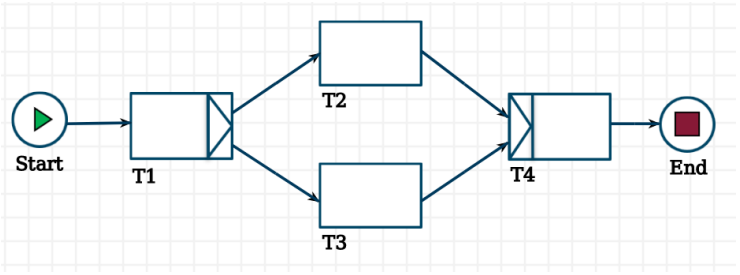

Figure 11: YAWL model where a deadlock will occur

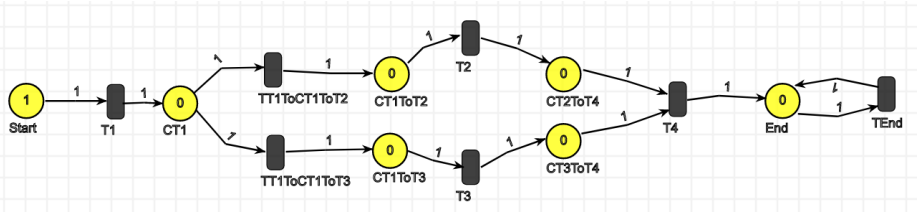

Figure 12: The petri-net corresponding to the model in figure 11

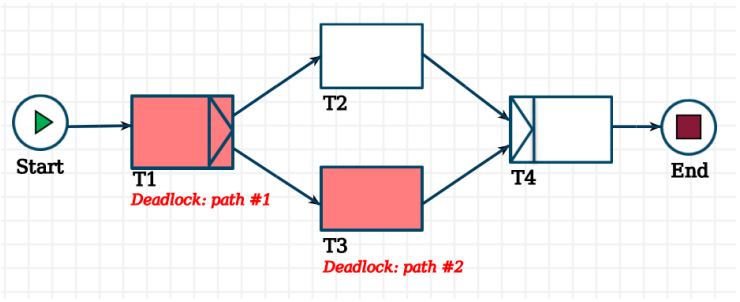

Figure 13: The results after analyzing the model in figure 11

Example of dead transition

An example of a model with a dead transition can be found in figure 14. It is pretty clear that `T2` will never fire, since its precondition `C1` can never have a token. Running the analysis gives us the results in figure 15.

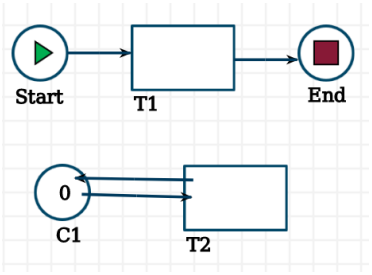

Figure 14: YAWL model that contains a dead transition

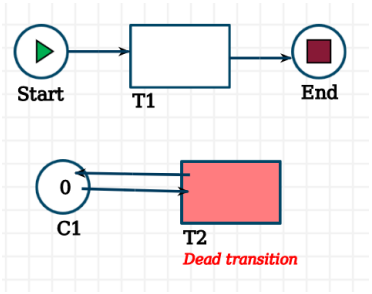

Figure 15: The results after analyzing the model in figure 14

Example of unreachable end

This example illustrates a model in which the end condition will never be satisfied, i.e. unreachable. Obviously, since there are no incoming edges in the end condition, the whole process can never end. As earlier mentioned, the analyzer will check if the final marking can be reached, where the end condition has a marking value of 1, which in our case does not happen.

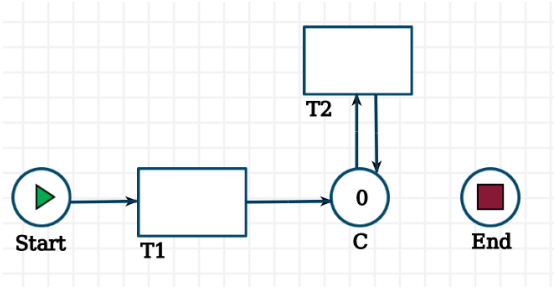

Figure 16: YAWL model with an unreachable end

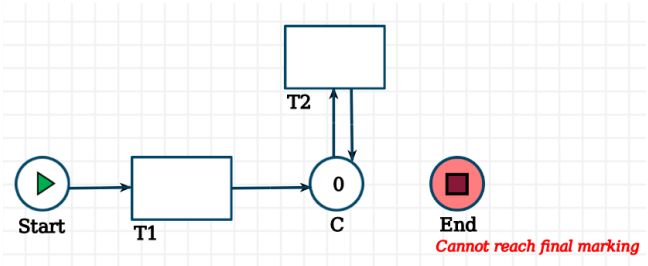

Figure 17: The results after analyzing the model in figure 16

Example of no proper completion

For a net to be sound, it also needs to have a proper completion. This example will show a scenario where this is not the case. This model does reach the end, but the AND-split task `T1` will also initiate another branch that will go on forever and thus cause an improper completion.

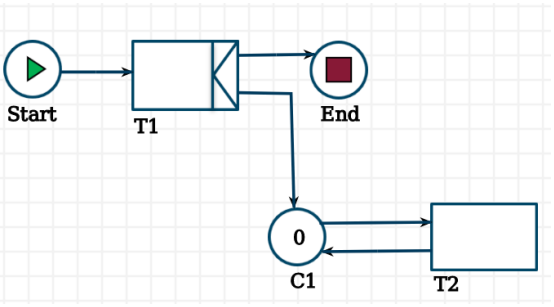

Figure 18: YAWL model with no proper completion

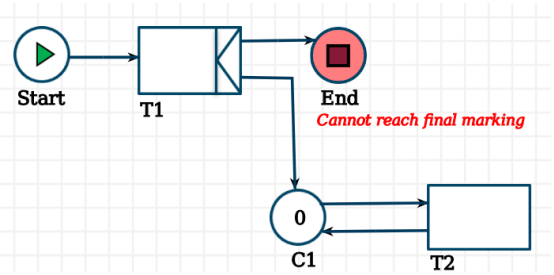

Figure 19: The results after analyzing the model in figure 18