Feature Petri Nets

Radu Muschevici Dave Clarke José Proença
DistriNet & IBBT, Dept. Computer Science
Katholieke Universiteit Leuven, Belgium
Email: {radu.muschevici, dave.clarke, jose.proenca}@cs.kuleuven.be

***Abstract*—In software product line (SPL) engineering, formal modelling and verification are critical for managing the inherent complexity of systems with a high degree of variability. The number of products in an SPL can be exponential in the number of features. Therefore, the challenge when modelling SPL lies in analysing and verifying large, complex models efficiently, in order to ensure that all products behave correctly. The choice of a system modelling formalism that is both expressive and well-established is therefore crucial. In this paper we propose two lightweight extensions to Petri nets: *Feature Petri Nets* provide a framework for modelling and verifying software product lines; and *Dynamic Feature Petri Nets* provide additional support for modelling dynamic software product lines.**

***Keywords*-software product lines; behavioural models; dynamic variability; Petri nets;**

I. Introduction

The need to tailor software applications to specific requirements, such as specific hardware, markets or customer demands, is growing. If each application variant is maintained individually, the management overhead quickly becomes infeasible [1]. Software Produce Line Engineering (SPLE) is seen as a solution.

A Software Product Line (SPL) is a set of software products that share a number of core properties but also differ in certain, well-defined aspects. The products of an SPL are defined and implemented in terms of *features*, which are subsequently combined in specific ways to obtain the final software products. The key advantage hereby over traditional approaches is that all products can be developed and maintained together. A challenge for the SPL approach is to ensure that all products meet their specifications without having to check each product individually, but rather checking the product line itself. This raises the need for novel SPL-specific formalisms to model SPL and reason about and verify their properties.

Petri nets [2] provide a solid formal basis for system modelling. They have been studied and applied widely, and they come with a wealth of formal analysis and verification techniques.

This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Methods (http://www.hats-project.eu), and the K.U.Leuven BOF-START project STRT1/09/031 DesignerTypeLab.

The main contribution of this paper is two Petri net variants suitable for modelling software product lines. These models enable the specification of system behaviour such as resource usage and workflow of the entire product line in one model. We extend Petri nets in two steps. We start by guiding the execution of a Petri net based on the feature selection, and later we introduce a mechanism to update the feature selection based on the execution of the Petri net. We call the first model *Feature Petri Nets* (FPN), and the second *Dynamic Feature Petri Nets* (DFPN). Our models provide an elegant separation between behaviour, modelled by the underlying Petri net, and available functionality, modelled by feature sets.

The paper is organised as follows. The next section shows a motivating example. Section III formally introduces the notion of Feature Petri nets. Section IV motivates Dynamic Feature Petri Nets and Section V presents their formal semantics. Section VI discusses the relation of FPN and DFPN to Petri Nets and how they can be used to model SPL. Section VII discuses related work. Section VIII presents our conclusions and future work.

II. Feature Petri Net Example

We illustrate the modelling challenge in SPLE using an example of a software product line of coffee machines. A manufacturer of coffee machines offers products to match different demands, from the basic black coffee dispenser, to more sophisticated machines, such as ones that can add milk or sugar, or charge a payment for each serving. Each machine variant needs to run software adapted to the set of hardware features. Such a family of different software products that share functionality is typically developed using an SPLE approach, that is, as one piece of software structured along distinct features. This has major advantages in terms of code reuse, maintenance overhead and so forth. The challenge is ensuring that the software works appropriately in all product configurations.

Petri nets are used to specify how systems behave. Figure 1 presents an example of a Petri net for a coffee machine. Places, represented by circles, can host tokens, represented by dots or a natural number, and the execution of a Petri net consists of the flow of tokens between places via transitions, depicted as filled rectangles. In our example, the coffee machine has a capacity for n coffee capsules; it can brew

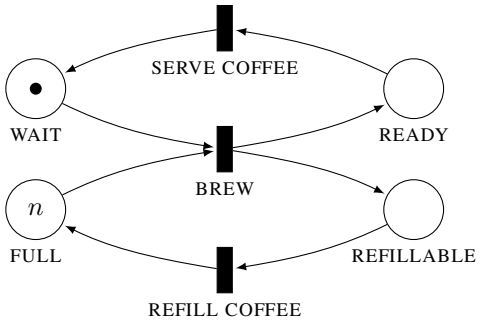

Figure 1: Petri net model of a basic coffee machine that can only dispense coffee.

and serve coffee, and refill the machine with new coffee capsules. Note that in our model the action 'refill' does not mean filling the coffee reservoir completely, but refilling only one unit of coffee.

If we now add the *Milk* feature, that is, if we assume that the coffee machine can also add milk to the coffee, we need to adapt the Petri net. Furthermore, for every new feature we would need to specify a new Petri net for each possible combination of features, resulting in an explosion of combinations.

We address this problem by annotating transitions with *application conditions* [3], which are logical formula over features that reflect when the transition is enabled. Our example considers a product line whose products are over the set of features $\{Coffee, Milk\}$. The new Petri net model is called *Feature Petri Nets* (FPN). Figure 2 exemplifies an FPN of a coffee machine with a milk reservoir. The conditions on the transitions reflect that the three transitions on the right-hand side can be taken only when both features *Coffee* and *Milk* are present, and the three transitions on the left-hand side can be taken when the *Coffee* feature is present. The restriction of the example net to the transitions that can fire for feature selection $\{Coffee\}$ is exactly the Petri net in Figure 1, after removing unreachable places.

III. Feature Petri Nets

Feature Petri nets (FPN) are a Petri net variant used to model the behaviour of an entire software product line. For this purpose, FPN have *application conditions* [3] attached to their transitions. An application condition is a boolean logical formula over a set of features, describing the feature combinations to which the transition applies. It constitutes a necessary (although not sufficient) condition for the transition to fire. In effect, if the application condition is false, it is as if the transition was not present.

We define Feature Petri Nets and give their semantics. We present two semantic accounts of FPN. First, when a set of features is selected, an FPN *directly* models the behaviour of the product corresponding to the feature selection. Second, by *projecting* an FPN onto a feature selection, one obtains a Petri net describing the behaviour of the same product. We show that these two notions of semantics coincide.

We start with some necessary preliminaries, first by defining multisets and basic operations over multisets. Then we define Petri nets and their behaviour.

Definition 1 (Multiset)**.** *A* multiset *over a set S is a mapping $M : S \to \mathbb{N}$.*

We view a set S as a multiset in the natural way, that is, $S(x) = 1$ if $x \in S$, and $S(x) = 0$ otherwise. We also lift arithmetic operators to multisets as follows. For any function $\odot : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ and multisets M_1, M_2, we define $M_1 \odot M_2$ as $(M_1 \odot M_2)(x) = M_1(x) \odot M_2(x)$.

A. Petri Nets

To ground our theory, we recall the terminology and notation surrounding Petri nets [4].

Definition 2 (Petri Net)**.** *A Petri net is a tuple (S, T, R, M_0), where S and T are two disjoint finite sets, R is a relation on $S \cup T$ (the* flow relation*) such that $R \cap (S \times S) = R \cap (T \times T) = \emptyset$, and M_0 is a multiset over S, called the* initial marking*. The elements of S are called* places *and the elements of T are called* transitions*. Places and transitions are called* nodes*.*

Sometimes we omit the initial marking M_0.

Definition 3 (Marking of a Petri Net)**.** *A marking M of a Petri net (S, T, R) is a multiset over S. A place $s \in S$ is* marked *iff $M(s) > 0$.*

Definition 4 (Pre-sets and post-sets)**.** *Given a node x of a Petri net, the set ${}^\bullet x = \{y \mid (y, x) \in R\}$ is the* pre-set *of x and the set $x^\bullet = \{y \mid (x, y) \in R\}$ is the* post-set *of x.*

Definition 5 (Enabling)**.** *A marking M* enables *a transition $t \in T$ if it marks every place in ${}^\bullet t$, that is, if $M \geq {}^\bullet t$.*

The behaviour of a Petri net is a sequence of states, where each state is defined by a marking. The change from the current state to a new state occurs by the firing of a transition. A transition t can fire if it is enabled. Firing transition t changes the marking of the Petri net by decreasing the marking of each place in the pre-set of t by one, and increasing the marking of each place in the post-set of t by one.

Definition 6 (Transition occurrence rule)**.** *Given a Petri net $N = (S, T, R)$, a transition $t \in T$* occurs*, leading from a state with marking M_i to a state with marking M_{i+1}, denoted $M_i \xrightarrow{t} M_{i+1}$, iff the following two conditions are met:*

$$M_i \geq {}^\bullet t \qquad \text{(enabling)}$$
$$M_{i+1} = (M_i - {}^\bullet t) + t^\bullet \qquad \text{(computing)}$$

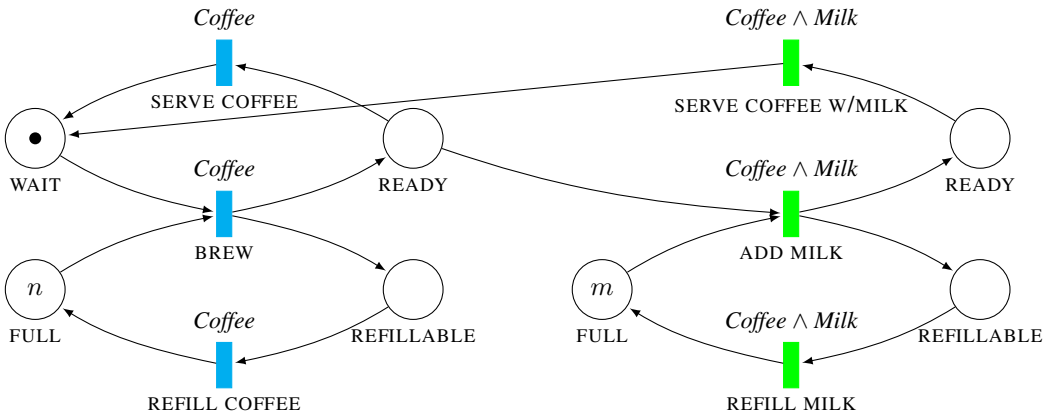

Figure 2: FPN of the product line $\{\{Coffee\}, \{Coffee, Milk\}\}$ showing each product in its initial state. Each transition has an application condition attached. Colour is used to visually group transitions according to application conditions.

The behaviour defined above is also known as the *firing* of a transition. Transitions fire sequentially, that is, only one transition occurs at a time.

Definition 7 (Petri net trace)**.** *Given a Petri net* $N = (S, T, R, M_0)$*, the behaviour the net exhibits by passing through a sequence of states with markings* $M_0, \ldots, M_n$*, where each change of marking is triggered by a transition occurrence* $M_i \xrightarrow{t_i} M_{i+1}$*, is called a* trace*. A trace is written* $M_0 \xrightarrow{t_0} M_1 \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} M_n$*.*

Definition 8 (Petri net behaviour)**.** *The behaviour of a Petri net is given by the set of all traces from a given initial marking.*

B. Feature Petri Nets

Definition 9 (Application condition)**.** *An* application condition φ *[3] is a logical (boolean) constraint over a set of features* F*, defined by the following grammar:*

$$\varphi \quad ::= \quad a \mid \varphi \wedge \varphi \mid \neg\varphi,$$

where $a \in F$*. The remaining logical connectives can be encoded as usual. We write* Φ_F *to denote the set of all application conditions over* F*.*

Definition 10 (Satisfaction of application conditions)**.** *Given an application condition* φ *and a set of features FS, called a* feature selection*, we say that FS satisfies* φ*, written as* $FS \models \varphi$*, iff*

$$\begin{array}{lll} FS & \models a & \text{iff } a \in FS \\ FS & \models \varphi_1 \wedge \varphi_2 & \text{iff } FS \models \varphi_1 \text{ and } FS \models \varphi_2 \\ FS & \models \neg\varphi & \text{iff } FS \not\models \varphi. \end{array}$$

We are now in the position to introduce Feature Petri Nets.

Definition 11 (Feature Petri Net)**.** *A Feature Petri net is a tuple* $N = (S, T, R, M_0, F, f)$*, where* (S, T, R, M_0) *is a Petri net,* F *is set of features, and* $f : T \to \Phi_F$ *is a function associating each transition with an application condition from* Φ_F*.*

For $f(t)$, the application condition associated with transition t, write φ_t. For conciseness, we say that a feature selection *FS* satisfies transition t whenever $FS \models \varphi_t$.

We now define the behaviour of Feature Petri Nets for a given (static) feature selection.

Definition 12 (Transition occurrence rule for FPN)**.** *Given an FPN* $N = (S, T, R, M_0, F, f)$ *and a feature selection* $FS \subseteq F$*, a transition* $t \in T$ occurs*, leading from a state with marking* M_i *to a state with marking* M_{i+1}*, denoted* $(M_i, FS) \xrightarrow{t} (M_{i+1}, FS)$*, iff the following three conditions are met:*

$$\begin{array}{ll} M_i \geq {}^\bullet t & \text{(enabling)} \\ M_{i+1} = (M_i - {}^\bullet t) + t^\bullet & \text{(computing)} \\ FS \models \varphi_t & \text{(satisfaction)} \end{array}$$

In the above definition the state of the Petri net is denoted by a tuple consisting of a marking and a feature selection, even though we assume the feature selection is static (constant). Later on, we will look at dynamic feature selections which can change during execution.

The transition rule for FPN is used to define traces that describe the FPN's behaviour in the same way as Petri nets.

Definition 13 (FPN Trace)**.** *Given an FPN* $N = (S, T, R, M_0, F, f)$ *and a feature selection* $FS \subseteq F$*, the behaviour the net exhibits by passing through a sequence of markings* $M_0, \ldots, M_n$*, where each change of marking is triggered by a transition occurrence* $(M_i, FS) \xrightarrow{t_i} (M_{i+1}, FS)$*, is called a* trace over *FS. A trace is written* $(M_0, FS) \xrightarrow{t_0} (M_1, FS) \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} (M_n, FS)$*.*

Given an FPN, there is a set of traces representing the behaviour of the FPN for each feature selection.

Definition 14 (FPN behaviour for a given feature selection)**.** *Given an FPN $N = (S, T, R, M_0, F, f)$ and a feature selection $FS \subseteq F$, the behaviour of N for FS, denoted* $\mathrm{Beh}(N, FS)$ *is the set of all traces over FS from the initial marking M_0.*

If we consider all possible feature selections, we can express the behaviour of the FPN.

Definition 15 (FPN Behaviour)**.** *Given an FPN $N = (S, T, R, M_0, F, f)$, we define* $\mathrm{Beh}(N)$ *to be the combined set of behaviours for all feature selections over F:*

$$\mathrm{Beh}(N) = \bigcup_{FS \in \mathcal{P}F} \mathrm{Beh}(N, FS).$$

C. Projection-based Semantics of FPN

We now present an alternative semantics of Feature Petri Nets. Given a feature selection, the semantics of an FPN is a Petri net consisting of just the transitions satisfying the feature selection.

Definition 16 (Projection)**.** *Given a Feature Petri Net $N = (S, T, R, M_0, F, f)$ and a feature selection $FS \subseteq F$, the* projection *of N onto FS, denoted $N{\downarrow}FS$, is a Petri net (S, T', R', M_0), with $T' = \{t \in T \mid FS \models \varphi_t\}$ and the flow relation $R' = R \cap ((S \cup T') \times (S \cup T'))$.*

One *projects* N onto a feature selection *FS* by evaluating all application conditions φ_t with respect to *FS* for transitions $t \in T$. If *FS* does not satisfy φ_t, then transition t is removed from the Petri net. All application conditions are also removed when projecting.

The behaviour of the projection of a Feature Petri net N onto a feature selection *FS* coincides with the behaviour of N for *FS*, as stated by the following theorem.

Theorem 1. *Given a Feature Petri Net N and $FS \subseteq F$, then:*

$$\mathrm{Beh}(N, FS) \;=\; \mathrm{Beh}(N{\downarrow}FS).$$

Proof: ($\subseteq$) We show that every trace $\sigma \in \mathrm{Beh}(N, FS)$ is also a trace in $\mathrm{Beh}(N{\downarrow}FS)$. Firstly, the initial markings M_0 coincide in both petri nets. Secondly, if $(M, FS) \xrightarrow{t} (M', FS)$ then, by Definition 14, $FS \models \varphi_t$, and by Definition 16 it is also a transition of $N{\downarrow}FS$. Hence, $M \xrightarrow{t} M'$.

($\supseteq$) Following a similar reasoning as before, we show that every trace $\sigma \in \mathrm{Beh}(N{\downarrow}FS)$ is also a trace in $\mathrm{Beh}(N, FS)$. Observe that, if $M \xrightarrow{t} M'$, then t is a transition of $N{\downarrow}FS$, and by Definition 16 $FS \models \varphi_t$. Hence, by Definition 14 we conclude that also $(M, FS) \xrightarrow{t} (M', FS)$. ■

IV. Dynamic Feature Reconfiguration Example

Assuming that a product is composed from a static selection of features is sometimes too restrictive. As an example, we can think of a modular appliance, some of whose features can be disabled temporarily. For example, a coffee machine using fresh milk instead of milk powder allows the removal of the milk reservoir, in order to store it in the fridge. That change in the hardware configuration may entail a change in the software configuration. Modelling the presence/absence behaviour of the *Milk* feature may entail a significant modelling effort.

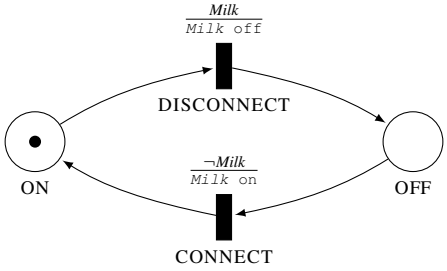

Figure 3: DFPN modelling the ability to connect/disconnect a feature at runtime.

To accommodate modelling this kind of dynamic feature reconfiguration, we introduce Dynamic Feature Petri Nets (DFPN). DFPN associate simple update expressions to transitions. Upon firing of a transition, updates affect the feature selection in effect.

In our example, switching the *Milk* feature on and off can be modelled by the DFPN in Figure 3, as an independent addition to the model in Figure 2. Associated to the DISCONNECT transition is the *update* expression "`Milk off`". By firing the DISCONNECT transition, the current feature selection is updated, dropping the *Milk* feature. This action globally disables all transitions whose application condition depends on the *Milk* feature (that is, ADD MILK, REFILL MILK and SERVE COFFEE W/MILK in Figure 2). Conversely, firing the CONNECT transition re-enables all transitions conditioned on the *Milk* feature.

The feature reconfiguration model can remain disconnected from the "functional" model if the user interaction of removing/reconnecting the *Milk* feature can occur independently of the state the coffee machine. Alternatively, we can assume that the reconfiguration of features depends on the functional model. Figure 4 shows a model where removing/reconnecting the milk reservoir is only allowed when the machine is in a waiting state, prohibiting, for example, its removal when the machine is in the process of brewing coffee.

V. Dynamic Feature Petri Nets

Dynamic Software Product Lines (DSPL) is an area of research concerned with runtime variability of systems [5]. DSPL is an umbrella concept that addresses dynamic reconfiguration of products (i.e. features are added and removed

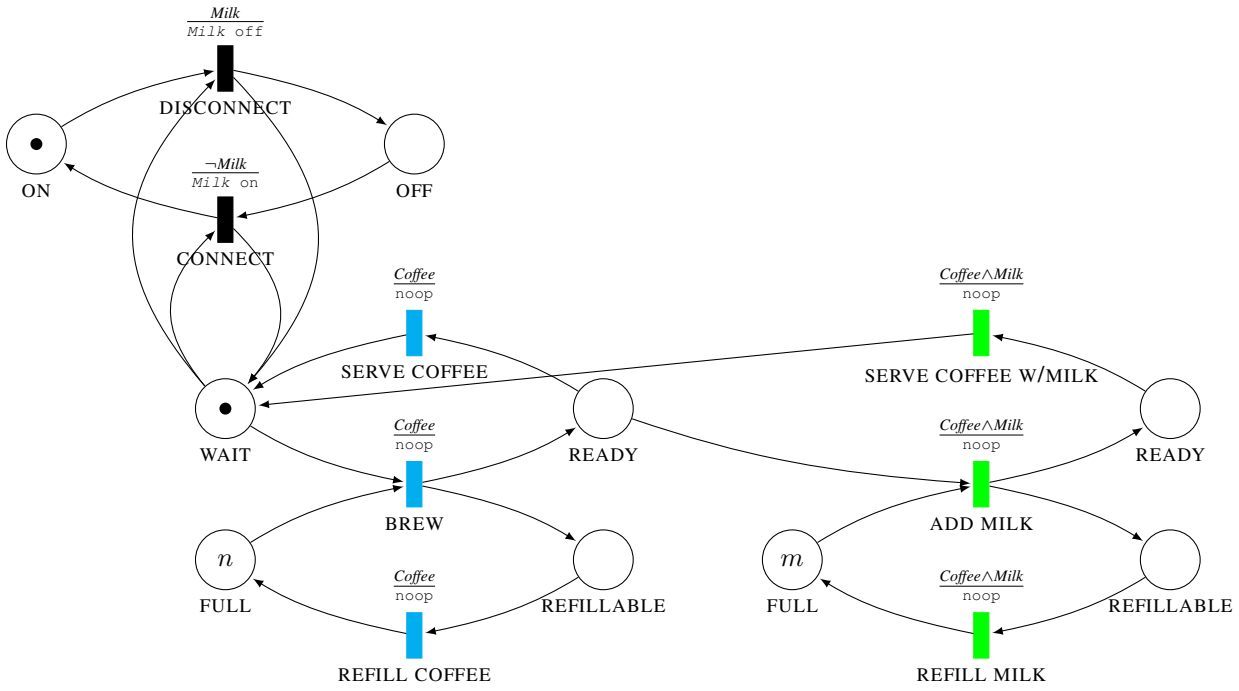

Figure 4: DFPN (initial state) of a dynamically reconfigurable product line. Whenever transition DISCONNECT fires, feature *Milk* is switched off, disabling all transitions that are conditioned on *Milk*. It is enabled again by firing CONNECT.

at runtime), but also dynamic evolution of the product line itself (typically referred to as "meta-variability"). Pushing the binding time of features to runtime is often motivated by a changeable operational context, to which a product has to adapt in order to provide context-relevant services or meet quality requirements.

We extend Feature Petri Nets to capture the dynamic reconfiguration of products, resulting in a more general Petri net model. In our approach we associate to each transition an *update expression* that describes how the feature selection evolves after the transition. The resulting model is called *Dynamic Feature Petri Nets* (DFPN). DFPN extend Feature Petri nets by adding a variable feature selection to the state of the Petri net, and associating application conditions and update expressions over the feature set to the transitions. This extension enable more concise descriptions of systems based on feature models, without adding expressive power with respect to Petri nets. We now define update expressions before formalising DFPN.

Definition 17 (Update)**.** *An* update *is defined by the following grammar:*

$$u \quad ::= \quad \texttt{noop} \mid a\ \texttt{on} \mid a\ \texttt{off} \mid u;u$$

where $a \in F$ and F is a set of features. We write U_F to denote the set of all updates over F.

Given a feature selection $FS \in F$, an update expression modifies FS according to the following rules:

$$FS \xrightarrow{\texttt{noop}} FS$$

$$FS \xrightarrow{a\ \texttt{on}} FS \cup \{a\}$$

$$FS \xrightarrow{a\ \texttt{off}} FS \setminus \{a\}$$

$$\frac{FS \xrightarrow{u_0} FS' \qquad FS' \xrightarrow{u_1} FS''}{FS \xrightarrow{u_0;u_1} FS''}$$

Definition 18 (Dynamic Feature Petri Net)**.** *A DFPN is a tuple $N = (S, T, R, M_0, F, f, u)$, where (S, T, R, M_0, F, f) is an FPN and u is a function $T \to U_F$, associating each transition with an update from U_F.*

We write u_t to denote the update expression $u(t)$ associated with a transition t.

Definition 19 (DFPN transition occurrence)**.** *Given a DFPN $N = (S, T, R, M_0, F, f, u)$ and an initial feature selection $FS_0 \subseteq F$, a transition $t \in T$* occurs*, leading from a state (M_i, FS_i) to a state (M_{i+1}, FS_{i+1}), denoted $(M_i, FS_i) \xrightarrow{t} (M_{i+1}, FS_{i+1})$, iff the following four conditions are met:*

$$M_i \geq {}^\bullet t \qquad \text{(enabling)}$$

$$M_{i+1} = (M_i - {}^\bullet t) + t^\bullet \qquad \text{(computing)}$$

$$FS_i \models \varphi_t \qquad \text{(satisfaction)}$$

$$FS_i \xrightarrow{u_t} FS_{i+1} \qquad \text{(update)}$$

Definition 20 (DFPN trace)**.** *Given a DFPN* $N = (S, T, R, M_0, F, f, u)$*, the behaviour the net exhibits by assuming a sequence of states* $(M_0, FS_0) \dots (M_n, FS_n)$*, where each change of state is triggered by a transition occurrence* $(M_i, FS_i) \xrightarrow{t_i} (M_{i+1}, FS_{i+1})$*, is called a* trace*. A trace is written* $(M_0, FS_0) \xrightarrow{t_0} (M_1, FS_1) \xrightarrow{t_1} \cdots \xrightarrow{t_{n-1}} (M_n, FS_n)$.

If we consider all possible traces, we obtain the behaviour of the FPN.

Definition 21 (DFPN Behaviour)**.** *Given a DFPN* $N = (S, T, R, M_0, F, f, u)$*, we define* $\mathrm{Beh}(N)$ *to be the set of all traces starting with the initial state* (M_0, FS_0).

VI. Discussion

Petri nets are a general modelling formalism, proposed for a wide variety of applications. FPN and DFPN leverage the power of Petri nets for modelling static and dynamic software product lines. They offer conciseness and convenience when modelling entire software families. Theorem 1 shows that an FPN is equivalent in behaviour to a set of Petri nets, one for each product defined by the SPL. DFPN additionally provision for dynamic SPL, by allowing explicit modelling of feature reconfiguration as part of the behavioural model.

By adding update expressions to Feature Petri Nets, Dynamic Feature Petri Nets do not gain more expressive power than Petri nets, but provide a more elegant separation of concerns. This approach offers orthogonality of the feature reconfiguration from the underlying behaviour, but in a way that enables the reconfiguration to depend upon the underlying behaviour and vice versa. We now justify intuitively this claim.

In our motivating example of the coffee machine, the availability of milk is represented by Petri net tokens, while the capability of doing actions related to the *Milk* feature is represented by the feature selection. However, the "activation" of transitions based on the available features can also be encoded using more complex Petri nets, where certain markings denote possible products. We illustrate this idea in Figure 5. The place MILK ON is associated to the selection of the feature *Milk*. When there is a token in this place, the transitions $t_1 \dots t_n$ are enabled. They are allowed to occur only when there is a token in MILK ON, and this token remains in the same place. A similar approach can be used to convert any Dynamic Feature Petri Net into a more complex Feature Petri Net.

We present Feature Petri Nets as a novel SPL modelling formalism, but we do not examine how well this approach fares in practice. If used on a real-world product line, issues of scalability, and the need for a more modular modelling workflow could arise. These are subject to future research, in which we expect to devise a workflow where partial models can be reconciled to a coherent global model. This approach will benefit from previous work on Petri nets, since methods

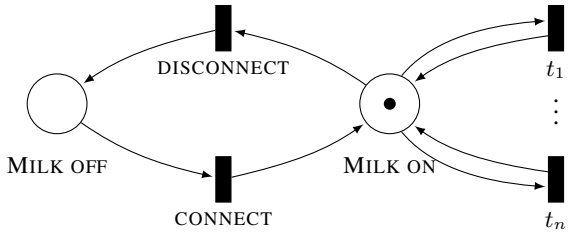

Figure 5: Encoding a feature selection as a Petri net marking

for composing and refining nets are well-studied topics [6]. In addition, many analysis techniques exist to determine the behavioural correctness of a Petri net design [2].

VII. Related Work

Our research relates to a number of areas, specifically Petri net based formalisms, behavioural specification of software product lines and dynamic SPL research. We highlight the most relevant works in these fields.

A range of Petri net extensions based on a modified transition occurrence rule have been proposed, some of which are specifically tailored for representing dynamic, self-modifying behaviour [7, 8, 9, 10]. Unlike our approach, these formalisms generally exceed the expressive power of Petri nets. As a general consequence, properties such as reachability, boundedness and liveness are not decidable for these extensions, and they lack the full range of mathematical tools available to analyse normal Petri nets.

Inhibitor arc Petri nets [7] can test whether a place is empty by conditioning transitions on the absence of tokens. By modelling individual features as places, the presence or absence of tokens could represent whether a feature is on or off. An application condition could be encoded by including feature places in the pre-sets of transitions, thereby conditioning its firing on the presence or absence of features. Compared to our proposed approach, this entails a more complex net, with unclear boundaries between the functional and structural models. *Conditional Petri nets* [9] associate a transition to a formal language over transitions. Extending the classical occurrence rule, a transition is enabled only if the sequence of transitions that occurred in the past is in that language. An FPN could be encoded as a conditional Petri net by encoding application conditions in a language over the alphabet of transitions.

In *self-modifying Petri nets* [8], the flow relation changes dynamically according to the number of tokens at certain places in the net. A transition is enabled if it can fire as many tokens as present in the places referenced by its incoming arcs. *Dynamic Petri nets* [11] are similar, but have an "external control" through which the net's structure can be changed by adding or removing arcs between nodes. Certain behaviour can thus be enabled or disabled by integrating

or isolating places and transitions. These Petri net designs, although sporting a mechanism of self-modification, are geared towards dynamic changes in throughput, rather that the discrete activation/deactivation of behaviour offered by DFPN. Using *net rewriting systems* [10], dynamic changes in the configuration of a Petri net are described using a "rewriting rule", which relates places and transitions of the two net configurations to each other. It is conceivable to model a dynamic SPL as a sequence of configurations and a set of rewriting rules which relate each configuration to the next. The DFPN approach, however, has the advantage of using a single model, in which each state clearly references a feature selection.

Compared to the surveyed Petri net formalisms, (D)FPN semantics are simpler, being closer to the application domain of variability modelling: through application conditions and update expressions they refer directly to the feature model of the SPL.

Various formalisms have been adopted for specifying the behaviour of software product lines, with the aim of providing a basis for analysis and verification of such models.

UML activity diagrams have been used to model the behaviour of SPL by superimposing several such diagrams in a single model [12]. Attached to the activity diagram's elements are "presence expression", which are similar in scope to application conditions. Models of products are obtained using model-to-model transformation by evaluating the presence conditions in the light of a given feature configuration. Compared to activity diagrams, Petri nets have a stronger formal foundation, with a larger spectrum of analysis and verification techniques, although, several studies have expressed the semantics of UML diagram using Petri nets (e.g. [13]).

Gruler et al. extended Milner's CCS with a product line variant operator that allows an alternative choice between two processes [14, 15]. This calculus, referred to as *PL-CCS*, includes information about variability: by defining dependencies between features, once can control the set of valid configurations.

Variability is often modelled using transition systems enhanced with product-related information. *Modal transition systems* (MTS) [16] allow optional transitions, lending themselves as a tool for modelling a set of behaviours at once [17]. Generalised extended MTS [18] introduce cardinality-based variability operators and propose to use temporal logic formulas to associate related variation points. Asirelli et al. encode MTS using propositional deontic logic formulas [19]. *Modal I/O automata* [20] are a behavioural formalism for describing the variability of components based on MTS and I/O automata. Mechanisms of component composition are provided to support a product line theory. These approaches do not relate behaviour to elements of a structural variability model. *Featured transition systems* (FTS) [21] are an extension of labeled transition systems. Similar to Feature Petri Nets, transitions are explicitly labeled with respect to a feature model, and a feature selection determines the subset of active transitions. In FTS, transitions are mapped to single features. Transition priorities are used to deal with undesired non-determinism when selecting from transitions associated to different features. Using constraints, priorities are no longer required because we can negate the features in other transitions to obtain the same effect. The authors also envisage an alternative approach in which boolean expressions over features could be used.

The research fields of dynamic software product lines, context-aware and self-adapting systems include works on modelling dynamic systems at the levels of architecture [22, 23] and implementation [24, 25]. Between these, there is a notable gap with respect to formal specification and analysis of dynamic SPL behaviour.

VIII. Conclusion and Future Work

This paper introduces Feature Petri Nets (FPN) and Dynamic Feature Petri Nets (DFPN), two lightweight Petri net extensions designed for modelling the behaviour of software product lines. The transition firing in an FPN is conditional on the presence of certain features through *application conditions*. Application conditions explicitly relate behaviour to feature configurations, while keeping that behaviour separate from the SPL structure. FPN capture the behaviour of entire product lines in a single, concise model, opening the way for efficient analysis and verification.

The DFPN model extends FPN with the ability to express dynamic variability. *Update expressions* associated with DFPN transitions make it easy to model changes in the feature selection of a product based on its execution: firing a transition updates the feature configuration in place. To our knowledge, this is the first model to capture both the variable and dynamic aspects of SPL in a single formalism.

In the future we expect to improve the modularity of our approach. Currently we use a single monolithic net to express the behaviour of all possible products. This issue can be addressed by adopting existing complexity managing techniques for Petri nets, such as abstraction, refinement, and composition [6], to allow the development of partial DFPN for views of a system, which can be later combined into a single coherent model. The main challenge becomes avoiding the development of one model for each possible product, and describing the global model by the combination of key partial DFPN.

Acknowledgment

We would like to thank the anonymous reviewers for their helpful suggestions and comments.

References

[1] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering*. Springer, 2005. I

[2] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989. I, VI

[3] I. Schaefer, "Variability modelling for model-driven development of software product lines," in *International Workshop on Variability Modelling of Software-intensive Systems*, Linz, Austria, 2010. II, III, 9

[4] J. Desel and J. Esparza, *Free choice Petri nets*. New York, NY, USA: Cambridge University Press, 1995. III-A

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

[6] M. D. Jeng and F. DiCesare, "A review of synthesis techniques for Petri nets with applications to automated manufacturing systems," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 23, no. 1, pp. 301–312, Jan.Feb. 1993. VI, VIII

[7] T. Agerwala and M. Flynn, "Comments on capabilities, limitations and "correctness" of Petri nets," in *International Symposium on Computer Architecture*. New York, NY, USA: ACM, 1973, pp. 81–86. VII

[8] R. Valk, "Self-modifying nets, a natural extension of Petri nets," *Automata, Languages and Programming*, pp. 464–476, 1978. VII

[9] F.-L. Ţiplea, *On conditional grammars and conditional Petri nets*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1994, pp. 431–455. VII

[10] M. Llorens and J. Oliver, "Structural and dynamic changes in concurrent systems: reconfigurable Petri nets," *IEEE Transactions on Computers*, vol. 53, no. 9, pp. 1147–1158, Sep. 2004. VII

[11] M.-K. Ghabri and P. Ladet, "Dynamic Petri nets and their applications," in *International Conference on Computer Integrated Manufacturing and Automation Technology*, 1994, pp. 93–98. VII

[12] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *Generative Programming and Component Engineering*. Springer, 2005, pp. 422–437. VII

[13] U. Farooq, C. P. Lam, and H. Li, "Transformation methodology for UML 2.0 activity diagram into colored Petri nets," in *IASTED international conference on Advances in computer science and technology*. Anaheim, CA, USA: ACTA Press, 2007, pp. 128–133. VII

[14] A. Gruler, M. Leucker, and K. Scheidemann, "Calculating and modeling common parts of software product lines," in *International Software Product Line Conference*, September 2008, pp. 203–212. VII

[15] ——, "Modeling and model checking software product lines," in *International Conference on Formal Methods for Open Object-based Distributed Systems*, ser. LNCS, G. Barth and F. de Boer, Eds., vol. 5051. Springer, 2008, pp. 113–131. VII

[16] K. Larsen and B. Thomsen, "A modal process logic," in *Logic in Computer Science*, 5-8 1988, pp. 203–210. VII

[17] D. Fischbein, S. Uchitel, and V. Braberman, "A foundation for behavioural conformance in software product line architectures," in *International Workshop on the Role of Software Architecture in Analysis and Testing (ROSATEA)*. New York, NY, USA: ACM, 2006, pp. 39–48. VII

[18] A. Fantechi and S. Gnesi, "Formal modeling for product families engineering," *International Software Product Line Conference*, pp. 193–202, 2008. VII

[19] P. Asirelli, M. H. ter Beek, S. Gnesi, and A. Fantechi, "Deontic logics for modeling behavioural variability," in *International Workshop on Variability Modelling of Software-intensive Systems*, 2009, pp. 71–76. VII

[20] K. Larsen, U. Nyman, and A. Wąsowski, "Modal I/O automata for interface and product line theories," *Programming Languages and Systems*, pp. 64–79, 2007. VII

[21] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: Efficient verification of temporal properties in software product lines," in *International Conference on Software Engineering*. IEEE, 2010, pp. 335–344. VII

[22] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch, "Using product line techniques to build adaptive systems," in *International Software Product Line Conference*, 2006, pp. 141–150. VII

[23] H. Shokry and M. A. Babar, "Dynamic software product line architectures using service-based computing for automotive systems," in *International Software Product Line Conference*, 2008, pp. 53–58. VII

[24] P. Costanza and T. D'Hondt, "Feature descriptions for context-oriented programming," in *International Software Product Line Conference*, 2008, pp. 9–14. VII

[25] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel, "Code generation to support static and dynamic composition of software product lines," in *International Conference on Generative Programming and Component Engineering*. New York, NY, USA: ACM, 2008, pp. 3–12. VII