Analysing Product Lines of Concurrent Systems with Coloured Petri Nets

Elena Gómez-Martínez Esther Guerra Juan de Lara
Universidad Autónoma de Madrid, Madrid, Spain
{MariaElena.Gomez, Esther.Guerra, Juan.DeLara}@uam.es

Abstract

Petri nets are a popular formalism to model and analyse concurrent systems. They can be combined with software product lines to support the specification of concurrent system families, like variants of controllers, process models, or configurations of flexible assembly lines. Specifically, a Petri net product line *(PNPL) comprises a (black and white) Petri net decorated with variability inscriptions, and a feature model controlling the derivation of admissible nets of the family. However, the derivable nets cannot be reconfigured at runtime, and the techniques to analyse properties of such reconfigurations are limited.*

To tackle these issues, we present a method to embed a PNPL into a standard Coloured Petri net. This embedding permits using the extensive simulation and analysis capabilities of powerful tools like CPN Tools, and enables the reconfiguration of the product nets at run-time. In this paper, we report on the translation of PNPLs into Coloured Petri nets, characterize the properties that can be analysed with this translation, and describe tool support on the basis of a case study in the area of flexible production systems.

1 Introduction

Petri nets [1] are a popular formalism to model and analyse concurrent systems. Their graphical nature makes modelling intuitive, and their strong theoretical basis enables powerful analysis possibilities. However, they are limited when the analysis of (possibly large) families of similar systems – like variants of process models [2], robots [3], configurations in flexible assembly lines [4] or reconfigurable manufacturing systems [5] – is required.

To solve this issue, in previous work, we proposed the notion of *Petri net product line* (PNPL) [6] as a compact way to specify families of net variants based on product line techniques. In essence, a PNPL combines a (black and white) Petri net annotated with presence conditions, and a feature model to express the allowed variability. Several Petri net analysis techniques have been lifted to enable the analysis of all nets of the family at once, instead of a case-by-case analysis [6]. However, the analysis is limited to structural properties (like marked graph, state machine and free-choice) and the reconfiguration between variants is not possible at run-time. This hinders the use of PNPLs in applications requiring dynamic reconfigurations, like self-adaptive cyber-physical systems [7, 8].

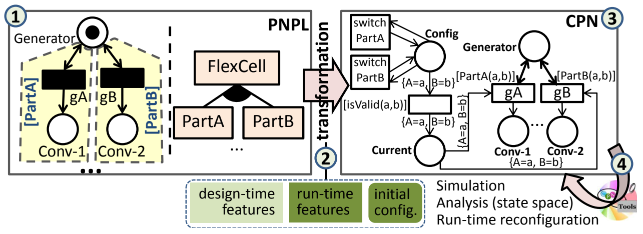


Figure 1. Overview of the approach.

To alleviate these issues, we propose to transform PNPLs into equivalent standard Coloured Petri nets (CPNs), as Fig. 1 shows. CPNs [9] extend Petri nets with data types, so that tokens can carry data, and arcs can query and produce tokens according to specified conditions. Our mapping synthesizes a CPN that explicitly represents the current configuration as a coloured token, emulates the presence conditions on the net elements via suitable arcs, and permits reconfiguring the system to a new feature configuration (cf. label 3 in Fig. 1). Prior to synthesizing the CPN (label 2), the user selects the features that may change at run-time, those fixed at design-time, and an initial configuration. Our mapping into CPNs enables the simulation of the running system and opens the door to useful analysis possibilities, e.g., based on model checking (label 4).

We have implemented the described mapping on an Eclipse plugin called TITAN. This tool supports the graphical modelling of PNPLs and the lifted analysis of structural properties. For this work, we have extended TITAN to transform a given PNPL into a CPN that can be simulated and analysed within CPN Tools [10].

2 Background

In this section, we introduce Petri nets (Sec. 2.1), PNPLs (Sec. 2.2) and CPNs (Sec. 2.3) using an example in flexible manufacturing systems.

2.1. Petri Nets

Petri nets [1] are a graphical formal notation to represent concurrent systems. A Petri net is a bipartite graph with two types of nodes: places (graphically depicted as circles) and transitions (drawn as rectangles). Places can be connected to transitions, and vice versa, via arcs. Petri nets have a *marking*, representing the distributed state of the net. The

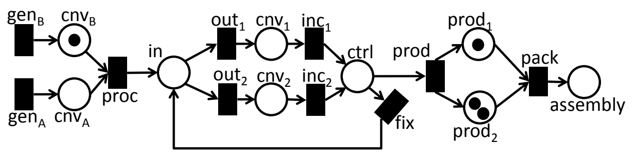


Figure 2. Petri net modelling an assembly line.

marking is given by sets of tokens (depicted as black circles within places) associated to each place.

Fig. 2 shows an example Petri net inspired by [6]. It represents an assembly line. Transitions gen$_A$ and gen$_B$ model generators of parts of types A and B, which transition proc processes and sends to any of two parallel conveyor belts (represented by places cnv$_1$ and cnv$_2$). After a quality control, transition fix sends the defective parts back. In a final step, two machines (represented by transitions prod and pack) process the parts, until they are assembled.

Petri nets can be simulated by the so-called token game. A transition is *enabled* if each input place has at least one token. Enabled transitions may fire at any moment. Firing a transition removes one token from each input place of the transition, and adds a token to each of its output places. In Fig. 2, transitions gen$_A$, gen$_B$ and pack are enabled. Firing pack would add a token to place assembly, and would remove one token from prod$_1$ and prod$_2$.

2.2. Petri Net Product Lines

Petri nets are powerful to represent concurrent systems, but they are less suitable to capture families of similar systems in a compact way. For example, each possible configuration of a flexible assembly line – with different types of input parts, fabrication layouts, and output products – should be represented as a separate net. This is problematic if there are many configurations, the features of the assembly line can change, or properties of the whole family need to be analysed (e.g., can the assembly line manufacture a certain type of part in all configurations?).

PNPLs [6] combine Petri nets with product lines [11, 12] to tackle this problem. A PNPL comprises a feature model (FM) [13] describing the variability space, and a Petri net (called *150% net*) whose elements define presence conditions (PCs). The latter are boolean formulae over features of the feature model. Specific Petri net products can be *derived* from the PNPL by selecting a configuration. This derivation process removes all elements from the 150% net whose PC evaluates to false after substituting the selected configuration features by true, and the rest by false.

Fig. 3 shows an example PNPL with the possible configurations of a flexible assembly line. The feature model in Fig. 3(a) allows selecting one or more kinds of input parts (PartA, PartB), a fabrication layout (optional QualityControl, optional Parallel conveyor), and one or more kinds of output

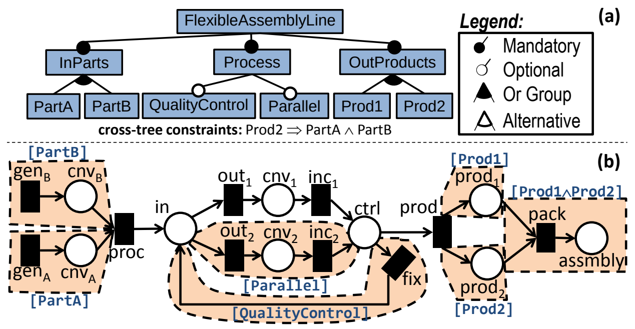


Figure 3. PNPL of a flexible assembly line. (a) Feature model. (b) 150% net.

products (Prod1, Prod2). A constraint forces that both types of input parts are selected to produce Prod2.

The 150% net in Fig. 3(b) has the same underlying net as Fig. 2, but its elements have PCs (shown in square brackets). We use dashed, coloured regions to assign the same PC to several elements. As an example, if a configuration does not select feature PartB, then transition gen$_B$, place cnv$_B$ and their adjacent arcs would be removed from the derived product net.

2.3. Coloured Petri Nets

Coloured Petri nets (CPNs) [9] extend Petri nets by allowing tokens to carry data. The data structure is given by assigning a type (a *colour*) to the places. In CPN Tools, colours are specified with the Standard ML functional language [14], which supports defining datatypes like enumerations, product, union, list, and record types.

Arcs in CPNs are annotated with expressions, which can encapsulate complex computations. They may also include free variables, that need to be bound to suitable values found in the tokens. Transitions can have *guards*, which are boolean expressions that need to evaluate to true for the transition to be enabled. They can be used to test values from the variables bound in input arcs.

3 Analysing PNPLs with CPNs

Next, we present the transformation of PNPLs into CPNs (Sec. 3.1) and the analysis possibilities (Sec. 3.2).

3.1. Transforming PNPLs into CPNs

Our approach to analyse behavioural properties of PNPLs relies on CPNs. The rationale for this transformation is to be able to activate or deactivate the elements in the net structure (places, transitions, arcs) by means of expressions or guards in the CPN, according to the selected feature configuration. To achieve this, we use the feature model (FM) and the 150% net to guide the transformation.

3.1.1 Transforming the feature model.

To translate the feature model into a CPN, we initially generate two colour sets to encode a configuration. We create a generic type called FEATURE (a boolean); and a record colour set CONFIGURATION with n fields of type FEATURE, being n the number of features in the FM. Each FEATURE_i stores whether the field is selected (true) or not (false) for a particular configuration. For the running example of Fig. 3, the created colour sets are the following:

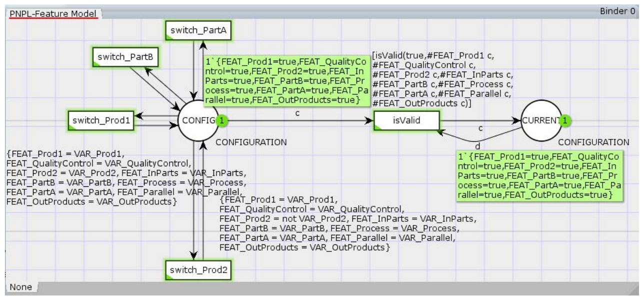
```
colset FEATURE = BOOL;
colset CONFIGURATION = record
  INPARTS : FEATURE * PARTA : FEATURE * PARTB : FEATURE *
  PROCESS : FEATURE * QUALITYCONTROL : FEATURE *
  PARALLEL : FEATURE * OUTPRODUCTS : FEATURE * PROD1 : FEATURE *
  PROD2 : FEATURE;
```

Then, a place CONFIG with one token and the colour set CONFIGURATION is added to the CPN to represent any configuration, valid or not, generated from the FM. Values for tokens are extracted from an initial configuration set beforehand by the user. Moreover, to allow changing the feature selection at runtime, one transition Switch_Feature_i per feature is included in the CPN. This transition is connected with the place CONFIG by two arcs: one input arc that reads the current value of the feature, and one outgoing arc switching its value. To reduce the size of the resulting CPN and consequently, its analysis time, we make the following optimization: if a feature needs to be selected in every valid configuration (i.e., it is mandatory) then we do not generate a transition to switch its value. In our running example, for instance, we do not generate such Switch transitions for features InParts, Process, OutProducts and FlexibleAssemblyLine. Similarly, the corresponding Switch transitions are not generated for the non-dynamic features that are fixed at design time (cf. step 2 in Fig. 1).

For the created transitions, the switched value is collected in a variable of type FEATURE. We also include two variables, named c and d, with type CONFIGURATION to store the current and the previous configuration, respectively. Therefore, the defined variables for the running example of Fig. 3 are the following:

```
var InParts, PartA, PartB, Process, QualityControl,
    Parallel, OutProducts, Prod1, Prod2 : FEATURE;
var c, d : CONFIGURATION;
```

The next step is to validate whether the current selection of features corresponds to a valid configuration. With this purpose, we incorporate a transition called isValid that reads a token from the place CONFIG. This transition has a guard, implemented as a function in the CPN ML language, which encodes the feature model as a propositional formula following the rules proposed in [15]. Therefore, only valid configurations satisfy this guard and enable the transition. Firing the transition passes the token with the variable c containing the value of all features for this valid configuration to the place CURRENT, and removes the old one. Thus,



```
1`{FEAT_Prod1=true,FEAT_QualityControl=true,FEAT_Prod2=true,FEAT_InParts=true,FEAT_PartB=true,FEAT_Process=true,FEAT_PartA=true,FEAT_Parallel=true,FEAT_OutProducts=true}

[isValid(true,#FEAT_Prod1 c,
#FEAT_QualityControl c,
#FEAT_Prod2 c,#FEAT_InParts c,
#FEAT_PartB c,#FEAT_Process c,
#FEAT_PartA c,#FEAT_Parallel c,
#FEAT_OutProducts c)]

{FEAT_Prod1 = VAR_Prod1,
FEAT_QualityControl = VAR_QualityControl,
FEAT_Prod2 = VAR_Prod2, FEAT_InParts = VAR_InParts,
FEAT_PartB = VAR_PartB, FEAT_Process = VAR_Process,
FEAT_PartA = VAR_PartA, FEAT_Parallel = VAR_Parallel,
FEAT_OutProducts = VAR_OutProducts}

{FEAT_Prod1 = VAR_Prod1,
FEAT_QualityControl = VAR_QualityControl,
FEAT_Prod2 = not VAR_Prod2, FEAT_InParts = VAR_InParts,
FEAT_PartB = VAR_PartB, FEAT_Process = VAR_Process,
FEAT_PartA = VAR_PartA, FEAT_Parallel = VAR_Parallel,
FEAT_OutProducts = VAR_OutProducts}

1`{FEAT_Prod1=true,FEAT_QualityControl=true,FEAT_Prod2=true,FEAT_InParts=true,FEAT_PartB=true,FEAT_Process=true,FEAT_PartA=true,FEAT_Parallel=true,FEAT_OutProducts=true}
```

Figure 4. CPN for the feature model of the PNPL (some arc expressions omitted).

this place contains the token with the current configuration which will be used to activate and deactivate elements in the 150% net.

The next listing shows the function implementing the guard to validate configurations for the running example. It uses implication and bidirectional functions that we have defined explicitly, since they do not exist in ML.

```
fun implies (p,q) = not p orelse q;
fun iff (p, q) = ((not p orelse q) andalso (p orelse not q));
fun isValid (PARAM_FlexibleAssemblyLine, PARAM_Prod1,
      PARAM_QualityControl, PARAM_Prod2, PARAM_InParts,
      PARAM_PartB, PARAM_Process, PARAM_PartA, PARAM_Parallel,
      PARAM_OutProducts) =
  iff(PARAM_InParts,PARAM_FlexibleAssemblyLine) andalso
  iff(PARAM_Process,PARAM_FlexibleAssemblyLine) andalso
  iff(PARAM_OutProducts,PARAM_FlexibleAssemblyLine) andalso
  (PARAM_PartA orelse PARAM_PartB) andalso
  (implies(PARAM_QualityControl,PARAM_Process) andalso
    implies(PARAM_Parallel,PARAM_Process)) andalso
  (PARAM_Prod1 orelse PARAM_Prod2) andalso
  implies(PARAM_Prod2,(PARAM_PartA andalso PARAM_PartB));
```

Fig. 4 shows the fragment of the CPN resulting from the feature model depicted in Fig. 3(a), where features QualityControl and Parallel are static and hence there are no Switch transitions for them.

3.1.2 Transforming the 150% net.

The structural elements of the 150% net (places, transitions, arcs) have an almost direct translation into the CPN. The type of all the places is INT and they are marked according to the initial marking of the 150% net. To access the current configuration, every transition of the 150% net is connected by a bidirectional arc to the place CURRENT. The arc expression is the variable c, which stores the current configuration.

We model the PCs by means of arc expressions and transition guards. They allow to activate or deactivate elements of the net, since both determine if a transition is enabled (for a given marking). Specifically, the PCs of transitions are directly transformed into transition guards in the CPN. Regarding places, there is no way to activate or deactivate them in the CPN, hence, we emulate this behaviour by expressions in their input and output arcs. The PCs of arcs are translated into *if-then-else* expressions, where the *if*-condition is the PC of the arc, the *then*-part is a new token to the output place, and the *else*-part is no token. For instance, the arc expression of the arc from transition pack to place assembly of the 150% net in Fig. 3 is:

```
if (#FEAT_Prod1 c andalso #FEAT_Prod2 c) then 1`1 else 0`1
```

where operator ♯ retrieves the field values within records. Overall, the expression adds a token only if the PC ($\mathsf{Prod1} \wedge \mathsf{Prod2}$) evaluates to true on the current configuration (given by c). As noted, PCs are parsed into the field names of the record colour set CONFIGURATION, and logical operators (and and or) are adapted to the ML language (andalso and orelse).

The resulting CPN model can be analysed using CPN Tools [10] in order to study behavioural properties. Fig. 5 shows the CPN obtained from the running example.

3.2. Analysing the PNPL

Most CPNs analyses are based on the *occurrence graph*: a graph-based representation of the state-space of reachable markings [9]. In PNPLs, this state-space represents all possible executions of the net, in all possible configurations allowed by the change of dynamic features. Once PNPLs are transformed into CPN, some of their properties that can be analysed are [10]:

Boundedness. A place is bounded if it can admit a limited number of tokens. CPN Tools reports the minimum and maximum bounds for each place. In PNPLs, the places associated to the feature model are bounded, and the bounds reported for the places in the 150% net are calculated for any possible configuration (no place in any configuration of our PNPL is bounded).

Home markings are those reachable from every reachable marking of the net. They may represent cyclic behaviours that might be desirable in all valid configurations (our PNPL lacks home markings).

Liveness is concerned with transitions staying active. CPN Tools reports dead markings (those where no firing is possible and the execution ends), live transitions (there is a firing sequence containing the transition in any reachable marking), and dead transitions (not enabled in any reachable marking). In PNPLs, the interest is on liveness of transitions of the 150% net, which refers to all possible configurations of dynamic features. Our PNPL lacks dead markings and dead transitions in any configuration.

Model checking allows formulating properties in temporal logics, to be checked on the state-space [16]. In PNPLs, these formulae can combine features of the feature model with properties of the markings in the 150% net. In our running example, we could check if, given a selection of static features, then for every configuration of the dynamic features, a token eventually reaches either prod_1 or prod_2 in every execution (so that products are produced). This is not the case in configurations with QualityControl.

4 Tool Support

We have built the tool TITAN (Tool for Petri net product line analysis) to support our approach. It is an Eclipse plug-in and uses the Eclipse Modeling Framework (EMF) [17] as the underlying modelling technology. It is available on `https://github.com/antoniogarmendia/titan`.

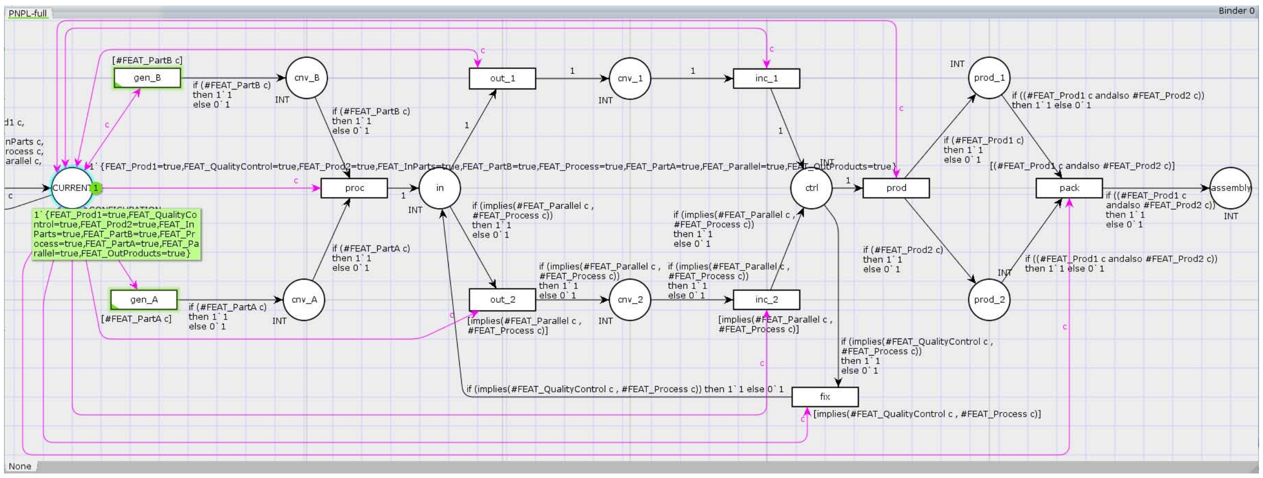
The tool integrates a graphical editor to define the 150% Petri net and its PCs. The editor is based on Sirius [18], a framework to create graphical modelling environments. TITAN also extends FeatureIDE [19] – a widely used plug-in in the field of product line engineering – to specify the feature model, the feature configurations and generate the product nets. TITAN supports the lifted analysis of structural properties of the PNPL, following the approach described in [6]. For this purpose, it relies on two libraries: the Sat4J solver [20] for solving boolean satisfaction problems (used to analyse properties state-machine, marked graph, free-choice and extended free-choice), and the JaCoP Java library as constraint programming (CP) solver [21] (used to analyse P- and T-invariants).

The architecture of TITAN is extensible via extension points with new analysis techniques and exporters to the input format of other Petri net tools. In addition to CPN Tools [10], which includes the presented transformation of PNPLs into CPNs, TITAN has exporters of the 150% net to GreatSPN [22], TimeNET [23] and WoPeD [24].

Fig. 6 shows TITAN with the running example. On the left, the Eclipse explorer contains the FeatureIDE project, which includes the PNPL of the running example, and the generated CPN file with its configuration file. The middle part shows the 150% net and its PCs. The right side displays the feature model. The result of the T-invariant analysis is presented on the bottom.

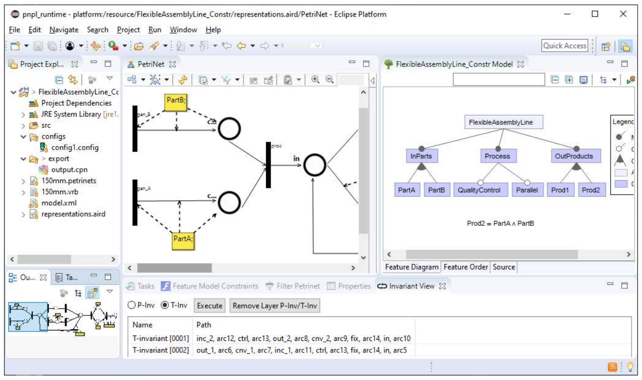
5 Related Work

Our proposal realises the notion of *dynamic software product line* (DSPL) for Petri nets. DSPLs [25] allow controlling the variability of adaptive systems at runtime. A DSPL can be seen as a system where the feature configurations correspond to system adaptations. Some authors analyse different aspects of DSPLs (but not for Petri nets). For instance, Sawyer et al. [26] use constraint solving to find the optimal configuration of self-adaptive systems, Olaechea et al. [27] employ trace checking to analyse the quality of service of all configurations of a DSPL, Göttmann et al. [28] translate DSPLs into timed automata to analyse the worst/best execution time of reconfiguration sequences, Ayala et al. [29] analyse DSPLs to predict the impact of recon-



PNPL-full
Binder 0
[#FEAT_PartB c]
gen_B
if (#FEAT_PartB c) then 1`1 else 0`1
cnv_B
INT
out_1
cnv_1
inc_1
INT
prod_1
if ((#FEAT_Prod1 c andalso #FEAT_Prod2 c)) then 1`1 else 0`1
CURRENT
proc
in
ctrl
prod
pack
assembly
if (#FEAT_Prod1 c) then 1`1 else 0`1
[(#FEAT_Prod1 c andalso #FEAT_Prod2 c)]
if (implies(#FEAT_Parallel c , #FEAT_Process c)) then 1`1 else 0`1
gen_A
[#FEAT_PartA c]
if (#FEAT_PartA c) then 1`1 else 0`1
cnv_A
out_2
cnv_2
inc_2
prod_2
[implies(#FEAT_Parallel c , #FEAT_Process c)]
if (#FEAT_Prod2 c) then 1`1 else 0`1
if ((#FEAT_Prod1 c andalso #FEAT_Prod2 c)) then 1`1 else 0`1
if (implies(#FEAT_QualityControl c , #FEAT_Process c)) then 1`1 else 0`1
fix
[implies(#FEAT_QualityControl c , #FEAT_Process c)]
1`{FEAT_Prod1=true,FEAT_QualityControl=true,FEAT_Prod2=true,FEAT_InParts=true,FEAT_PartB=true,FEAT_Process=true,FEAT_PartA=true,FEAT_Parallel=true,FEAT_OutProducts=true}
None

Figure 5. CPN for the 150% net and PCs of the PNPL.



Name	Path
T-invariant [0001]	inc_2, arc12, ctrl, arc13, out_2, arc8, cnv_2, arc9, fix, arc14, in, arc10
T-invariant [0002]	out_1, arc6, cnv_1, arc7, inc_1, arc11, ctrl, arc13, fix, arc14, in, arc5

Figure 6. Screenshot of TITAN.

figurations on the system behaviour, and Quinton et al. [30] detect inconsistencies that may arise upon evolving a DSPL. Compared to them, our proposal relies on a translation of Petri nets into CPNs, which enables the analysis of properties based on model checking and the reachability graph.

Other works translate SPLs into Petri nets for analysis. For example, Martínez et al. [31] translate Orthogonal Variability Modeling (OVM) models capturing the variability of an SPL, into Petri nets which are analysed to uncover variants that do not appear in any SPL configuration. Their goal (analysing OVM models) is different from ours (analysing Petri net families), and the analysed properties also differ.

Closer to our approach, some works extend Petri nets with variability and variability-aware analysis techniques. *Feature nets* (FNs) [32] add variability to nets by attaching PCs to either transitions or arcs (but not to both at the same time as we do). The analysis of FNs is lifted to a variable reachability graph extended with PCs; instead we reuse proven standard analysis tools for CPNs out-of-the box. *Dynamic FNs* (DFNs) [32] extend FNs by enabling the firing of transitions to update the feature selection; instead, we use a feature model to decouple the net structure from its variability. *Adaptive Petri nets* [33] use modules to represent the variability of Petri nets. When selecting a configuration, the net is flattened by merging all the modules into a standard Petri net with inhibitor arcs. In our case, we capture the variability with a feature model, and the features are represented in the resulting CPN and can change at run-time.

Finally, Petri nets have been used to implement dynamic reconfigurations in different domains. Weyers [34] uses reference Petri nets (a kind of CPN) to model adaptive graphical user interfaces. Grobelna [35] represents reconfigurable modules of FPGAs as Petri nets, and model checks the satisfaction of the system requirements. Zhang [36] uses object-oriented CPNs with changeable structures to model reconfigurable manufacturing systems. We believe that our work could serve to define and provide analysis capabilities to these and other reconfigurable Petri net-based approaches.

6 Conclusions and Further Work

In this paper, we have presented a mapping from PNPLs into CPNs. The mapping enables the run-time adaptation of the net, and allows using the tooling and analysis methods available for CPNs. We have demonstrated the feasibility of the approach by an implementation atop the TITAN tool, which is able to target CPN Tools.

In the future, we plan to translate the analysis results from CPN Tools back into TITAN. We will also enrich PNPLs with time, to profit from the timing analysis capabilities of CPN Tools. Finally, we would also like to work on animating the token-game, lifting it to the PNPL level.

Acknowledgement

Work funded by the Spanish Ministry of Science (RTI2018-095255-B-I00) and the R&D programme of Madrid (P2018/TCS-4314).

References

[1] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[2] M. L. Rosa, W. M. P. van der Aalst, M. Dumas, and F. Milani, "Business process variability modeling: A survey," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 2:1–2:45, 2017.

[3] S. García, D. Strüber, D. Brugali, A. D. Fava, P. Schillinger, P. Pelliccione, and T. Berger, "Variability modeling of service robots: Experiences and challenges," in *Proc. VaMoS*. ACM, 2019, pp. 8:1–8:6.

[4] Z. Nabi and T. Aized, "Modeling and analysis of carousel-based mixed-model flexible manufacturing system using colored Petri net," *Adv. in Mech. Eng.*, vol. 11, no. 12, pp. 1–14, 2019.

[5] J. Li, X. Dai, and Z. Meng, "Automatic reconfiguration of petri net controllers for reconfigurable manufacturing systems with an improved net rewriting system-based approach," *IEEE Trans Autom. Sci. Eng.*, vol. 6, no. 1, pp. 156–167, 2009.

[6] E. Gómez-Martínez, J. de Lara, and E. Guerra, "Extensible structural analysis of Petri net product lines," *Trans. Petri Nets Other Model. Concurr.*, vol. XV, no. 12530, pp. 1–23, 2021.

[7] D. Perez-Palacin, J. Merseguer, and R. Mirandola, "Analysis of bursty workload-aware self-adaptive systems," in *Proc. ICPE*. ACM, 2012, pp. 75–84.

[8] R. Seiger, S. Huber, and T. Schlegel, "Toward an execution system for self-healing workflows in cyber-physical systems," *Softw. Syst. Model.*, vol. 17, no. 2, pp. 551–572, 2018.

[9] K. Jensen, *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, ser. EATCS Monographs on Theoretical Computer Science. Springer, 1992.

[10] K. Jensen, L. M. Kristensen, and L. Wells, "Coloured petri nets and CPN tools for modelling and validation of concurrent systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 3–4, pp. 213–254, 2007, see also https://cpntools.org/.

[11] L. Northrop and P. Clements, *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[12] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering. Foundations, Principles and Techniques*. Springer-Verlag Berlin Heidelberg, 2005.

[13] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-90-TR-021, 1990.

[14] R. Milner, R. Harper, D. MacQueen, and M. Tofte, *The Definition of Standard ML*, revised edition ed. MIT press, 1997.

[15] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[16] E. M. Clarke, T. A. Henzinger, and H. Veith, "Introduction to model checking," in *Handbook of Model Checking*. Springer, 2018, pp. 1–26.

[17] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[18] Sirius, https://www.eclipse.org/sirius/.

[19] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, and G. Saake, *Mastering software variability with FeatureIDE*. Springer, 2017.

[20] D. L. Berre and A. Parrain, "The Sat4j library, release 2.2," *JSAT*, vol. 7, no. 2-3, pp. 59–6, 2010.

[21] K. Kuchcinski and R. Szymanek, "JaCoP - Java Constraint Programming solver," in *CP Solvers: Modeling, Applications, Integration, and Standardization*, 2013.

[22] E. G. Amparore, "Reengineering the editor of the greatspn framework," in *Proc. PNSE@Petri Nets*, ser. CEUR Workshop Proceedings, vol. 1372. CEUR-WS.org, 2015, pp. 153–170.

[23] A. Zimmermann, "Modelling and performance evaluation with timenet 4.4," in *QEST*, ser. LNCS, vol. 10503. Springer, 2017, pp. 300–303.

[24] T. Freytag and M. Sänger, "WoPeD - An Educational Tool for Workflow Nets," in *BPM (Demos)*, ser. CEUR Workshop Proceedings, vol. 1295. CEUR-WS.org, 2014, p. 31.

[25] S. O. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *Computer*, vol. 41, no. 4, pp. 93–95, 2008.

[26] P. Sawyer, R. Mazo, D. Diaz, C. Salinesi, and D. Hughes, "Using constraint programming to manage configurations in self-adaptive systems," *Computer*, vol. 45, no. 10, pp. 56–63, 2012.

[27] R. Olaechea, J. M. Atlee, A. Legay, and U. Fahrenberg, "Trace checking for dynamic software product lines," in *Proc. SEAMS@ICSE*. ACM, 2018, pp. 69–75.

[28] H. Göttmann, L. Luthmann, M. Lochau, and A. Schürr, "Real-time-aware reconfiguration decisions for dynamic software product lines," in *Proc. SPLC*. ACM, 2020, pp. 13:1–13:11.

[29] I. Ayala, A. V. Papadopoulos, M. Amor, and L. Fuentes, "ProDSPL: Proactive self-adaptation based on dynamic software product lines," *J. Syst. Softw.*, vol. 175, p. 110909, 2021.

[30] C. Quinton, M. Vierhauser, R. Rabiser, L. Baresi, P. Grünbacher, and C. Schuhmayer, "Evolution in dynamic software product lines," *J. Softw. Evol. Process.*, vol. 33, no. 2, 2021.

[31] C. Martínez, H. P. Leone, and S. M. Gonnet, "A petri net approach for representing orthogonal variability models," *International Journal of Computers & Technology*, vol. 9, no. 1, pp. 995–1003, 2013.

[32] R. Muschevici, J. Proença, and D. Clarke, "Feature nets: Behavioural modelling of software product lines," *Softw. Syst. Model.*, vol. 15, no. 4, pp. 1181–1206, 2016.

[33] C. Mai, R. Schöne, J. Mey, T. Kühn, and U. Assmann, "Adaptive Petri nets: A Petri net extension for reconfigurable structures," in *Proc. ADAPTIVE*. Springer, 2018, pp. 15–23.

[34] B. Weyers, "Formal description of adaptable interactive systems based on reconfigurable user interface models," in *The Handbook of Formal Methods in Human-Computer Interaction*. Springer International Publishing, 2017, pp. 273–294.

[35] I. Grobelna, "Model checking of reconfigurable FPGA modules specified by petri nets," *J. Syst. Archit.*, vol. 89, pp. 1–9, 2018.

[36] L. L. Zhang and A. Ittoo, "Development of an rms model based on colored object-oriented petri nets with changeable structures," in *Proc. IEEM*. IEEE, 2009, pp. 1719–1724.