UNIVERSITÄT AUGSBURG

Algorithms and Tools for Petri Nets - Proceedings of the Workshop AWPN 2018

Robert Lorenz , Johannes Metzger (Eds.)

Report 2018-02 Oktober 2018

INSTITUT FÜR INFORMATIK
D-86135 AUGSBURG

Contents

Full papers

Extended Abstracts

Preface

The Algorithms and Tools for Petri Nets (AWPN) workshop is organized by the Special Interest Group "Petri nets and related system models" of the German Gesellschaft für Informatik (GI) with the focus on algorithms and tools for Petri nets.

AWPN 2018 took place at Augsburg University in Germany on October 11-12. The emphasis of the meeting was on the exchange of experiences and discussions.

Papers did not undergo a detailed reviewing process, but were inspected for relevance with respect to the topics of AWPN 2018. Papers related to theoretical issues for analysis, validation and simulation of Petri nets, on application and adaption of Petri net based modeling techniques to different application areas and on experiences with the implementation of visualization, analysis, simulation and teaching tools were presented at AWPN 2018. Overall, the quality of the submitted papers was very good and all submissions matched the workshop goals very well.

The workshop startet with an invited talk on *Carl Adam Petri's Synchronic Distance* by Jörg Desel (without being published in the proceedings). Seven full papers and four extended abstracts were presented at the workshop. We thank the authors and the presenters for their contributions.

Enjoy reading the proceedings!

Robert Lorenz, Johannes Metzger
October 2018

Part I

Full Papers

Spike - a command line tool for continuous, stochastic & hybrid simulation of (coloured) Petri nets

Jacek Chodak, Monika Heiner

Computer Science Institute, Brandenburg University of Technology
Postbox 10 13 44, 03013 Cottbus, Germany
jacek.chodak@b-tu.de, monika.heiner@b-tu.de
http://www-dssz.informatik.tu-cottbus.de

Abstract Spike is a command line tool for continuous, stochastic & hybrid simulation of (coloured) Petri nets (PN). It allows import from and export to various PN data formats. Its abilities comprise the manipulation of PN models by changing arc weights, markings or functions. It also unfolds coloured PN. To comply with the demand for reproducible simulation experiments, Spike is supported by a script language which allows for model and simulation configuration.

Keywords: continuous, stochastic, hybrid, coloured (hierarchical) Petri nets · simulation · configuration · reproducibility

1 Objectives

Many tools allow simulation of PN models. However, most of them have a graphical user interface (GUI) which usually involves additional dependencies and hinders batch processing. Simulation of PN models can be a time and memory consuming process. For performance reasons such simulations should be delegated to run on a server side. Due to the reasons above, GUI tools are not well suitable to be executed on a server.

Running simulation on a server helps to save user resources and speeds up simulations. On a server, the user can schedule multiple simulation experiments which can be executed simultaneously or sequentially. Often, a user wants to check how a model behaves for different sets of parameters. In this case, the user is forced to make changes in the model using an appropriate tool. Each time a model is changed, the simulation needs to be repeated. To compare how a model behaves under different types of simulation (stochastic, continuous, hybrid), it is necessary to configure, each time separately, the simulation and the model. This scenario can require to use separate tools for different types of simulations.

To ensure reproducible simulations, all parameters of model and simulation configurations have to be saved. To simplify the workflow, the configuration of the model and simulation should be supported by a script language, which allows easy modification of parameters.

Spike is meant for running different types of simulations of PN models. It is supported by configuration scripts which permit to configure the model as well as the simulation and to run sequentially multiple simulation experiments. Storing configurations in scripts allows Spike to reproduce simulations in a user friendly way.

2 Functionality

Spike is a slim, but powerful brother of Snoopy [2] - it is the latest addition to the PetriNuts family of tools for modelling, analysis and simulation with Petri nets, specifically tailored to the investigation of biochemical reaction networks. The main focus of Spike lays on efficient and reproducible simulation of PN models. Spike also offers import and export of various exchange data formats and some basic reduction of PN models.

Simulation Similar to Snoopy, Spike is capable to run three basic types of simulations: stochastic, continuous and hybrid, each comes with several algorithms. Simulation of coloured stochastic, continuous and hybrid PN models is supported by unfolding them automatically to uncoloured models.

A given model is simulated according to the specified simulation type, despite of place and transition types in the model. That means all places and transitions are converted to the appropriate type. For example if a user wants to run stochastic simulation on a continuous model, all places and transitions are converted to the stochastic type. Likewise for stochastic models to be simulated continuously, all stochastic transitions are converted to continuous type.

Simulation results can be saved in CSV files which can be used later for analysis and visualisation. They may comprise user-defined combinations of traces of place markings, transition rates, as well as observers (auxiliary variables).

Conversion Spike supports the following data formats of PN models:

- ANDL and CANDL - human readable formats for Petri nets and Coloured Petri nets, respectively, used internally by the PetriNuts framework,
- SBML (the Systems Biology Markup Language) - an XML-based representation format designed to exchange computational models of biological processes [4],
- PNML - an XML-based interchange format for Petri nets [6],
- ERODE - a tool for the evaluation and reduction of chemical reaction networks [1].

Table 1 shows the data format conversions currently supported by Spike.

Reduction Spike is also able reduce structurally the model, by pruning clean siphons and constant places. In both cases, clean siphons and constant places

Table 1. Data format conversions currently supported by Spike.

From	To
ANDL	PNML, ERODE
CANDL	ANDL, PNML, ERODE
SBML	ANDL, PNML, ERODE
ERODE	ANDL, PNML

can be calculated by Spike or loaded from a file. It is also possible to save results of the calculation to a file, which can be used later by Spike or for other analysis purposes.

Further reductions may be applied by converting a model to the ERODE format, if the model is to be read as ordinary differential equations (ODEs). Reductions of a model can have a significant impact on simulation time.

Reproducibility To comply with the demand for reproducible simulations, Spike reads a script which allows for model and simulation configuration. The structure of the script is easily readable for the user and does not require any special tools for editing: a simple text editor is enough.

The configuration script allows among others:

- definition of constants, e.g.:

```
constants: {
  // name of a group, see ANDL specification
  all: {
    /* if constant does not exist
    * then it will be created and
    * can be used in the configuration,
    * for example in defining a place marking
    */
    M: "D/2 + 1"
  }
}
```

- set marking for places, e.g.:

```
places: {
  // example of use of the newly created constant M
  P: "1000`(M,M)"
  P_2_2: 500
}
```

- definition of auxiliary variables (observers) which allow for extra measures by defining numerical functions; depending on the type of observer, it can be defined for places, transitions or simultaneously for places, transitions and constants, e.g.:

```
observers: {
  place: {
    OP01: {
      function: "P_1_1 + P_2_3"
    }
  }
}
```

- defining multiple simulation configurations, which permits to run multiple experiments for one model configuration;
- defining multiple exports of simulation results by use of regular expressions over the nodes of which the simulation traces are to be recorded; it is possible to combine the results of places, transitions and observers, coloured and uncoloured, in one file, e.g.:

```
export: {
  // Array of places to save,
  // including colored places like P
  // in this example (if empty, export all)
  places: ["P_1_1", "OP01", "Grid.*", "D", "P"]
  // Array of transitions to save,
  // including colored transitions
  transitions: ["t3_1_1_1_2", "t3", "t3" ]
  // Array of observers to save (if empty, export all)
  observers: ["M01", "OT01"]
  to: "sim01-file01.csv"
}
export: {
...
}
```

3 Architecture

Spike is written in C++, it is available for Linux, Mac and Windows. It has a modular structure, where the modules are basically decoupled from each other. This allows for easily adding new features.

Modules communicate with each other using command patterns and a queue of commands which is globally accessible. Each module has its own list of commands with specific parameters, which must be registered to the queue during initialisations of a module. Table 2 shows a summary of all commands currently available in Spike.

Commands are processed in a sequential way. Each command is executed by the module which is responsible for it. Let's consider the following use case illustrated in Fig. 1 – the execution of a simple configuration script. When the command "exe" is at the head of the command queue, the module Configure will execute it. During execution, the configuration module communicates with other modules by appending new commands to the queue.

Table 2. List of Spike's modules with their commands.

Module	Command	Description
Main	version	display version of Spike
CLI	help	display help for a given command
Configuration	exe	execute configuration script
Converter	load	load a model from a given file
	save	save a model to a given file
	prune	prune a model
	eval	evaluate constants and places
	unfold	unfold a coloured model
Simulation	sim	run a simulation of the model

4 Use cases

Spike permits to run simulations on a server as well as on the user side. It can be done in batch mode or by integration of Spike as a service. Algorithm 1 illustrates a typical scenario, which allows, e.g., to compare how a model behaves under different types of simulation algorithms or under different configurations of a given simulation algorithm.

The discussion of more scenarios exceeds the given space limit.

Algorithm 1: Use case to run multiple simulation configurations.

```
1 Load model
2 Set model configuration
3 Set simulation configurations
4 for each simulation configuration do
5 |   Run simulation
6 |   Save results of the simulation
7 end
```

5 Comparison with other tools

So far there is no tool, which allows to conveniently configure simulation experiments with support of a wide range of time-dependent Petri nets classes and simulation types. For comparison of Spike, two tools were chosen which provide partially similar functionality.

COPASI [3] supports stochastic and hybrid simulation of biochemical networks. It allows the definition of multiple result exports. However, there is no direct support for Petri nets. Configuration files follow an XML-based format, which hinders their readability by a user.

Renew (The Reference Net Workshop) [5] supports modelling and simulation. It permits simulation on a server [7]. However, the Renew core does not support quantitative (stochastic, continuous and hybrid) net classes.

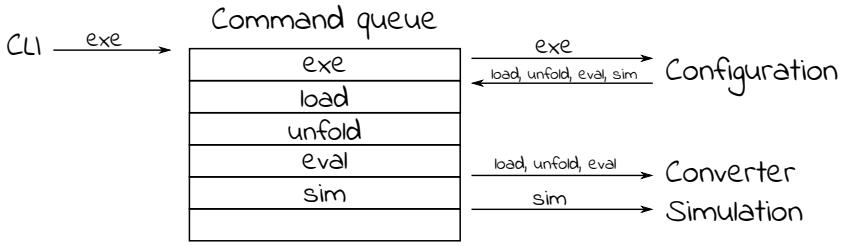

Figure 1. This example shows the flow of commands through the modules of Spike when a user types the command "exe".

To summarise, Spike is specifically suitable for scenarios, when user experiments require different configurations of a model and/or simulation.

6 Installation

Spike is available for Linux, Mac and Windows. Binaries can be downloaded from its website `http://www-dssz.informatik.tu-cottbus.de/DSSZ/Software/Spike`. See also Spike's website for installation instruction and manual. Currently, Spike is under extensive development and we are open for suggestions.

Acknowledgement

Spike uses software libraries that have been developed by former staff members and numerous student projects at Brandenburg University of Technology, chair Data Structures and Software Dependability.

References

1. Cardelli, L., Tribastone, M., Tschaikowski, M., Vandin, A.: ERODE: A Tool for the Evaluation and Reduction of Ordinary Differential Equations. In: TACAS 20017. pp. 310–328. Springer, LNCS 10206 (2017)
2. Heiner, M., Herajy, M., Liu, F., Rohr, C., Schwarick, M.: Snoopy – A Unifying Petri Net Tool. In: ATPN 2012. pp. 398–407. Springer, LNCS 7347 (2012)
3. Hoops, S., Sahle, S., Gauges, R., Lee, C., Pahle, J., Simus, N., Singhal, M., Xu, L., Mendes, P., Kummer, U.: Copasi–a complex pathway simulator. Bioinformatics **22**(24), 3067–3074 (2006)
4. Hucka, M.: Systems biology markup language (sbml). Encyclopedia of Computational Neuroscience pp. 2943–2944 (2015)
5. Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Köhler, M., Moldt, D., Rölke, H., Valk, R.: An Extensible Editor and Simulation Engine for Petri Nets: Renew. In: ATPN 2004. pp. 484–493. Springer, LNCS 3099 (2004)
6. Petri Net Markup Language (PNML): Systems and software engineering – High-level Petri nets – Part 2: Transfer format (2009), ISO/IEC 15909–2:2011
7. Polasek, P., Janousek, V., Ceska, M.: Petri net simulation as a service. In: PNSE@ Petri Nets. pp. 353–362 (2014)

Can a Single Transition Stop an Entire Petri Net?

Jörg Desel

FernUniversität in Hagen, Germany
joerg.desel@fernuni-hagen.de

Abstract. A transition t stops a place/transition Petri net if each reachable marking of the net enables only finite occurrence sequences without occurrences of t (i.e., every infinite occurrence sequence contains occurrences of t). Roughly speaking, when t is stopped then all transitions of the net stop eventually. This short contribution shows how to identify stopping transitions during the construction of the coverability graph of the net.

1 Introduction

We consider the following problem in this paper: Assume a place/transition Petri net (without inhibitor arcs or capacity restrictions) and a transition t of this net. Given any reachable marking m of the net, can we eventually stop the behavior of the net by forbidding occurrences of t in occurrence sequences enabled at m, or, equivalently, does no reachable marking m enable an infinite occurrence sequence without occurrences of t?

Apparently, this question is relevant for several applications of Petri nets. For example, given a robot (or any kind of machine) modeled by a Petri net, can some component modeled by a particular transition be used as a cut out? As we know from our computers, immediate stops are not always desirable, but rather forced shut down processes. A transition t stops a Petri net model if it enforces a shutdown process which will eventually lead to a marking which enables no transition, except possibly transition t.

The problem tackled in this article could be solved by any standard mechanism involving temporal logics, for example the temporal logic LTL. In [4] it is shown that the model checking problem for Petri nets and LTL formulas is decidable, although according algorithms applied to unbounded Petri nets have a huge complexity. Instead, this article provides a solution which is purely based on Petri net analysis techniques. A typical advantage of these techniques is that the user gets more insight to the actual behavior of the net. Often, analysis methods tailored for Petri nets are more efficient as analysis techniques based on a translation to other languages, at least for certain classes of inputs. This might also be the case for the approach presented in this paper; a detailed study to identify such classes is, however, still missing and a topic for further research.

Throughout this paper we consider place/transition Petri nets without capacity restrictions or inhibitor arcs. We call these place/transition Petri nets

just nets. As usual, we assume that the sets of places and transitions of a net are finite. We do, however, consider unbounded nets, i.e., nets with unbounded places (a place is unbouded if, for any number b, some reachable marking assigns more than b tokens to the place). We assume the standard notations of nets to be known, including the concept of a coverability graph for unbounded nets (this concept goes back to [5]). The coverability graph represents aspects of infinite behavior by finite means, and thus abstracts heavily from behavioral details. However, it can be used to identify if a place is bounded. As we will show, the coverability does not contain information about stopping transitions, but a modified variant of it does.

For definitions and notations, see any textbook on Petri nets, e.g. [6], or [3]. In particular we will use the concepts of reachability tree and graph, as well as of coverability tree and graph. Notice that often the coverability graph is defined as a result of a non-deterministic algorithm and his hence not unique. The algorithm constructs the reachability graph for bounded nets and a finite coverability graph otherwise.

2 Terminating Petri nets

To warm up, we first consider the question whether a net terminates eventually, i.e., whether all its occurrence sequences are finite.

Obviously, a bounded net terminates if and only if its reachability graph has no cycles. In fact, if the reachability graph has a cycle, then each occurrence sequence from the initial marking to any marking represented by a vertex of the cycle can be extended infinitely, following the arcs of the cycle (remember that each vertex of the reachability graph represents a reachable marking). Conversely, a bounded net has only finitely many reachable markings, because the set of places of the net is finite. Since each occurrence sequence corresponds to a directed path of the reachability graph, each infinite occurrence sequence corresponds to a directed path that passes through at least one vertex more than once; thus the reachability graph has a cycle.

Unbounded nets do not terminate anyway. To see this, consider the construction of the reachability tree. Since the set of transitions is finite, each vertex of this tree has finitely many immediate successors. By König's Lemma, the tree has an infinite path, corresponding to an infinite occurrence sequence.

Hence, an obvious algorithm to check termination of a net first checks boundedness, for example by the coverability graph construction. In case the considered net is bounded, the algorithm constructs the reachabilty graph and checks whether this graph has a cycle. Actually, this two-step approach is not necessary, because the coverability graph of a bounded net equals its reachability graph, and cyclicity of this graph is implicitly checked during the coverability graph construction. A perhaps more elegant algorithm[1] first adds a place to the net which has all transitions of the net in its pre-set and no transition in its

[1] communicated by Karsten Wolf

post-set, and then checks boundedness of this place, again by construction of the coverability graph. Obviously, this additional place is bounded if and only if the length of all occurrence sequences is bounded. Since the set of transitions is finite, this is the case if and only if there is no infinite occurrence sequence.

3 Termination after stopping a transition – bounded case

We now come back to the question asked initially: Does a net terminate if a given transition t of the net is stopped eventually? This is the converse of the question: Is there an infinite occurrence sequence, enabled at some reachable marking, without occurrences of t? An even simpler formulation of the same property is: Is there an initially enabled infinite occurrence with only finitely many occurrences of t? In fact, an infinite occurrence sequence enabled at a reachable marking m is suffix of an infinite sequence enabled initially, and the finite prefix up to m can contain only finitely many occurrences of t. Conversely, assume an infinite occurrence sequences containing only finitely many occurrences of t. Then the minimal prefix containing all these t-occurrences leads to a reachable marking which enables the according infinite suffix without occurrences of t. In the sequel, we say that a transition t stops the net if every infinite occurrence sequence of the net contains infinitely many occurrences of t.

For bounded nets, there is again a very simple algorithmic solution to the problem whether a transition t stops its net: Construct the reachability graph and check whether every cycle of this graph contains at least one arc labeled by t. If there is a cycle without t-labeled arc, then – as above – some infinite occurrence sequence starts with a finite sequence to some vertex of this cycle (which might include occurrences of t) and then runs along the cycle infinitely. Conversely, assume that each cycle has at least one t-labeled arc. Each infinite occurrence sequence passes through some vertex of the reachability graph infinitely often. All (infinitely many) subsequences between two subsequent passes through that vertex correspond to a cycle. By assumption all these subsequences contain an occurrence of t, whence t occurs infinitely often in the sequence.

Algorithmically, we can delete all t-labeled arcs in the reachability graph (which does *not* necessarily lead to a connected graph) and check for cycles.

4 Dito – unbounded case

Finally, we consider the case that the considered net is unbounded. Does it eventually terminate provided a given transition t occurs only finitely often? Unfortunately, the coverability graph does not bring immediate help. Consider the simple example of a net with only one initially unmarked place, a single input transition i, and a single output transition o (see Figure 1).

In this example, transition i eventually stops the net, whereas transition o does not. However, both transitions occur in the coverability graph in quite the same way, namely as labels of arcs leading from the ω-marking labeled by ω to itself. These are the only cycles of this coverability graph. While the complete

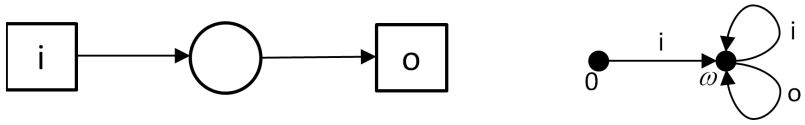

Fig. 1. A simple net and its coverability grap

coverability graph does thus not lead to an algorithmic solution, we can solve the problem during its construction, as shown below.

Remember that, during the (nondeterministic) construction of the coverability graph, we compare new ω-markings with already constructed ω-markings. An ω-marking is a marking of the places of a net where some places can have the entry ω, meaning that these places can carry arbitrarily many tokens. When a new vertex of the coverability graph is constructed, the algorithm compares the ω-marking m corresponding to this new vertex with the ω-markings m' corresponding to vertices which are on paths from the initial vertex to the new one (according to the graph constructed so far). If, for all places, the new marking m is identical to m', then the new vertex is identified with the vertex corresponding to m'. Otherwise, if $m(s) \geq m'(s)$ for each place s (where $\omega > n$ for every integer n), then m is modified as follows: For each place s with $m(s) > m'(s)$, we set $m(s) := \omega$, because the sequence from the vertex corresponding to m' to the newly constructed vertex can be repeated arbitrarily often, leading to an unbounded token growth on the place s.

In the above example, the marking reached by the occurrence of transition i is greater than the initial marking for the only place of the net; hence in the coverability graph this place receives an ω-entry. Further occurrences of transition i are possible, leading to the same ω-marking, because ω already means "arbitrarily many". Notice, however, that transition i can occur infinitely often, no matter if transition o occurs, whereas o cannot occur arbitrarily often without i, and in particular there is no infinite occurrence sequence $o\ o\ o \ldots$ enabled at any marking, a fact which is not reflected by the coverability graph.

Now we come back to the problem whether some transition t eventually stops its net. To this end, we modify the coverability graph construction as follows. When adding a new vertex and comparing ω-markings with previously reached ω-markings, we also look at the occurrence sequences leading from the previously reached ω-marking to the current one. If all such sequences contain at least one occurrence of t, we proceed as in the original algorithm. Otherwise, we consider the occurrence sequences without t leading from a previously reached ω-marking m' to the actual ω-marking m which satisfy $m'(s) \leq m(s)$ for each place. We define the effect of an occurrence sequence to a place s as the difference between the number of occurrences of output transitions in the sequence and the number of occurrences of input transitions of the sequence. That is, by the occurrence of the sequence, the token count on s is decreased or increased by the effect of the sequence to s. If $m(s) \neq \omega$ then the effect of the occurrence sequence to s must not be negative by construction. However, if $m(s) = \omega$, then the occurrence

sequence might actually decrease the number of tokens on s, as it happens in our example by the short occurrence sequence o.

If we find an occurrence sequence (without t) from some suitable previously reached marking m' to m with non-negative effect to all places s, then we stop the algorithm with output *no*, i.e., the algorithm comes to the result that transition t does not stop the net. Otherwise we proceed as in the usual construction of the coverability graph. If the construction algorithm reaches its regular end, i.e., if it never answered *no*, then it delivers the output *yes*, thus detecting that t actually stops the net.

If we apply our modified algorithm to the above trivial example and ask whether o stops the net, then we immediately identify the occurrence sequence $m_0 \xrightarrow{i} m$ which neither contains o nor has a negative effect on any place (but a positive effect on the only existing place). So the algorithm terminates with output *no*. If we apply it with respect to transition i, then the only relevant cycle is given by the arc labeled by o, which is actually a loop. The short occurrence sequence o decreases the token count of the only existing place. So it has a negative effect to this place. Therefore, the algorithm finally constructs the complete coverability tree and ends with output *yes*.

To prove the algorithm correct, we first observe that it proceeds like the usual coverability graph construction algorithm, except that it might terminate earlier. So it terminates eventually, as the unmodified coverability graph construction algorithm terminates eventually.

If the algorithm terminates with ouput *no*, then there is an ω-marking in the coverability graph constructed so far which enables an occurrence sequence without occurrences of t and with non-negative effect to all places. Remember that an ω-marking enables a finite occurrence sequence if the regular marking constructed by replacing all ω-entries by the length of the sequence enables the occurrence sequence (this replacement ensures that none of the transitions of the sequence lacks tokens on places marked by ω). By construction of the coverability graph, we can actually reach such a marking by pumping up the sets of tokens on all ω-marked places. Since the occurrence sequence has no negative effect to any place, the marking reached by the sequence assigns at least as many tokens to each place as the marking enabling the sequence. Therefore, the occurrence sequence can be repeated infinitely often. Thus, transition t does not stop the net.

Conversely, assume that a transition t does not stop the net. We proceed indirectly and assume that the algorithm stops with output *yes*, thus constructing the full coverability graph. Since t does not stop the net, there exists a reachable marking m that enables an infinite occurrence sequence without t. In this occurrence sequence, we reach markings m' and m'' (reached after m') such that $m''(s) \geq m'(s)$ for each place s (this is the core of the proof of finiteness of the coverability graph, based on Dickson's Lemma). Let σ be the occurrence sequence leading from m' to m''. Clearly, σ also does not contain t, and it has a non-negative effect to all places. It is known that the ω-markings of the coverability graph cover all reachable markings. Hence some ω-marking m'_ω covers m',

i.e., $m'_\omega(s) \geq m'(s)$ for each place s. During the construction of the coverability graph the algorithm will find out that m'_ω enables σ, which leads to another ω-marking m''_ω covering m''. However, comparing m''_ω with m'_ω and considering the occurrence sequence σ would lead to an earlier termination of the algorithm with output *no* – a contradiction.

5 Conclusion

We have shown how to decide whether a single transition is able to stop an entire net, i.e., with only finitely many occurrences of t the net terminates eventually. The proposed algorithm can easily be generalized to sets of transitions (if we forbid all transitions of this set at some marking, will the net eventually terminate?). Another obvious generalization refers to arc weights; the procedure works for nets with arc weights with only small changes.

Other tool for identifying transitions that stop a net are given by transition invariants, which are closely related to cyclic occurrence sequences, and by transition sur-invariants, which are related to occurrence sequence with non-negative effect to all places. Both types of invariants can be derived by linear algebraic means, see e.g. [1]. These techniques lead to much more efficient algorithms, but unfortunately provide only sufficient criteria for termination problems.

Yet another approach to solve the problem is to consider cycles in coverability graphs (see [2]), representing cyclic behavior. The calculation of such cycles requires, however, by far more effort than the algorithms suggested in the present contribution.

References

1. Jörg Desel. Basic linear algebraic techniques for place/transition nets. In Reisig and Rozenberg [7], pages 257–308.
2. Jörg Desel. On cyclic behaviour of unbounded Petri nets. In Josep Carmona, Mihai T. Lazarescu, and Marta Pietkiewicz-Koutny, editors, *13th International Conference on Application of Concurrency to System Design, ACSD 2013, Barcelona, Spain, 8-10 July, 2013*, pages 110–119. IEEE Computer Society, 2013.
3. Jörg Desel and Wolfgang Reisig. Place/transition Petri nets. In Reisig and Rozenberg [7], pages 122–173.
4. Javier Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Inf.*, 34(2):85–107, 1997.
5. Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969.
6. Wolfgang Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.
7. Wolfgang Reisig and Grzegorz Rozenberg, editors. *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, based on the Advanced Course on Petri Nets, Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*. Springer, 1998.

Using the Event Coordination Notation for Validation

Ekkart Kindler[1], Pétur Ingi Egilsson[1], and Lom Messan Hillah[2]

[1] DTU Compute, Technical University of Denmark
[2] Univ. Paris Nanterre & Sorbonne Université CNRS, LIP6, Paris

1 Introduction

The *Event Coordination Notation* (*ECNO*) can be used for defining how related model elements in a system must coordinate their behaviour. It can be used to model software systems and then generate the code from these models fully automatically [1]. In a previous study, we demonstrated that a large software system, a workflow management system, could be modelled by ECNO and then the code for it could be generated automatically [2, 3].

The ECNO, however, can also be used for rapid prototyping, where the generated code is just a means to try out specific behaviour; the generated code is not the final result. Once the prototyping results in the desired behaviour, the system is implemented manually; for example, when the code runs in a distributed way[3]. In that case, the ECNO model can serve as more than just a prototype for the final implementation: in the end, we can use the original ECNO model to validate the final implementation.

This idea first came up when we used ECNO for modelling a large and distributed banking system [4] for prototyping. It roughly took a day or two to come up with a first model and play with the automatically generated code. Initially, we used this model for prototyping only. Later, in his masters' project, Egilsson [5] designed and implemented an extension of the ECNO Tool, which then allowed us to validate an implementation of the banking system against the ECNO prototype.

This paper discusses this idea of validating manual implementations against ECNO models and some of its main ingredients. In Sect. 2, we briefly introduce the core ideas of ECNO by using an example. In Sect. 3, we discuss the state space of ECNO models. In Sect. 4, we describe traces as an abstraction of the observed behaviour of the manually implemented system. In Sect. 5, we show how a trace of the implementation can be validated against the ECNO model by mapping the trace to the state space of the ECNO model.

2 ECNO by Example

To explain the main idea of ECNO, we use the simple introductory example from the ECNO report [1]. It models a company in which *workers* are required

[3] Up to now, the ECNO code generator cannot generate such code automatically.

to jointly do some *jobs*. Only if all the workers that are *needed* for the job are available, the job can be done. To make things slightly more interesting, we assume that the workers share *cars* for coming in for work and for leaving again. Therefore, the workers sharing the same car will *arrive* and *depart* together. And only when a worker is at *work*, the worker is available for *doing* a job.

The underlying structure of this system is modelled as a class diagram in Fig. 1. The association between classes Worker and Car represent which workers share the same car. The association between classes Worker and Job represents which workers are *needed* for doing a job. A worker can be *assigned* many jobs – but a worker can only do one job at a time. Note that a single job may need more than one worker to participate.

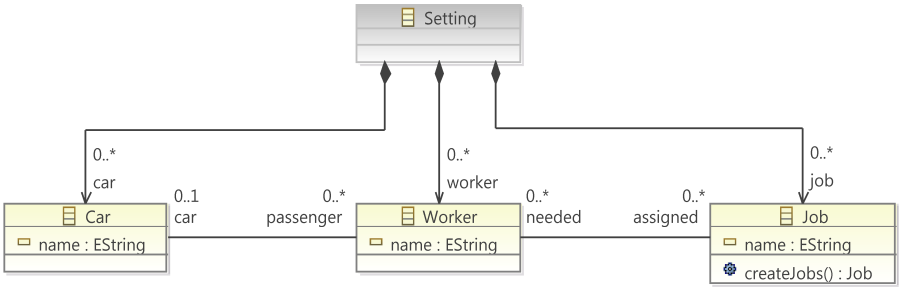

Fig. 1. Structural model

Figure 2 shows an object diagram with an example situation of a company with some workers, jobs and cars and how they are related.

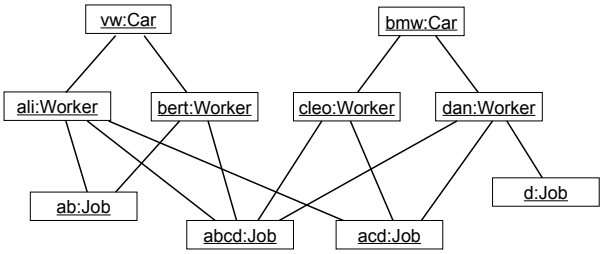

Fig. 2. Some configuration of a company

For modelling the system's behaviour, the ECNO distinguishes between the *life cycles* of objects and the *coordination* of the behaviour among the different objects. For clarity, objects that have a life cycle and for which the ECNO defines how to coordinate their behaviour are called *elements*.

The basis for defining and coordinating behaviour in ECNO are *events*. To this end, the ECNO allows us to define the types of events in the system. In our workers example, the events are arrive and depart, which mean that the workers and cars are arriving at or departing from the work site. Moreover, there is the event doJob, which means that a job is done; and there is an event cancelJob, which means that an existing job is cancelled.

These events are formally defined in the ECNO *coordination diagram* of Fig. 3 – shown as rounded rectangles. More importantly, the coordination diagram defines the coordination of how different elements are supposed to participate in events together. To this end, it equips some parts of the structural model from Fig. 1 with some additional annotations, which are explained below.

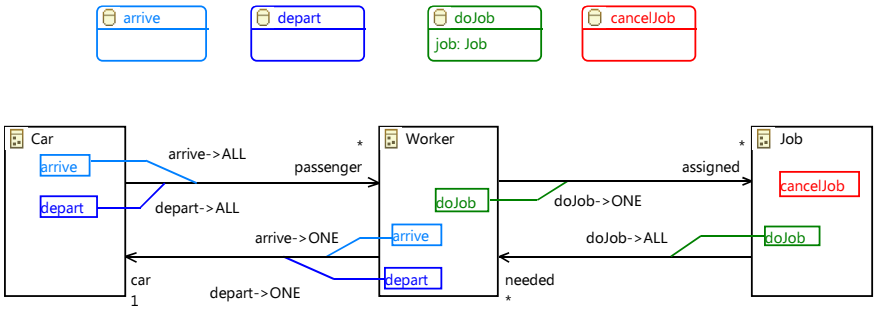

Fig. 3. Coordination diagram

When a Car is ready to participate in an arrive event, the coordination diagram requires that all workers sharing that car participate in this arrive event, too. To this end, there is a box with label arrive in the Car. This box is called a *coordination set* for event arrive of element Car. This coordination set is linked to the reference passenger with an annotation arrive->ALL. This annotation is called a *coordination annotation* and says that for a given car participating in an arrive event, every passenger (i. e. every worker at the other end of the link corresponding to passenger in the given situation) must also participate in the arrive event. Now, for a Worker participating in an arrive event, there is another coordination set for arrive, which imposes additional requirements of other elements participating. The Worker has a coordination set with a coordination annotation arrive->ONE linked to reference car. This means that for a Worker participating in an arrive event, there must be one Car at the end of the link car that must also participate in that arrive event. This gives us a combination of different elements participating in different events. Once such a combination of elements and events meets all the coordination requirements of the coordination diagram for each event, we call this combination an *interaction*. An interaction can be *executed*, which means that all involved elements execute the associated events in an atomic way.

Now, we assume that all workers have arrived at work and that the configuration is as shown in Fig. 2. This means that, according to their life cycle, each worker can participate in a doJob event. Let us assume that cleo would want to participate in a doJob event. Since there is a coordination set for event doJob for Worker, other elements would be required to participate. The coordination annotation doJob->ONE would require one of the jobs assigned to cleo to participate in the doJob event, too. In our example, there are two possibilities: job acd and abcd. Let us investigate job acd. The coordination set of Job for event doJob with the coordination annotation doJob->ALL linked to reference needed, specifies that all workers at the end of the link also need to participate in the doJob event. For the job acd, this would be the workers ali, cleo, and dan. The worker cleo is already participating in the doJob event – we started from there. But, now ali and dan also need to participate. This way, the coordination diagram makes sure that also all needed workers participate in the doJob event when the job acd does.

Note that, in our example, all elements participate in the same event. In general however, there can be more events involved in an interaction, and even a single element can participate in different events at the same time.

As mentioned before, the coordination diagram defines only the coordination of the behaviour among different elements. The life cycle for each element is defined separately: the life cycle defines when an element can participate in an event. In ECNO, one possibility for defining the life cycle for an element is through *ECNO nets*, which are a form of Petri nets. Figures 4 and 5 show the life cycles of a worker and a car, respectively. We omit the life cycle of the jobs here. The transitions labelled with the respective events indicate when the element can participate in which event.

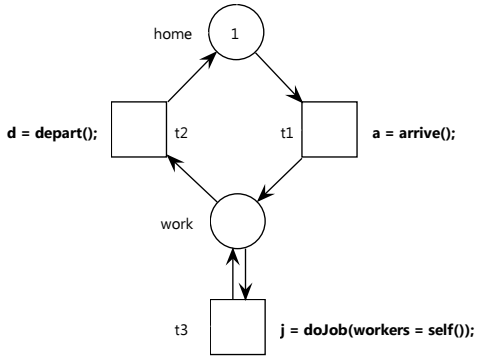

Fig. 4. Life cycle of a Worker

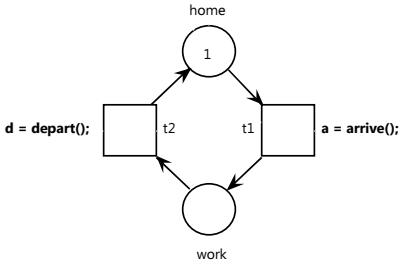

Fig. 5. Life cycle of a Car

In this paper, an ECNO model consists of a structural model, the coordination diagram and the ECNO nets for the life cycles of the elements.

3 ECNO State Space

The state space of such an ECNO model, basically, consists of the states, which represent situations like the one shown in Fig. 2, where additionally for each element the state of its life cycle of and the value of each of its attributes would be stored. In our case, the state of the life cycle would be a vector of natural numbers representing the marking of the resp. ECNO net.

The transitions of the state space would be the interactions. For each interaction in the state space, the involved elements, the links between them and which events are associated with which elements and links are represented.

Figure 6 shows an abstract representation of a part of the state space of the example from Sect. 2. The details of the states are not represented in that figure at all, since the focus of the validation is on the interactions.

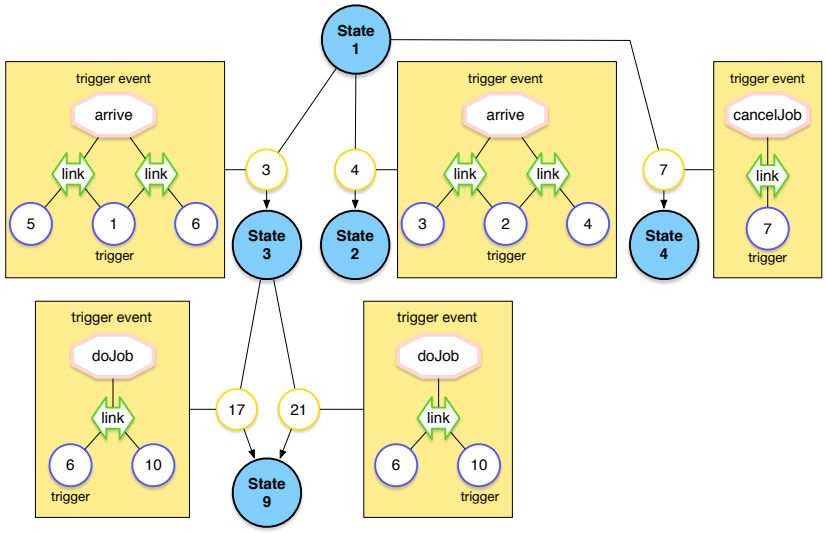

Fig. 6. Abstract representation of a part of the state space

The ECNO Tool[4] provides a *state space generator*, which systematically generates the state space for an ECNO application starting from some initial configuration. Actually, it is also possible for the user to run the ECNO application normally and record the fragment of the state space which the application comes by while running.

[4] See http://www2.compute.dtu.dk/ ekki/projects/ECNO/

4 Traces

As an abstraction of the behaviour of the implementation of the system, we use *traces*. Each trace corresponds to one execution of the system, which is represented as a sequence diagram showing how the different parts of the system invoke each other. Figure 7 shows an example of a trace of the final implementation of our example. This trace corresponds to the workers cleo and dan arriving at work together and then dan doing job d. The separation of the two steps is indicated by the green bars, which represent a state in the state space. But the green bars are actually not part of the trace. They will be computed later when mapping the trace to the state space.

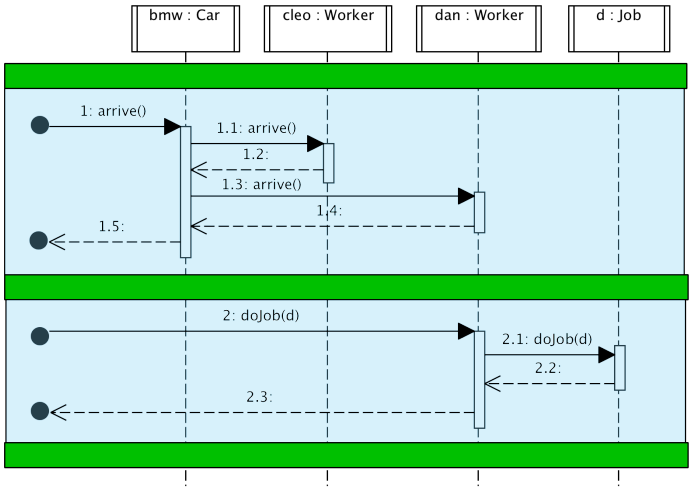

Fig. 7. Implementation behavior: Traces

Egilsson [5] implemented a tracer that could record a trace in the running implementation. It basically used AspectJ for recording the relevant method invocations.

5 Mapping Traces to State Space

The most important part of validating an implementation against an ECNO model (or actually its state space) is identifying parts of the trace which correspond to an interaction in the state space. These parts will be called segments of the trace (the parts between the green bars in Fig. 7). The tricky part is that interactions in ECNO are executed atomically, whereas the segments in traces are not executed atomically and can actually be intertwined. But, the segments

of the communication in the sequence diagram need to correspond to a complete interaction in the state space. And the validation consists of computing segments in traces and mapping them to interactions in the state space.

The mapping algorithm looks for the communication among the same elements in the trace, which also can be found as a link in an interaction. But, there can be more than one invocation for one link between these elements in a trace and the order can be arbitrary. The mapping algorithms starts matching the communication from the beginning of the trace to the interactions in the state space starting from its initial state. And once all communication for an interaction is found the segment is created at this point; the end of which corresponds to the next state. From there, the process continues with a lot of backtracking for alternative matches.

Egilsson [5] implemented such a mapping algorithm and showed that computing such mappings was feasible for the original banking example. Figure 8 shows the idea of this mapping with a short trace and a small excerpt of the state space from the original banking example.

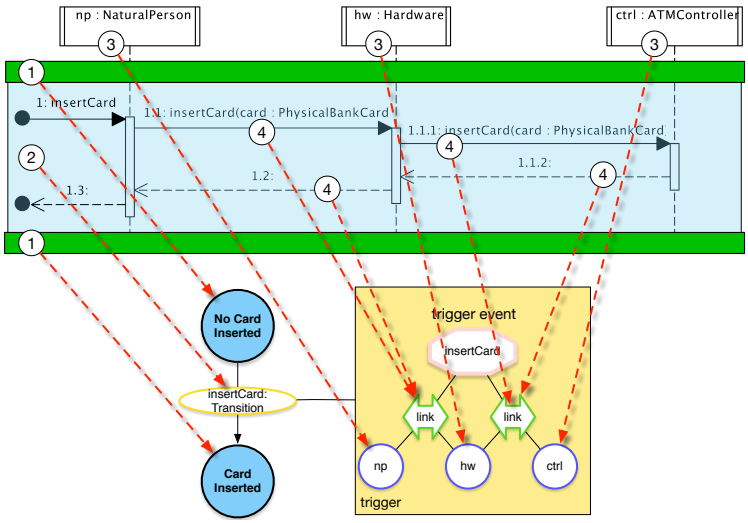

Fig. 8. Mapping

Of course, there are much more details in the mapping algorithm, which we cannot discuss here (see Egilsson [5] for more information).

6 Conclusion

In this paper, we have discussed the main idea of how to validate the final implementation of a system against its original ECNO specification. More details can be found in Egilsson's master thesis [5]. But even this thesis is just the beginning, since ECNO has more features which are not yet covered by the mapper implemented in the master thesis.

References

1. Kindler, E.: Coordinating interactions: The Event Coordination Notation. Technical Report DTU Compute Technical Report 2014-05, DTU Compute, Kgs. Lyngby, Denmark (2014)
2. Jepsen, J.: Realizing a workflow engine with the Event Coordination Notation. Master's thesis, Technical University of Denmark, DTU Compute (2013) IMM-M.Sc.-2013-101.
3. Jepsen, J., Kindler, E.: The Event Coordination Notation: Behaviour modelling beyond mickey mouse. In Roubtsova, E., McNeile, A., Kindler, E., Gerth, C., eds.: Behavior Modeling – Foundations and Applications. International Workshops, BM-FA 2009-2014, Revised Selected Papers. Volume 6368 of LNCS. Springer (2015)
4. The EU FP7 MIDAS project - 318786: Deliverable 5.1: Methods and tools for the intelligent planning and scheduling of test campaigns, http://web.itainnova.es/midas/dissemination/public-deliverables/d5-1-methods-and-tools-for-the-intelligent-planning-and-scheduling-of-test-campaigns/. (2013)
5. Egilsson, P.I.: Model-based software validation: Validating distributed systems against ECNO specifications. Master's thesis, Technical University of Denmark, DTU Compute (2016)

Petriflow in Actions: Events Call Actions Call Events

Juraj Mažári[1,2], Gabriel Juhás[1,2,3], and Milan Mladoniczky[1,2]

[1] Faculty of Electrical Engineering and Information Technology
Slovak University of Technology in Bratislava,
Ilkovičova 3, 812 19 Bratislava, Slovakia,
[2] NETGRIF, s.r.o.,
Jána Stanislava 28/A, 841 05 Bratislava, Slovakia
netgrif@netgrif.com
Home page: http://www.netgrif.com
[3] BIREGAL s. r. o.,
Klincova 37/B, 821 08 Bratislava, Slovakia
biregal@biregal.sk
Home page: http://www.biregal.com

Abstract. In this paper, we present how to model complex business processes and synchronisation between their instances using Petriflow language which extends Petri Nets with other components. Small snippets of code called Actions are introduced to show the capabilities of inter-process communication. Examples of searching, constructing new instances, executing tasks, and data manipulation are provided.

Keywords: Petriflow, Business process modelling, Inter-process communication

1 Petriflow

Petriflow language is an XML based extension of Petri nets[1]. As the underlying model, we use place/transition nets enriched by reset arcs, inhibitor arcs and read arcs. The read arcs appear quite necessary in order to model an unbounded number of concurrent reading of data in a case. To meet modern business modelling requirements other layers were brought to the language on top of Petri nets. Roles are the first layer to extend Petri nets. Roles layer defines who can fire transitions to which they are bound. Data variables were added as the second layer on top of modelled processes. Data variables represent all properties of an instance of a process during its life-cycle. Data variables are bound to transitions by dataRef tag in the underlying XML creating a dataset of the transition. To have more control over process instance data, data field actions were added to Petriflow. Actions can define relations or dependencies between data fields in the model of a process or generate values based on a process instance state. All extensions and layers create the right tool for modelling complex, yet simple to understand models of any process that comes to mind.

Each transition represents a task [2] that can be executed when the transition is enabled. The task life-cycle can be modelled with a Petri net in figure 1. Each task consists of four basic events:

1. Assign - triggered when an actor starts the task execution,
2. Delegate - triggered when an actor assigns the task to another actor,
3. Cancel - triggered when an actor stops the task execution,
4. Finish - triggered when an actor finishes the task execution.

Process roles restrict which event can be triggered by which actor. Data fields can be edited only if the task is active and only by the actor executing the task.

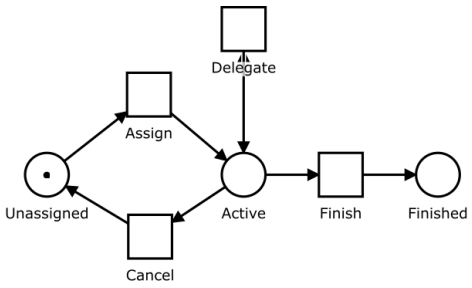

Fig. 1. Task representation net

2 Petriflow universe

Petriflow universe consists of a set of process models and their instances - cases. Each case is a deep copy of the original process model with its own set of data. This concept is similar to relational databases.

Process model represents an entity. In analogy to a relational database, Petriflow model represents a table. It defines a set of data variables one can work with, their data types and validations which is an analogy to table columns. Table rows represent instances of given entity. In Petriflow cases are instances of the original model. Set of tables forms a database, in our universe it is a process driven application working with multiple Petriflow models.

In relational databases, foreign keys are used to create relationships between tables. Since Petriflow is focused on processes we use tasks and events to transfer data between cases and change the state (marking) of a case.

3 Inter-process communication

Inter-process communication can be used to model complex real-life processes by creating an interface between distinct Petriflow models. This interface consists

of models transitions and its events, which can be triggered by an authorised actor. This means that one case can create a new instance of Petriflow model, read and write tasks data, and trigger events on tasks. Each task event (assign, delegate, cancel, finish) can be triggered by an application user or from another process using Actions.

Actions are small pieces of Groovy code, that can be executed at different places in our model. They can be used to change data variables values and behaviour, change case attributes such as title or icon. Customer specific actions can be defined in the application and used in the model as well.

Actions can be triggered by following events:

- assign task,
- delegate task,
- cancel task,
- finish task,
- read value of data field,
- set a new value of data field.

A very important feature of Petriflow is that it allows specifying if the action should be triggered before the event or after. Some actions need to be executed before the event due to change in the marking of the net or if a failure in execution of the action should prevent the event.

Imagine that we have a custom function, one updating document record in a DMS via a web service. A task will update document status on its finish event. It should not be possible to finish this task if the update fails. Other action needs to be executed after the finish event. For example, a task called `aTask` should be assigned to a user immediately after the finish event of the current task produces tokens. Both actions can be assigned to the same event.

Listing 1.1. Example of case search

```
<event type="finish">
  <actions phase="pre">
    <action>
      updateDocumentRecord()
    </action>
  </actions>
  <actions phase="post">
    <action>
      assignTask(aTask, user)
    </action>
  </actions>
</event>
```

As it was already illustrated above, actions can not only be triggered by these events, but actions can trigger events on other tasks (as an event `assignTask` assigned `aTask` to a user inside of an action). Moreover, actions can be used to search for an entity (case, transition, task etc.), create a new case and work with data of a different case.

3.1 Finding objects

Searching for information is a fundamental part of every enterprise application. In terms of inter-process communication, search action is used to find a specific instance to read or change its data values, or to find a task that should be executed in order to continue.

Actions use QueryDSL language to formulate search predicates. Search predicates use an example search object called `it`. It supports complex querying using logical operators and dynamic expressions. Every entity can be searched by any of its properties. This allows to search for a case with a specific data variable value:

Listing 1.2. Example of case search

```
List<Case> cases = findCases( {
    it.dataSet.get("email").eq("mazari@netgrif.com")
} )
Case aCase = findCase( {
    it.author.id.eq(loggedUser().id)
} )
```

For each entity, there are two find actions. One that will return a list of all entities that match the given predicate. The other will always return only one entity, even if multiple entities match the predicate. In that case, the first entity is returned.

In the example above, `it.dataSet` returns the set of all data variables of example case `it`. `get("email")` returns the data variable with id `"email"` of that dataset. The rest of the predicate compares the value of this data variable `"email"` with a string `"mazari@netgrif.com"`.

3.2 Creating new instances

Creating new cases is the basic function in each process-driven application. To create a new instance we only need a reference or the Id of the net. All the other parameters such as case title, colour and author are optional and default values will be used.

Listing 1.3. Example of case constructor

```
Case aCase = createCase("insurance", "My insurance")
```

In this example a new case will be created by calling the `createCase` action with Petriflow net Id and case title.

3.3 Triggering task event

As mentioned before, each of the four basic task events can be triggered by an action. These actions take a transition Id or task reference as the first parameter. The second parameter is the actor whose identity will be used to trigger the

event. If the parameter is omitted logged user will be used instead. Imagine that we have a net with data variable `email` and another net called `"some_net"` with a transition `"first_task"`.

Listing 1.4. Triggering task event

```
if (email.value != null) {
  def user = findUser(it.email.eq(email.value))
  def nC = createCase(identifier:"some_net",author:user)
  assignTask("first_task", nC, user)
}
```

In the example above, `email` refers to a data variable of the current case with id `"email"`. The user whose email equals to the value of the variable `email` is stored to the local variable `user`. Then a new case of `"some_net"` is constructed by the found user stored in the local variable `user`. The constructed case is stored in the local variable `nC`. Finally calling `assignTask` function will assign the found user to the task with id `"first_task"` of the constructed case of `"some_net"`.

3.4 Reading and writing data

Data variables in Petriflow are bound to transitions and can be accessed by reading and writing using these transitions. Since only authorised actors can execute transitions this allows you to restrict access to data values of each model using process roles. In addition, data behaviour can be defined that specifies if the value is visible, optional, required or hidden. This behaviour not only restricts the read and write operations but also restricts triggering of the finish event. If the required data variable does not have a proper value task cannot be finished.

To read the tasks dataset we need to specify the task data of which should be read. This can be done by providing:

1. reference of the task itself by calling `findTask` which returns the task object,
2. reference of the tasks transition (works only on tasks in the current case),
3. tasks id and case.

Function `getData` will use the third method of specifying the task. It will return a map where the key is the data variable Id and the value is the data variable. Returned data variable can be used to access the value of the data variable and other properties such as title, placeholder, behaviour attributes specifying whether it is required, editable, hidden, etc.

Imagine that we have a net with transition with id `"view_limit"`, with data variable with id `"actual_limit"` such that data variable `"actual_limit"` belongs to the dataset of transition `"view_limit"`.

Listing 1.5. Reading data values

```
def usecase = findCase({ it.title.eq("Limits") })
def data = getData("view_limit", usecase)
change actual_limit value {
    data["remote_limit"].value
}
```

In the example above, first case with title `"Limits"` is returned to the local variable `usecase`. Then the dataset of transition `"view_limit"` is returned to the local variable `data` in form of a map described above. Then value of the data variable `"actual_limit"` of the current case is set to the same value as the value of data variable `"remote_limit"` in first case with title `"Limits"` which is stored in the local variable `usecase`. Writing data works similarly to reading, the first parameter specifies a task, the second parameter is a map of new values with data variable Id as the key.

Listing 1.6. Writing data values

```
def aCase = findCase({ it.title.eq("Limits") })
setData("edit_limit", aCase, [ "new_limit": 10000 ])
```

The example above set the value of the data variable `"new_limit"` in the dataset of the transition `"edit_limit"` of the first case with title `"Limits"` stored in local variable `aCase` to the value 10000.

It is not mandatory to provide values of all data variables that are bound to the task. In some scenarios, you want to set new values in steps, especially if the values are dependent on each other. For example, there can be one data field where a user can enter a country name and a second data field where he has to select a city of the chosen country. In that case, you would need to first call `setData` to select the country, then call `getData` to read the list of cities and finally select one city by calling `setData` again.

4 Conclusion

Inter-process communication is the future of Petriflow. This concept has proven to be simple enough for customers to understand it and also simple to model at the same time. Multiple applications using its modelling capabilities were developed and deployed to production.

Inter-process communication opens new possibilities of modelling real-life scenarios. Hotel reservations, online shopping, accounting information system, these are just examples of applications that can be now developed using Petriflow. Whole user management could be replaced with a right Petriflow model allowing to further customise the application.

References

1. Mladoniczky, M., Juhás, G., Mažári, J., Gažo, T., Makáň, M.: Petriflow: Rapid language for modelling Petri nets with roles and data fields. Algorithms and Tools for Petri Nets, 45. (2017)
2. Riesz, M., Seckár, M., Juhás, G.: PetriFlow: A Petri Net Based Framework for Modelling and Control of Workflow Processes. In ACSD/Petri Nets Workshops (pp. 191-205). (2010)
3. Van der Aalst, W. M.: The application of Petri nets to workflow management. Journal of circuits, systems, and computers, 8.01, 21-66. (1998)

Process Communication in Petriflow: A Case Study

Milan Mladoniczky[1,2], Gabriel Juhás[1,2,3], and Juraj Mažári[1,2]

[1] Faculty of Electrical Engineering and Information Technology
Slovak University of Technology in Bratislava,
Ilkovičova 3, 812 19 Bratislava, Slovakia,
Home page: https://fei.stuba.sk
[2] NETGRIF, s.r.o.,
Jána Stanislava 28/A, 841 05 Bratislava, Slovakia
netgrif@netgrif.com
Home page: https://netgrif.com
[3] BIREGAL s. r. o.,
Klincova 37/B, 821 08 Bratislava, Slovakia
biregal@biregal.sk
Home page: http://www.biregal.com

Abstract. In this paper, we explain the usage of Petriflow language in a multi-process environment with inter-process communication via process events. We introduce an example that demonstrates the advantages of Petriflow language when synchronisation between two and more processes is required.

Keywords: Petri nets · workflow · Petriflow · process events · inter-process communication · Petriflow actions

1 Introduction

Petriflow[1] is modelling language for developing process-driven applications. It is based on Petri nets with an extension of reset, inhibitor, and read arcs to enable multiple concurrent readings on a transition. Petriflow can be divided into several layers. The first layer is a Petri net process model itself. Process roles are the second layer of Petriflow language. They provide access control over transitions of a process model. The third layer also called data-set of a process, consists of all data variables of a model. The fourth and the last layer are actions. Actions are small snippets of code written in Groovy programming language. Actions are a powerful tool in Petriflow language. They can define relations between data variables of a process, generate values, communicate with external services or synchronise different instances of processes.

When a Petriflow process model is deployed to an application server, to execute process, an instance of the process is created. In Petriflow, an instance of a process is also called a case. A case is a deep copy of the original process with its specific marking and data-set values. In Petriflow, each enabled transition is

a task[2]. A task consists of four events: assign, cancel, delegate, finish. Each task event can be triggered, by a user or a system, to execute a specific function of a task. Petriflow provides means to react to such events via actions. It is also possible to trigger process events inside of an action and created chain of events and reaction influencing different instances of different processes.

2 Multi-process environment

The real world is very complex. It is one of many reasons why process-driven applications consist of a vast number of processes. It is important to be able to define a way of process communication. Petriflow allows to model inter-process communication with process events and actions. All events can be invoked by some system entity, like system user or another process. Also, a reaction can be defined to every process event in a form of an action. The important part of inter-process communication is the search of all entities of a process model. Petriflow provides search capabilities via library QueryDSL that is easy to use and enables to write both simple and complex queries on every entity of a process model. With Petriflow capabilities, we can model processes that can communicate with each other within an application environment.[3]

3 Process hierarchy

It is rather difficult to model a hierarchy in an observed system with original concept of Petri nets. Even more, if the observed system is very large. A net that tries to capture the hierarchy of a system often results to be large and cumbersome to work with. For modelling hierarchy and encapsulation of components of a system, Nested Petri nets[4] are usually used. Nested Petri nets model a system behaviour in different levels of detail to define relations between parts of an observed system. Nested Petri nets can become difficult to read when a modelled system has multiple levels of complexity. However, modelling hierarchy between processes can be also achieved by writing invocation and reaction to the events in processes in Petriflow language.

Let us introduce an example of this problem. An observed system, which behaviour will be synthesised into Petriflow process models, consists of three entities.

1. Volume - An abstract representation of a whole space inside the system.
2. Folder - An abstract representation of a part of the system space that is a specified part of the volume. The volume of the system can contain one or more folders. A folder also can contain one or more sub-folders.
3. File - An abstract representation of the smallest entity of the system. Every file has to be located inside a folder. A file cannot be further divided and create other system entities.

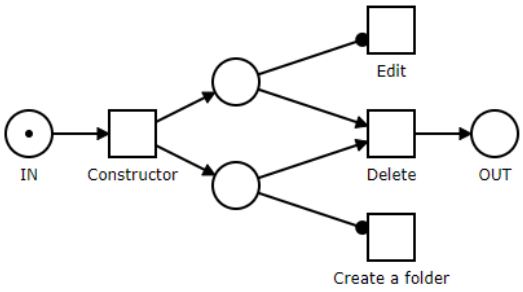

Fig. 1. The Volume process

From an analysis of the three system entities, three Petriflow models are created. Each model consists of transitions to manage the modelled system entity and its data-set.

The Volume process, in the Figure 1, contains data variables for a name of a volume instance, a name for a new folder, and an array of reference objects to all Folder process instances which are located inside the system volume.

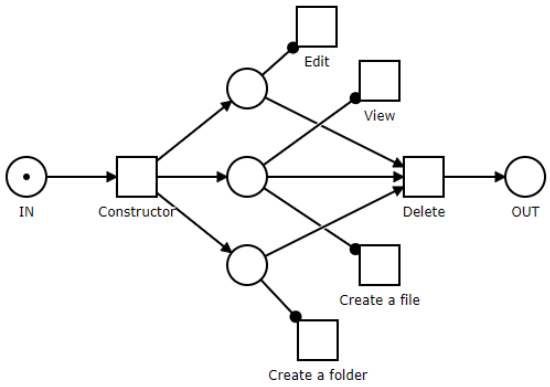

Fig. 2. The Folder process

The Figure 2 illustrates the Folder process. The process has data-set containing data variables of an array of its sub-folders and an array of its files.

The File process, in the Figure 3 is quite simple. Its data-set contains reference to its parent and raw bytes of it content.

It is clear, from the models in Figure 1, Figure 2, and Figure 3, that key transitions are `Create a folder` in the Volume process, `Constructor`, `Create a folder`, `Create a file` in the Folder process and `Constructor` in the File process. As these models are written in Petriflow language each enabled tran-

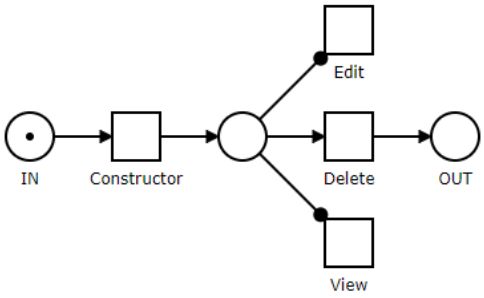

Fig. 3. The File process

sition is a Petriflow task with its process events. First, an action to the finish event on the transition `Create a folder` is defined.

Listing 1.1. An action in the Volume process to create a new folder instance

```
<event type="finish">
  <actions phase="post">
    <action>
      Case folder = createCase("Folder",folderName.value);
      change folders value {
        folders.value.add(folder.id);
        return folders.value;
      };
    <action>
  <actions>
</event>
```

When a user assigns a task `Create a folder`, fills out data about a new folder, e.g. folder's name, and then finishes the task, the action is executed. The new folder instance is constructed by calling the `createCase` function with the Folder process `id` and a new instance title stored in data variable `folderName`. The created instance is stored in the local variable `folder`. Then the `id` of the constructed instance is added to values of data variable `folders`.

This example perfectly expresses the parent-child relationship between the Volume instance and the Folder instance. Likewise, the relationship between different instances of the Folder process and between instances of the Folder and the File process can be defined.

The Second important transition in the processes is the `Constructor` transition. It is the first transition in the process and it is responsible for initialising and setting the process data of a new instance. In the example above, only a name of an instance is sent to the new instance of the Folder process. If a new file is created, it is required to set reference to the parent folder. To achieve this

functionality, the required data value can be sent to the `Constructor` task of a newly created File instance.

Listing 1.2. An action to create a new file and pass its parent folder in the Constructor task

```
<event type="finish">
  <action phase="post">
    <action>
      Case file = createCase("File",fileName);
      change files value {
        files.value.add(file.id);
        return files.value;
      };
      Task constructor = findTask {
        it.title.eq("Constructor")
        .and(it.caseId.eq(file.id))
      };
      if(constructor){
        constructor = assignTask(constructor);
        setData(constructor,["parent":useCase.id]);
        finishTask(constructor)
      }
    </action>
  </action>
</event>
```

The action to create and set up a new file instance is called when a user finishes the task `Create a file` of the Folder process instance. The new file instance is created by calling the `createCase` function with the File process `id` and a name of the new file. The returned reference to the created file instance is added to the data variable of the Folder process instance `file`, which stores references to all files stored in the folder. The next step in the action is to send the required data to the file instance. First, the `Constructor` task of the new file instance is found via `findTask` function with QueryDSL expression parameter. The entity search is based on "Query by example" principle. As it can be seen in the example, the Listing 1.2, the keyword `it` in the expression is the example object of the task entity. If the `Constructor` task is returned, assign it to the currently logged user. The parent folder reference is set by function `setData` where the first parameter is the `Constructor` task and the second parameter is a map of data variables. The key of the map is a data variable `id` of the File process and the value of the map is a desired value of the data variable. In the example above, the Listing 1.2, the data variable with id `parent` is set to reference the current Folder process instance. At last, the `Constructor` task is finished.

4 Conclusion

Inter-process communication modelled in Petriflow language can be applied to the countless applications. As the example illustrated in this paper, it can be used to express hierarchy between instances of different processes. It can be also used to separate often repeated parts of a process as a standalone process model and then referenced from the original process. Even large and complex processes can be modelled with communication via Petriflow process events with ease and preserved readability of Petri nets.

References

1. Mladoniczky, M., Juhás, G., Mažari, J., Gažo, T. and Makáň, M.: Petriflow: Rapid language for modelling Petri nets with roles and data fields. Proceedings of the Workshop Algorithms and Tools for Petri nets 2017, October 19-20, 2017, Technical University of Denmark, Kgs. Lyngby, Denmark, (2017)
2. Riesz, M., Seckár, M., Juhás, G.: PetriFlow: A Petri Net Based Framework for Modelling and Control of Workflow Processes. In ACSD/Petri Nets Workshops (pp. 191-205). (2010)
3. Mažari, J., Juhás, G., Mladoniczky, M.: Petriflow in Actions:Events Call Actions Call Events. Proceedings of the Workshop Algorithms and Tools for Petri nets 2018, October 11-12, 2018, University of Augsburg, Germany, (2018)
4. Irina A. Lomazova, Philippe Schnoebelen: Some decidability results for nested Petri nets. Springer LNCS 1755, 208-220 (2000)
5. Van der Aalst, W. M.: The application of Petri nets to workflow management. Journal of circuits, systems, and computers, 8.01, 21-66 (1998)

Some Ideas for Modeling a Generic IoT- and Edge-Computing Architecture

Daniel Moldt, Dennis Schmitz, Michael Haustermann, Matthias Feldmann, David Mosteller, Thomas Wagner, Jan Henrik Röwekamp, Lawrence Cabac, and Michael Simon

University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences, Department of Informatics, http://www.informatik.uni-hamburg.de/TGI/

Abstract The application area of the Internet of Things (IoT) has continued to increase in recent years. However, adequate modeling metaphors regarding the design and specification of the relationships and interactions between versatile entities are still lacking. Edge computing provides a more specific design due to the given purpose or context of entities (devices, services, and applications).
In this paper, we introduce a four-level architecture for edge computing systems. It helps modelers gain clarity about the relationships and interactions between entities and, thus, specify efficient and well-structured IoT architectures. Using an exemplary context, we discuss the four levels and point out the mental challenges that are addressed by this modeling perspective.

Keywords: Modeling, Software Architectures, Reference Nets, Petri Nets, IoT, Edge-Computing

1 Introduction

The continuously increasing number of processors requires more and more advanced software architectures. Simple single processor-based machines have developed via simple networks and distributed computing systems to the current highly distributed, large-scale systems/ultra large scale systems (ULSS) (see [11]). With the notion of IoT (Internet of Things), an even larger scale of systems has to be addressed. Processors and systems are now omnipresent and require a conceptual embedding in the ULSS architectures. The challenge for informatics is to provide adequate hardware and software systems for such environments.

Shah proposes to integrate IoT systems into (cognitive) agent systems [14]. This contribution does not focus on the technical aspects. Instead, we propose a general modeling architecture that refers to entities at different modeling levels. These relations are supposed to support modelers to choose the right level of abstraction for each entity in ULSS architectures.

Specifically, we address the modeling problems of edge computing applications. Our proposed four-level reference architecture is called Edge-Mulan. In this paper, we restrict our discussions to the general aspects that are relevant

for modeling edge computing-based systems (which might be/are embedded in ULSS). For the EDGE-MULAN architecture, we will explain the main properties of the four-levels and their three general relations briefly.

In the following Section 2 we will briefly give an overview of our previous work, especially MULAN, on which we build our new proposal. The reference architecture EDGE-MULAN for edge computing is then introduced in Section 3 based on the MULAN reference architecture. In Section 4 we embed our work into the work of others. We end with a brief discussion, a conclusion and an outlook in Section 5.

2 Our Previous Related Work

This section covers an excerpt of our previous work that is highly relevant to the context of this paper.

In our group, we have developed several proposals for modeling and implementing complex system architectures. They all are based on Coloured Petri nets [8] and Reference nets [9], which are high-level Petri nets and allow modeling complex systems while covering concurrency, non-determinism, conflicts and several other essential properties of systems. With RENEW we have an integrated development environment (IDE) [3] that, among other purposes, serves for designing, specifying, implementing and executing applications that are built with Reference nets.

Our base reference architecture for the development of complex systems is called MULAN (*Multi-agent Nets Reference Architecture*) [12,2]. MULAN has four major modeling entities *multi-agent system*, *Platform*, *Agent* and *Protocol* with three principal relationships (see Figure 1). Due to the flexibility of our modeling technique, each entity can implement any of these three relationships (e.g., for our MULAN system we often extend platforms to behave as agents). This allows for arbitrarily nested hierarchies. Using value semantics for nets (at most one reference to each Reference net is allowed within the whole system) nesting is strictly hierarchical. Borusan et al. call this the *physical relation* [1] and communication can only be done in a clear context. Nevertheless, reference semantics allow any number of references to an entity. Borusan et al. call this the *logical relation*.

A MULAN model is used to specify an entire multi-agent system (MAS). The multi-agent system net represents the complete system to be modeled including the platforms. The relationship between the multi-agent system net and the platforms is the provision and usage of communication services, respectively. Platforms provide a more specific service environment for autonomous agents depending on the purpose of the overall system. Agents are located on platforms and consist of protocols.

One of the extensions of MULAN is called ORGAN. It takes a more abstract perspective than MULAN and provides a reference architecture for organization-oriented contexts [17]. In [18] organizations (as opposed to agents) are proposed as the basic building blocks of systems and homogeneously published in [17] as

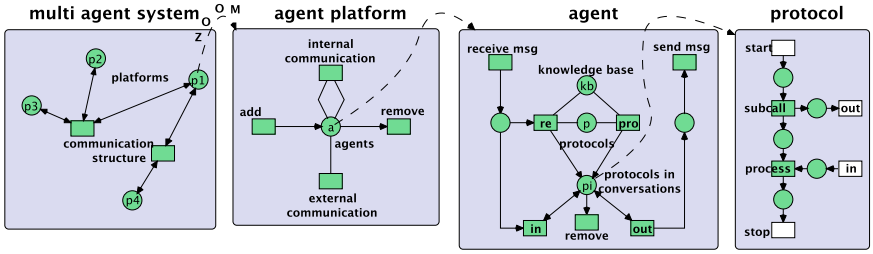

Figure 1: MULAN overview; from [12] / [2]

ORGAN. While [7] follows a similar idea and searches for more abstract entities than agents, Hewitt does not discuss a general system reference architecture. Like MULAN, ORGAN has four levels in which organizations are embedded in markets/fields and consist of smaller entities (MAS). In this paper, we will go in the other direction. Since agents and especially organizations offer a high degree of abstraction and implement good general architectures, efficient system design gets more difficult.

All above ideas and architectures are based on a homogeneous perspective that is motivated by the UNIT-THEORY [16,15,10,6], which makes it possible to map all concepts to infinite condition / event Petri nets (which in principle are Turing-complete). The UNIT-THEORY comes into play again as a mental concept in the modeling of IoT architectures. In [13] the unit theory ([10] building on earlier work (see [16,1,15])) is applied to propose some generic units / entities that are completed by internal entities. The purpose of these entities is to provide an interface to the entities.

3 Proposal for Modeling Principle

While ORGAN introduced a more abstract kind of architecture for MULAN, we now introduce a more specific one. By doing so, we hope to address more specific needs while having a smooth embedding in our previous conceptual and technical framework. Therefore, we briefly discuss the general ideas of entities and their relationship at different levels of the system/software stack of IoT systems/edge computing.

What is necessary for this approach is that again four levels for this kind of abstraction are defined. Good technical examples can be found in [5]. A brief discussion of this work will be given in Section 4. Here, we introduce the notion of a *Thing* (in the IoT context) as the central abstraction level. A thing needs a different, more straightforward interface than an agent. We do not consider autonomy as a first-order concept of the things. Agents can, e.g., communicate with the speech-act theory and ontologies via messages. In the context of IoT, the things instead communicate indirectly by using and providing services.

Taking the above considerations into account, we end up with our modeling principle called EDGE-MULAN (or for short EDGEN). The name indicates the near relation to MULAN and ORGAN. This is necessary since the arbitrarily high level of nesting of abstraction levels is a central feature of edge computing. Due to the modeling bias / application area of our reference architecture we want to achieve the top level as *Cloud* and not *Edge* or *Fog*. We define the four levels from top to bottom as follows:

- **cloud** is the top-level environment that provides computing and storage resource as well as communication, security, privacy, reliability and infrastructure services
- **rooms** reside in a *cloud* and provide a certain set of local services; also an integration of multiple *rooms* within the *cloud* is allowed. The term room implies a physical embedding. However, considering the room as **context** will allow for a logical arrangement of overlapping modeling.
- **things** reside in *rooms*. They encapsulate the structure, minimal resources, and their possible behavior. Things can provide services and can access the services that are provided by other *things*, *rooms* and the *cloud*. Rules that describe the thing handles certain behavior options and resources that might enable behavior or provide means in general.
- **parts** implement the specific properties of the *things*. The direct usage of rules and resources provided/ held/controlled by the thing might be used via services.

Figure 2 illustrates the four levels and their relations analogously to Figure 1 that illustrates the MULAN architecture. Since we do not want to go into the details of the individual units in this contribution, only some of the entities' basic internal functionalities and references are depicted.

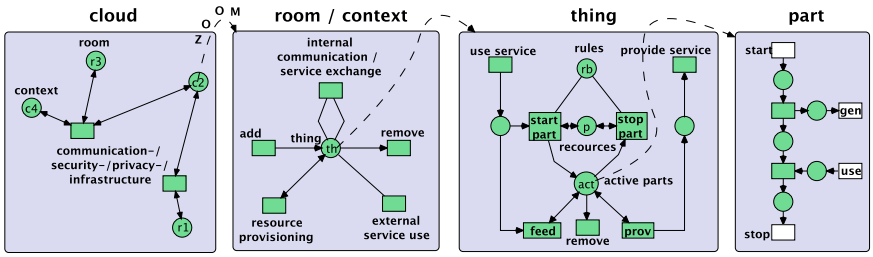

Figure 2: EDGE-MULAN overview

To illustrate the advantages of this modeling principle, we consider the following smart workplace. The building is the *cloud* and consists of several physical spaces. These physical spaces can each be a room, and some physical spaces can even be combined into one digital *room*. Each *room* consists of several *things*

and may provide services through these *things*. Every employee is wearing a transponder. The transponder is a *thing* that is build up by multiple *parts* (e.g. RFID chip, battery, storage, Bluetooth, WiFi). When the employee enters the digital *room* kitchen, the water in the coffee machine may be preheated, or the kettle turned on because the employee prefers to drink tea at this time of day. In this case, the transponder and the coffee machine or the kettle are related because both are digitally located in the same *room*. The *room* interacts with the transponder and asks for the habits of the employee. Next, the *room* interacts with a *thing* (the coffee machine or kettle) to serve the employee.

In the above example, the kitchen (*room*) is the context in which the benefits of edge computing are used. The transponder does not have to communicate with a central data center in the building (*cloud*); instead, it just uses the kitchen to communicate with the named devices (*things*). On the other hand, the kitchen provides the communication infrastructure and thus, the communication between the devices is standardized. The intuitive mapping of real entities and their relations to EDGE-MULAN entities and their relations is one of the main advantages of this modeling principle.

In the individual units of EDGE-MULAN, their structure, internal behavior and relevant IoT concepts such as security, privacy, energy, resources, and communication must be covered. In this contribution, we will not further examine these specific characteristics. The explicit specification of entities, possible relationships, and standardized interfaces facilitates modeling within each level. If these concepts were too abstract, efficiency in designing and specifying such systems would suffer. Modelers could not build the models and systems quickly and well enough. The same applies when sufficient abstract concepts are missing.

4 Related Work

Carrez et al. published a comprehensive investigation and their results from the *Internet of Things – Architecture* project [5]. Among others, they propose a functional-composition for their architecture. Locality principles similar to today's edge computing method are also taken into account. Additionally, they provide well-descriptive and motivating real-world examples. However, they do not consider a clear modeling strategy for a hierarchical / reference architecture.

In [4] health tracking is addressed. Edge computing is used to implement an optimal environment for such applications. However, safety and privacy, as well as functionality, are relevant topics that need to be addressed besides the technical efficiency. For these topics, modeling proposals are missing so far.

Health systems are good examples of the challenges posed by new opportunities in the development of distributed applications. For a user and its environment (containing, for example, doctors, family, pharmacies, medical institutions, insurances, technical equipment, medical devices and supporting machines, legal or social requirements, security, and privacy) several different contexts exist. There are often inconsistencies in the requirements for each context. This type of context separation and the separation of a user's interests/roles must be ad-

dressed both by user support systems and by the user's environment systems. EDGE-MULAN provides such means in a generic form. We cover concepts such as logical and physical entities and their relationship, autonomy, encapsulation, concurrency, (non-)determinism, conflict, synchronization, abstraction.

5 Conclusion

In this contribution, we presented our ideas on supporting modelers for edge computing architectures with an adequate modeling principle to allow an optimal application/IT-alignment. These ideas lead to the proposal of a four-level reference architecture called EDGE-MULAN that contains a *cloud*, *platforms*, *things* and *parts* from top to bottom and their relations and interactions. This architecture supports modelers in their mental challenges in modeling well-structured and efficient IoT architectures. It is based on Reference Nets, which can hold references to other nets, allowing any desired hierarchical structuring. Each entity contains its structure and internal behavior, which is not fully covered by our proposed architecture and often depends on the specific context.

However, without powerful tool support, such an architecture will not be applicable in practice. Suitable modeling constructs must be provided for designing and specifying the properties and relationships of individual entities. The modelers must be empowered to use such concepts intuitively, easily, directly and efficiently. We will consider complex social systems (like large worldwide distributed teams and organizations) with complex processes and numerous devices and subsystems to test our proposal. How to ensure privacy and security in heterogeneous systems is of particular interest. Future work should be in line with our current efforts of meta-modeling, distributed simulation, expressive modeling concepts/constructs within our Petri net environment and incorporation of leading-edge technology.

References

1. A. Borusan and D. Moldt. A method for developing CIM-systems with coloured Petri nets. In Michel Cotsafis and Francois Vernadat, editors, *Advances in Factories of the Future, CIM and Robotics*, pages 91–100. Elsevier, Amsterdam London New York, 1993.
2. L. Cabac. *Modeling Petri Net-Based Multi-Agent Applications.* Dissertation, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, April 2010.
3. L. Cabac, M. Haustermann, and D. Mosteller. Renew 2.5 - towards a comprehensive integrated development environment for petri net-based applications. In F. Kordon and D. Moldt, editors, *Application and Theory of Petri Nets and Concurrency - 37th International Conference, PETRI NETS 2016, Toruń, Poland, June 19-24, 2016. Proceedings*, volume 9698 of *Lecture Notes in Computer Science*, pages 101–112. Springer-Verlag, 2016.
4. S. Distefano, D. Bruneo, F. Longo, G. Merlino, and A. Puliafito. Personalized health tracking with edge computing technologies. *BioNanoScience*, 7(2):439–441, Jun 2017.

5. F. Carrez et al. Internet of Things - Architecture IoT-A Deliverable D1.5 - Final Architectural Reference Model for the IoT v3.0. Available at: `https://www.iotforum.org/wp-content/uploads/2014/09/D1.5-20130715-VERYFINAL.pdf`, 2013. Online; accessed 20.09.2018.
6. M. Hewelt. Grundlegende Konstrukte einer einheitentheoretischen Modellierungstechnik. Diploma thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 2010.
7. C. Hewitt. Perfect disruption: The paradigm shift from mental agents to orgs. *IEEE Internet Computing*, 13(1):90–93, 2009.
8. K. Jensen. *Coloured Petri Nets: Volume 1; Basic Concepts, Analysis Methods and Practical Use*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin Heidelberg New York, 1992.
9. O. Kummer. *Referenznetze*. Logos Verlag, Berlin, 2002.
10. D. Moldt. Petrinetze als Denkzeug. In B. Farwer and D. Moldt, editors, *Object Petri Nets, Processes, and Object Calculi*, number FBI-HH-B-265/05 in Report of the Department of Informatics, pages 51–70, Vogt-Kölln Str. 30, D-22527 Hamburg, August 2005. University of Hamburg, Department of Computer Science.
11. L. Northrop. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon, 2006.
12. H. Rölke. *Modellierung von Agenten und Multiagentensystemen – Grundlagen und Anwendungen*, volume 2 of *Agent Technology – Theory and Applications*. Logos Verlag, Berlin, 2004.
13. B. Schleinzer. Flexible und hierarchische Multiagentensysteme – Modellierung und prototypische Erweiterung von Mulan und Capa. Diploma thesis, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, December 2007.
14. V. S. Shah. Multi-agent cognitive architecture-enabled iot applications of mobile edge computing. *Annales des Télécommunications*, 73(7-8):487–497, 2018.
15. V. Tell and D. Moldt. Ein Petrinetzsystem zur Modellierung selbstmodifizierender Petrinetze. In Karsten Schmidt and Christian Stahl, editors, *Proceedings of the 12th Workshop on Algorithms and Tools for Petri Nets (AWPN 05)*, pages 36–41. Humboldt Universität zu Berlin, Fachbereich Informatik, 2005.
16. R. Valk. Modelling of task flow in systems of functional units. Technical Report FBI-HH-B-124/87, University of Hamburg, Department of Computer Science, Vogt-Kölln Str. 30, D-22527 Hamburg, 1987.
17. M. Wester-Ebbinghaus. *Von Multiagentensystemen zu Multiorganisationssystemen – Modellierung auf Basis von Petrinetzen*. Dissertation, University of Hamburg, Department of Informatics, Vogt-Kölln Str. 30, D-22527 Hamburg, 12 2010.
18. M. Wester-Ebbinghaus and D. Moldt. Auf dem Weg zu organisationsorientierter Softwareentwicklung. Available at: http://www.informatik.uni-hamburg.de/TGI/publikationen/public/data/2006/Wester+06/Wester+06.pdf, September 2006.

Investigating the Java Spring Framework to Simulate Reference Nets with Renew

Jan Henrik Röwekamp

University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences,
Department of Informatics, http://www.informatik.uni-hamburg.de/TGI/

Abstract Renew as a simulator for reference nets does not offer a native web based communication technology. To bring it towards running a single distributed simulation implementing modern communication means is a crucial step. The Java Spring Framework offers interesting capabilities to achieve this goal. However, the implementation is not straightforward and therefore advantages and disadvantages are investigated in this paper. Concluding the advantages outweigh the disadvantages.

Keywords: Petri Nets, Reference Nets, Distributed Simulation, Synchronous Channels, Spring, Renew

1 Introduction

Reference nets have been introduced by Kummer around the year 2000[11]. They are a high-level Petri net formalism based on the Java Programming language[1] and colored Petri nets. One crucial part of reference nets is the differentiation between *net* and *net instance*. Similar to a class in object oriented programming a net only fixes structural aspects and initial markings, but is never simulated itself. Using this analogy a net instance correlates to an object, in a way, that arbitrary amounts of net instances may be generated from one net *during* the simulation. Reference nets follow the nets within nets paradigm [18] and realize it by enabling to hold references to other net instances as token value. Net instance reference tokens may be used to create new or destroy net instances and also to interact with them. As for communication the mechanism of synchronous channels are used [5,10].

In figure 1 an example of a simulation of two reference nets can be found. Upon start of the simulation a net instance of NetA is created. The net instance of NetA then proceeds to create a net instance of NetB and passes the string "Hello!" over the synchronous channel "ch". The single transition of NetB may only fire if it can synchronize with another transition to bind a value to parameter *msg*. After all transitions have fired, the bottom most place in the sole instance of NetA holds a reference to the net instance of NetB created earlier, while the bottom most place in NetB holds the string value "Hello!".

[1] http://www.java.com/

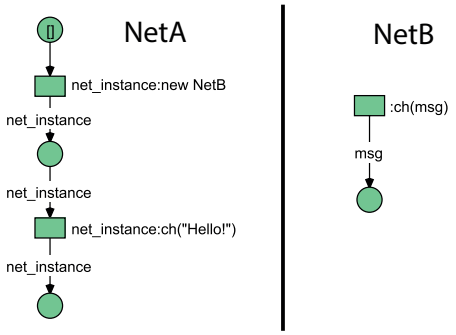

Figure 1: `NetA` example and `NetB` example with synchronous channel `ch`

A simulator capable of simulating reference nets is the reference net workshop (Renew)[2] [3] maintained by our research group at the University of Hamburg. Renew is based on Java and essentially allows for reference nets to utilize arbitrary Java code as transition inscription.

2 Motivation, Vision and Goals

As the number of simulated *net instances* is essentially unknown prior to execution of the simulation, and also Renew in its current form is optimized for single machine execution, the question arises to utilize multiple machines for very large simulations. The central topic this paper is to be filed under therefore is *scalability* of the Renew simulator. The vision behind the research and the ultimate goal is to be able to dynamically launch additional simulation parts upon demand, that are communicating with each other using synchronous channels. As this is a vision it is most definitely not addressed in total here, but given to draw the big picture. This paper is related to but in itself independent from earlier publications [14] [13].

3 Related Work

Early steps towards distribution have already been taken early on by using Java network code in transition inscriptions. While this basically allows communication between multiple simulations, it essentially breaks the formalism and the true concurrency semantics of Petri nets. Communication should be done using synchronous channels. Several further early approaches were made and published in respective papers, e.g., [7,17,9,4,8,12].

[1] utilizes virtual machines in cloud environments. However the usage is limited to single execution runs and the solution does not distribute the entire

[2] http://www.renew.de/

model itself. The simulations in virtual machines are run in isolation and are not part of one big simulation.

An approach by [2] used web technologies, but only for external communication. The core simulation by itself is not altered.

A first attempt in really distributing the simulation in itself was undertaken by Simon [15,16] . The method rewrote some parts of the Renew core simulation to use Java RMI. However, this method was limited in regard to binding search for activation checks of transitions. Also, setting up Java RMI can be tedious work, when connecting new instances dynamically.

4 Implementing a RESTful Renew Core

Looking at recent developments in distributed systems a trend towards RESTful webservices [6] based on HTTP is apparent. RESTful services are stateless and usually also offer interfaces, that implement idempotency. To implement a REST service in Java there are several options, like implementing it directly using a Tomcat webserver or frameworks. One of the most popular frameworks is the Spring Framework[3] by Pivotal. Spring is in essence a dependency injection and inversion of control framework useful for enterprise level development. Coming with a built-in webserver and using annotations it also features extremely accelerated development of RESTful webservices. Spring is therefore a very strong candidate to be included in Renew to achieve flexibility in regards to dynamically launching simulation parts.

This paper addresses thoughts about what needs to be changed within the Renew software to incorporate the Java Spring Framework. Combining Spring with our previous research will enable us to bundle Renew/Spring instances using containerization like Docker[4] to deploy multiple instances dynamically. The way this will be achieved, will be covered in a later paper and is out of scope here.

5 Spring in Renew

For the remainder of this paper it will be discussed how REST/Spring may be integrated into Renew. A general idea of the change can be found within figure 2. On the left side the current implementation is depicted. Several plugins access the simulation core. (Not every plugin needs to interact necessarily with the simulation core itself) On the right side the proposed new method can be seen. The core has been splitted into independent parts, which are tied together by Spring. Plugins are now interacting with the Spring layer instead of the core directly.

[3] https://spring.io/
[4] http://www.docker.com/

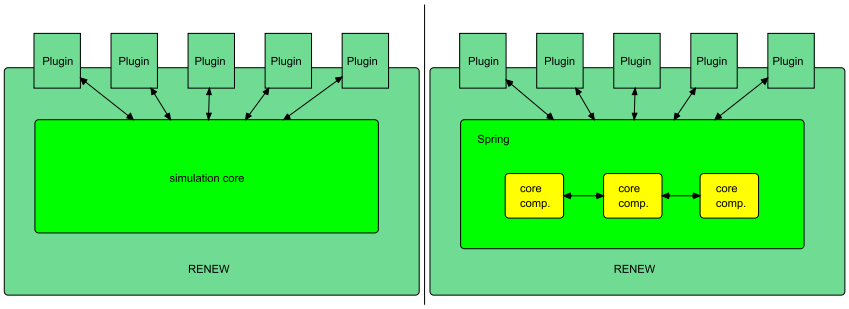

Figure 2: `System net` example and `Object net` example with synchronous channel `ch`

5.1 Data Transfer

First of all using Spring the data transfer is usually realized with JSON[5]. Therefore only key value tuples may be passed limiting the use of objects to serializable ones. Also, to use custom Java objects a converter for simple objects need to be implemented or the programmer needs to implement a JSON-serialization. This requires synchronous channels to utilize value semantics or more sophisticated techniques like global unique object identifiers. Transferring nets by value is possible, as Renew can serialize nets into .rnw files. Transferring net instances may require more thought as they might hold complex tokens, that possibly reference objects, which are not inherently serializable.

5.2 Compatibility With Existing Renew Plugins

Spring should also be realized as Renew plugin, so the base simulator could still be used without this addition. Also, compatibility to the present plugin ecosystem would be highly desirable to make the Spring based simulation core available to previous research based on the original core. This could be achieved by implementing existing interfaces, but certainly constrains the development of the Spring based core.

In general such a plugin would change behaviour of the underlying simulation core similar to the Distribution plugin by Simon [15,16]. It should be considered to define a special type of plugin like "core plugin" for plugins changing base functionality and by that possibly breaking other plugins.

5.3 Compatibility With Other Services

RESTful webservices are widespread throughout organizations, companies and the internet. Using an underlying framework capable of the same protocol would enable far easier communication and integration of foreign services and by that easing the further research and development of the Renew software.

[5] Java Script Object Notation

5.4 Architecture

A typical Spring application listens to events and fires actions upon events, which correlates pretty well with the simulation of net instances, especially distributed ones, that listen for incoming messages. Spring also synergizes well with a microservice architecture, so the simulation core might be broken down into separate parts even more.

Further, the dependency injection functionality of spring can be utilized to dynamically load nets and net instances of these.

5.5 Development Activity

Spring is very active in terms of development and new emerging technologies and protocols are likely to be implemented into Spring by the community. Renew could take advantage of this fact to stay up to date more easily. As an example Spring also supports the relatively new GraphQL[6], which was initially released in 2015. GraphQL is in direct competition with REST and might likely replace it in the future.

5.6 Drawbacks

On the downside, Spring is a very large framework, increasing the difficulty for further contributors to the Renew project. Another negative aspect is the dependency on another framework, that requires maintenance and possibly security relevant updates to be implemented.

5.7 Challenges

The main challenge in implementing Spring into the simulation core is to keep the proven algorithms for binding search, firing and unification intact. Distributing these algorithms has been discussed by Kummer back in 2002[11] and a first attempt at Java RMI based distribution was undertaken by [16] in 2016. Similar problems are bound to arise while implementing a Spring based version of the simulation core.

6 Conclusion and Outlook

The idea of incorporating the Java Spring framework into the simulation core of the reference net simulator software Renew has been presented. Spring comes with a great amount of possibilities for Renew as the drawback of increased complexity of the application itself. Transferring the simulation core into a Spring based version enables compatibility to other REST based webservices and possibly yields a far more scalable architecture.

Further research will include a first prototype of a Spring based Renew core, as well as considerations about a possible microservice architecture of Renew.

[6] https://graphql.org/

References

1. Bendoukha, S.: Multi-Agent Approach for Managing Workflows in an Inter-Cloud Environment. Dissertation, University of Hamburg, MIN Faculty (2016)
2. Betz, T., Cabac, L., Duvigneau, M., Wagner, T., Wester-Ebbinghaus, M.: Software engineering with Petri nets: a Web service and agent perspective. Transactions on Petri Nets and Other Models of Concurrency IX (ToPNoC) pp. 41–61 (Dec 2014)
3. Cabac, L., Haustermann, M., Mosteller, D.: Renew 2.5 - towards a comprehensive integrated development environment for petri net-based applications. In: Kordon, F., Moldt, D. (eds.) PETRI NETS 2016, Toruń, Poland, June 19-24, 2016. Proceedings. LNCS, vol. 9698, pp. 101–112. Springer (2016)
4. Chiola, G., Ferscha, A.: Distributed simulation of Petri nets. IEEE Parallel Distrib. Technol. **1**(3), 33–50 (Aug 1993)
5. Christensen, S., Hansen, N.: Coloured Petri nets extended with channels for synchronous communication. In: Valette, R. (ed.) ICATPN. LNCS, vol. 815, pp. 159–178. Springer (1994)
6. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000), aAI9980887
7. Hauschildt, D.: A Petri net implementation. Fachbereichsmitteilung FBI-HH-M-145/87, University of Hamburg, Department of Computer Science (1987)
8. Kaim, W.E., Kordon, F.: An integrated framework for rapid system prototyping and automatic code distribution. In: Proceedings of RSP, Grenoble, France. pp. 52–61. IEEE (1994)
9. Kordon, F.: Prototypage de systèmes parallèles à partir de réseaux de Petri colorés, application au langage Ada dans un environment centralisé ou réparti. Dissertation, Université P & M Curie (May 1992)
10. Kummer, O.: Simulating synchronous channels and net instances. In: Desel, J., Kemper, P., Kindler, E., Oberweis, A. (eds.) 5. Workshop AWPN. pp. 73–78. No. Forschungsbericht Nr. 694, Fachbereich Informatik, Universität Dortmund (1998)
11. Kummer, O.: Referenznetze. Logos Verlag, Berlin (2002)
12. Pommereau, F., de la Houssaye, J.: Faster simulation of (coloured) petri nets using parallel computing. In: van der Aalst, W.M.P., Best, E. (eds.) PETRI NETS. LNCS, vol. 10258, pp. 37–56. Springer (2017)
13. Röwekamp, J.H., Moldt, D., Simon, M.: A simple prototype of distributed execution of reference nets based on virtual machines. In: Proceedings of the Algorithms and Tools for Petri Nets (AWPN) Workshop 2017. pp. 51–57 (Oct 2017)
14. Röwekamp, J.H., Moldt, D., Feldmann, M.: Investigation of containerizing distributed Petri net simulations. In: Moldt, D., Kindler, E., Rölke, H. (eds.) Applications and Theory of Petri Nets and Concurrency. CEUR Workshop Proceedings, vol. 2138, pp. 133–142. CEUR-WS.org (2018), `http://ceur-ws.org/Vol-2138/`
15. Simon, M.: Concept and Implementation of Distributed Simulations in Renew. Bsc thesis, University of Hamburg, Department of Informatics (2014)
16. Simon, M., Moldt, D.: Extending Renew's algorithms for distributed simulation. In: Cabac, L., Kristensen, L.M., Rölke, H. (eds.) PNSE'16. CEUR Workshop Proceedings, vol. 1591, pp. 173–192. CEUR-WS.org (2016)
17. Taubner, D.: On the implementation of Petri nets. In: Rozenberg, G. (ed.) Advances in Petri Nets 1988. LNCS, vol. 340, pp. 418–439. Springer (1988)
18. Valk, R.: Petri nets as token objects - an introduction to elementary object nets. In: Desel, J., Silva, M. (eds.) 19th International Conference on Application and Theory of Petri nets, Lisbon, Portugal. pp. 1–25. No. 1420 in LNCS, Springer, Berlin Heidelberg New York (1998)

Part II

Extended Abstracts

Detecting Noise in Event Logs using Statistical Inference

Johannes Metzger and Robert Lorenz

Department of Computer Science
University of Augsburg, Germany
robert.lorenz@informatik.uni-augsburg.de
johannes.metzger@informatik.uni-augsburg.de

In [1], Dumas advocates event structures, and in particular labelled prime event structures (LPES), as the unifying representation of process models and event logs in the context of process mining. In [2], a new technique for the representation of the non-sequential behavior of concurrent systems based on LPES is presented. This technique extends LPES by a list of cutoff events in order to finitely represent inifinite behavior. The resulting model is the most compact and the most expressive existing representation of the non-sequential behavior of concurrent systems. In particular, it may represent the behavior of arbitrary bounded place/transition Petri nets (p/t-nets).

Based on these results, we are developing a synthesis-based two-step approach for process discovery in the area of process mining:

- First step (preprocessing): Construct an LPES with cutoff-list from an event log through detecting noise, causality, concurrency and loops.
- Second step (synthesis): Synthesize a Petri net from the result of the first step.

Since LPES with cutoff-lists are the most general existing model representing the behavior of concurrent systems, in the first step, causal structures can be detected with less loss of information than with approaches using other models, as for example Petri net based models. That means, LPES with cutoff-lists have a lower representational bias in the context of process mining compared to other models.

Concerning the second step, in [2], the authors present a theory for the token flow region based synthesis of bounded p/t-nets from LPES with cutoff-lists. The unfolding of the synthesized p/t-net includes the given LPES and has minimal additional behavior. In [3], a practical implementation is provided together with first experimental results.

An important aspect of the first step is the detection of noise. We present a new approach for the detection of noise using statistical methods. The main advantage of this new approach is the obtained statistical foundation of the results including an upper bound for the risk of false classification of traces (as noise or not as noise) and flexible adaptability of the used probability for the occurrence of noise.

Our approach is based on counting the number of occurrences of a pair of direct neighbors ab within the traces of the event log (frequency of ab). Roughly speaking, we identify such an occurrence as noise if the number of occurrences is "very low" compared to the number of all pairs xy with predecessor $x = a$ or follower $y = b$. For the decision, we use one-sided hypothesis tests based on the binomial distribution for each pair ab. We formulate the following two hypotheses:

- Null hypothesis $H_0 : p \leq p_0$: The occurrence of ab should be considered as noise.

– Alternative hypothesis $H_1 : p > p_0$: The occurrence of ab should not be considered as noise.

The value p_0 is the probability for the occurrence of noise and p is the probability for the occurrence of the pair ab within a trace of the event log. The choice of p_0 depends on the considered event log and field of application. A higher value of p_0 leads to the classification of more pairs xy as noise.

The aim is to restrict the risk of falsely inferring that H_1 is true when indeed H_0 is (error type 1), by fixing an upper bound α. That means, we determine the smallest value k, such that

$$p(\text{frequency of } ab \geq k \mid p \leq p_0) \leq \alpha,$$

and decide for H_1 if the frequency of ab is greater or equal to k.[1]

If a trace contains a pair ab which is classified as noise, we omit the whole trace, i.e. we delete the trace from the event log and do not consider it for the construction of the LPES.

Note that not all kinds of noise can be detected through considering frequencies of direct neighbors. The classification and treatment of such kinds of noise is a direction of future research.

Further steps in the construction of an LPES from an event log are the detection of causality, concurrency and loops. A simple possibility for the detection of causality and concurrency is to proceed as in the α-algorithm and its improvements using information about direct neighbors from the noise-free event log.

The detection of loops is a direction of future research, since existing techniques cannot be adapted to LPES with a list of cutoff events.

References

1. M. Dumas and L. García-Bañuelos. Process mining reloaded: Event structures as a unified representation of process models and event logs. In R. Devillers and A. Valmari, editors, *Application and Theory of Petri Nets and Concurrency*, volume 9115 of *Lecture Notes in Computer Science*, pages 33–48. Springer International Publishing, 2015.
2. G. Juhás and R. Lorenz. Synthesis of bounded Petri Nets from Prime Event Structures with Cutting Context. In W. M. P. van der Aalst, R. Bergenthum, and J. Carmona, editors, *Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data 2016 Satellite event of the conferences: 37th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2016 and 16th International Conference on Application of Concurrency to System Design ACSD 2016, Torun, Poland, June 20-21, 2016.*, volume 1592 of *CEUR Workshop Proceedings*, pages 58–77. CEUR-WS.org, 2016.
3. R. Lorenz, J. Metzger, and L. Sorokin. Synthesis of bounded petri nets from prime event structures with cutting context using wrong continuations. In *Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data 2017 Satellite event of the conferences: 38th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2017 and 17th International Conference on Application of Concurrency to System Design ACSD 2017, Zaragoza, Spain, June 26-27, 2017.*, pages 21–38, 2017.

[1] In this setting, we control the false classification of noise by α. It is also possible to exchange H_0 and H_1: then the false classification of "normal behavior" is controlled by α.

A Browser Based Tool for Workflow Guided Tutorials

Joachim Schmidberger, Matthias Feldmann, and Daniel Moldt

University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences,
Department of Informatics, http://www.informatik.uni-hamburg.de/TGI/

Overview For the support of our teaching projects Web-based support of the learning process is needed. Here we discuss the development of our browser-based tool for workflow guided tutorials with JointJS and Vue.js frameworks in Javascript and HTML templates.
Keywords: Petri nets, workflow nets, education, tutorials, Javascript

Motivation As described in [1] we support our teaching project workflows with detailed process descriptions during the first weeks. To support automation of these processes we strived for a browser-based tool support to allow for an even better teaching process. So we were looking for the best solution.

Problem For the description of the processes, we use a kind of workflow nets. A natural implementation would be to use Renew as the execution engine. However, we experienced that students expect access to such a tool from anywhere with their usual look and feel. A browser-based alternative had, therefore, to be discussed. Former solutions were outdated, as can be seen in our browser-based implementation in the project Sisol[1] from 2005.

Approach Since the workflows were created with the help of the tool Renew[2], they were available in the tool's rnw file format. Import and export functions to the XML based file format PNML already exist in Renew. The planned Web application needs to read, parse, display and simulate these files. Since the PNML exporter does not provide all formatting information (e.g. arc waypoints), the application should also contain an editor for the loaded nets. The desired process using the application is to step through the workflow by clicking on task transitions, starting from the initial marking of the net. Meanwhile, the application displays the corresponding exercise texts for the activated tasks.

A prototype was created using HTML and Javascript. In the first step, various graphics frameworks for Javascript were tested for their fitness. There are multiple ways to create a web-based application. A framework provides much help getting started since templates can be used to output values from JavaScript to the HTML document. Frameworks also provide a predefined structure and

[1] http://www.informatik.uni-hamburg.de/TGI/forschung/projekte/sisol/tutorial_big.gif
[2] Renew homepage: http://www.renew.de

already implement a structure for inter-component communication. In the fast-moving world of Javascript development, one of the first experiences was that active development and maintenance of the frameworks had to be one decisive selection criteria.

The first prototype developed with Draw2d Touch[3] as a graphical framework was no longer executable after the switch to the next browser generation. At the time of writing this text, the further development of the framework was discontinued. JointJS[4] and mxGraph[5] were shortlisted because they have an active developer community. mxGraph offers all the classic editing features, while JointJS provides better support for simulations. The current prototype is based on JointJS, which already offers a simple Petri net plugin.

Vue.js[6] was chosen as the basic framework since it is lightweight and uses native Javascript without syntax extensions. This combined with the large, active community makes it easy to learn the framework. Additionally, Vue.js is used by large companies like Alibaba, Baidu, and Tencent which increases the probability that it will be further developed in the future.

Results The current prototype already supports the functionality to read and display PNML files by converting these into JSON Objects which are then rendered with JointJS. Basic editing and simulation functionality is available. At this point, it is evident that the tools currently chosen are well suited for the intended tutorial support. Reading through the tasks on the tutorials can now be simulated in corresponding workflow nets.

Open problems and outlook For the next prototypes, the editing functions have to be extended, and the persistence of the tutorials has to be ensured. The synchronization between workflow simulation and task display still has to be implemented. The project could then prove its usefulness in lessons by using the application instead of the worksheets. The workflows also show the progress the students have made in the exercise. In future works, this could be used to analyze the progress and optimize the exercises. There could be a version of the tool for the teacher that displays the current status of the learners in different colored tokens. A connection of the tool to Redmine to open task related tickets could also be a useful addition.

References

1. D. Schmitz, D. Moldt, L. Cabac, D. Mosteller, and Haustermann M. Utilizing Petri Nets for Teaching in Practical Courses on Collaborative Software Engineering. In *ACSD 2016, Toruń, Poland, June 19-24, 2016*, pages 74–83. IEEE Computer Society, 2016.

[3] Draw2d homepage: `http://www.draw2d.org/draw2d/`
[4] Jointjs Homepage: `https://www.jointjs.com/`
[5] Mxgraph homepage: `https://github.com/jgraph/mxgraph`
[6] Vue.js Homepage: `www.vuejs.org`

About the Development of a Curry-Coloured Petri Net Simulator

Michael Simon and Daniel Moldt

University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences,
Department of Informatics, http://www.informatik.uni-hamburg.de/TGI/

Overview / Motivation Several inscription languages for Petri net simulators exist to extend the expressibility and usability of Petri nets in system development. Renew uses Java and attempts to add a functional language (Scheme) were successful with respect to language, however, the performance was poor and the language (and its implementation) do not allow for side effect free development. CPN-Tools uses Standard ML as the inscription language. This tool is very widely used, however, ML has side effects. A former tool inscribed with Haskell had only poor usability.

Avoiding side effects has the underlying promise that analysis of such nets might be easier. Therefore, a new attempt was made to develop a Petri net simulator that incorporates a language without side effects. After some investigations about the best options the purely functional logic programming language Curry was chosen. Haskell as the implementation language was therefore a natural choice. In the following, we briefly sketch the development and the main features of the newly introduced Petri net formalism.

Approach Following the Paose-Approach (Petri net-based, Agent- and Organization-Oriented Software Engineering) the most critical aspects were addressed in a stepwise approach. Based on our experiences with Renew and Java as the implementation language and a thorough investigation about the state-of-the-art of similar simulators several features were identified that should be covered in the new simulator. The complete work and the development process are described in the master thesis of the first author.

As in Renew the concurrency and the practical usability of the simulator was a of high priority. Since Coloured Petri nets (CPN) by Kurt Jensen is the basic formalism the transition binding was of central interest. To avoid the development of a whole graphical interface just for CCPN Renew was used for the graphical user interface (gui). This required some modification within the Renew framework what was also done in context of the master thesis as a smaller portion of the workload. Several milestones were set up to reach a fully usable simulator. In a systematic way the key features of the tool were developed.

Results A graphical user interface was made available via a new Renew plugin. Editing, inspection, observation and control of CCPN nets and simulations was implemented via an XML interface. As the first step for the provision of analysis tools the reachability graph can be generated under the gui control.

CCPN can be used as a software library by other Haskell programs. Several interfaces are available to control the simulation. Access to the net markings and the firing of specific transition sequences are made available.

The Petri net formalism allows the inscription of the nets with Curry code. Concurrent Haskell implementation allows for the execution in an efficient manner. Most important is that transition execution happens without side effects. The strictly strongly typed programming language environment allows to ensure this together with the software design of the binding and firing rules implemented for CCPN. The evaluation of functions does not have side effects and the logic program evaluation for the transition binding search are of central importance.

Overall, CCPN provides a new powerful Petri net formalism based on a logic programing language, what is the central idea of the master thesis. The starting time of a simulation is relatively high due to the prototypical implementation. Emphasize was laid on the usability study of the different features within the simulation. All features could be implemented in a very efficient way. The search for transition bindings to fire is delegated to the underlying Curry compiler. It is central to the simulation algorithm which is inspired by Renew and implemented in Haskell. Doing so, the upcoming developments within the Haskell and Curry language development can directly improve the CCPN simulator. Therefore, important parts heavily rely on the KiCS2 Curry compiler[1].

Modelers can define purely functional logic programs for initial place markings and transition inscriptions. Two main libraries allow, via the Curry compiler and its libraries, for dynamically compiled and loaded nets and software. The simulator core is agnostic to the inscription language and does not depend on the KiCS2 Curry compiler integration. Other CPN formalisms are therefore possible in the future. Three interface provide sufficient means and different levels to use and control of CCPN. Embedding into the Renew environment via XML allows a convenient usage.

Open problems and outlook Optimization of the compilation can be acquired by the pre-compilation of time intensive parts. Usage of compilation results should reduce the setup time for a simulation considerably. For this some strategies have to be designed for our domain and the intended use of CCPN. Once everything is compiled the simulation is executed fast. User experiences are therefore much better during the simulation runs. CCPN is based on the current 0.6.0 stable version of the KiCS2 compiler. KiCS2's current development release adds type classes similar to Haskell, a major type system improvement. Porting CCPN is necessary to use this new feature and future KiCS2 improvements.

Most interesting is the extension of the CCPN formalism itself. Test arcs or nets-within-nets or other hierarchy concepts are highly interesting. Due to the concurrent implementation and the functional language background distributed simulation is of high interest. Most importantly the options of analysis of the net formalism is very promising.

[1] see `http://www-ps.informatik.uni-kiel.de/kics2/`

Improvement of the Renew Editor User Interface

Martin Wincierz, Daniel Moldt, and Michael Haustermann

University of Hamburg, Faculty of Mathematics, Informatics and Natural Sciences,
Department of Informatics, http://www.informatik.uni-hamburg.de/TGI/

Overview / Motivation For most of it's existence, the user interface of Renew has used the same graphical framework and window management. For several usage scenarios this was not the state-of-the-art anymore. Here we discuss some improvements for a better user experience of Renew.

Problem The original graphical interface was developed on the basis of the framework JHotDraw[1] by Erich Gamma. There was no architectural design for a plugin system, when Renew branched from JHotDraw. In the master thesis of the first author[2] two main goals were addressed: 1) (Maintainability) Enable the graphical interface to be more easily extendable via plugins. 2) (Usability) Improve the general usability of Renew.

Approach Former attempts to improve the Renew GUI were used to check options. Other tools that are used in a similar manner as Renew (e.g. other Petri Net editors such as CPN-Tools or graphical editors such as GIMP) were analyzed for useful features and design patterns. Furthermore experiences from several smaller development studies, student thesis and teaching projects were collected and checked for the most pressing needs of improvement.

A list of improvements were set up and systematically tested for possible realizations. It is important to notice that the two main goals required different levels of investigation. For the plugin extensibility the internal design of Renew had to be improved. Some technical debts had to taken into account and parts of the software had to be re-engineered. For the usability, alternative designs of the interface and features of the software had to investigated.

Based on these thorough investigations several decisions had to be made about what could be done within the given timeframe of the master thesis of Wincierz. For the methodological setting the software development Paose-approach[3] (Petri net-based, Agent- and Organization-Oriented Software Engineering) was applied: involvement of users (at the architectural and the application level respectively). deep analysis of the given system; definition of clear sprint results; systematic prototyping with state-of-the-art agile methods; separation of concerns by dividing problems into manageable parts.

[1] JHotDraw: http://www.jhotdraw.org

[2] Wincierz, Martin: Verbesserung der Erweiterbarkeit und Benutzbarkeit der grafischen Oberfläche des Petrinetz Simulators Renew, University of Hamburg, 2018

[3] Paose: http://paose.de

Results Improving the maintainability required the main efforts during this development process. Integrating the Docking Frames framework[4] allowed shifting several functionality to an external library. Workload of maintaining the code within the Renew development team now changes. The framework will need to be monitored and continuously be integrated into our code base. Future development will, however, be much easier since extensibility is improved due to the better support of plugin architectures by the framework.

Via the usage of several design patterns (bridge, adapter, factory, observer, etc.) the maintainability of the Renew code base could be enhanced considerably. Reimplementation of the `DrawApplication` class improved the readability of the corresponding code base. Many code comments were added to the inspected code (improvement of readability) and new code was created directly with comments (ensuring readability). Java assertions were partially applied to allow for validation of interface conditions. Overall the maintainability of the Renew code base was improved considerably. At the same time new features could and can be added in an easier way.

Usability improvements resulted directly from the usage of the new graphical framework. The look-and-feel with respect to the window management follows now more modern styles. Zooming was implemented and hence directly supports task adequacy. Context-sensitive information can now be acquired more easily by new plugin-based tools. Control of the program was shifted from a purely menu-based system to the usage of buttons with graphical icons which clearly communicate their meaning. This facilitates easier learning of the application's features and their faster control by expert users. Tool bars can now be adapted via individual configurations and at runtime the window management is more flexible with dock-able windows. Overall the usability has been improved in many facets without losing former features.

Open problems and outlook While several aspects could be improved, multiple up to now hidden problems were revealed. Future work can be done by improvement of the menu management. Since future plugins need to register their own new menus, general conventions regarding naming and menu usage as well as more automation in ordering these menus would be of help. I/O Operations can be improved. Due to the new framework and the decoupling of representation and I/O operations other kinds of graphics than Renew drawings could be opened and displayed. The aesthetics of the Renew GUI still does not match current expectations of UI design. For acceptance at teaching and practical use the intuitive representation and usage of the functionality has to be taken into account. Newest trends in usability have to be obeyed to keep the motivation high. Due to the size of Renew and its whole eco system of tools and frameworks a complete analysis of the user interface has not been done. Further improvements are possible in many parts.

[4] `http://www.Docking-Frames.org`