SCISPACE formerly Typeset

Open access • Journal Article • DOI:10.1007/S10270-015-0475-Z

Feature Nets: behavioural modelling of software product lines — Source link

Radu Muschevici, José Proença, Dave Clarke

Institutions: Technische Universität Darmstadt, University of Minho, Uppsala University

Published on: 01 Oct 2016 - Software and Systems Modeling (Springer Berlin Heidelberg)

Topics: Feature model, Software product line, Process architecture, Petri net and Software system

Related papers:

- A Classification and Survey of Analysis Strategies for Software Product Lines
- Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking
- Model checking lots of systems: efficient verification of temporal properties in software product lines
- Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints
- Modal I/O automata for interface and product line theories

Share this paper:

View more about this paper here: https://typeset.io/papers/feature-nets-behavioural-modelling-of-software-product-lines-lb95qvdfjw

Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Feature Nets

Behavioural modelling of software product lines

Radu Muschevici · José Proença · Dave Clarke

Received: date / Accepted: date

Abstract Software product lines (SPL) are diverse systems that are developed using a dual engineering process: (a) *family engineering* defines the commonality and variability among all members of the SPL, and (b) *application engineering* derives specific products based on the common foundation combined with a variable selection of features. The number of derivable products in an SPL can thus be exponential in the number of features. This inherent complexity poses two main challenges when it comes to modelling: Firstly, the formalism used for modelling SPLs needs to be modular and scalable. Secondly, it should ensure that all products behave correctly by providing the ability to analyse and verify complex models efficiently. In this paper we propose to integrate an established modelling formalism (Petri nets) with the domain of software product line engineering. To this end we extend Petri nets to *Feature Nets*. While Petri nets provide a framework for formally modelling and verifying single software systems, Feature Nets offer the same sort of benefits for software product lines. We show how SPLs can be modelled in an incremental, modular fashion using Feature Nets, provide a Feature Nets variant that supports modelling dynamic SPLs, and propose an analysis method for SPL modelled as Feature Nets. By facilitating the construction of a single model that includes the various behaviours exhibited by the products in an SPL, we make a significant step towards efficient and practical quality assurance methods for software product lines.

Keywords Behavioural Modelling · Software Product Lines · Petri Nets · Variability

Radu Muschevici
Dept. Computer Science, TU Darmstadt
E-mail: radu.muschevici@cs.tu-darmstadt.de

José Proença
iMinds-Distrinet, KU Leuven
and HASLab/INESC TEC, Universidade do Minho
E-mail: jose.proenca@cs.kuleuven.be

Dave Clarke
Dept. Information Technology, Uppsala University
and iMinds-Distrinet, KU Leuven
E-mail: dave.clarke@it.uu.se

1 Introduction

The need to tailor software applications to varying requirements, such as specific hardware, markets or customer demands, is growing. If each application variant is maintained individually, the overhead of managing all the variants quickly becomes infeasible [33]. Software Produce Line Engineering (SPLE) is seen as a solution to this problem. 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 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 SPLE is to ensure that all products meet their specifications without having to check each product individually, by checking the product line itself. This raises the need for novel SPL-specific formalisms to model SPLs and reason about and verify their properties.

This paper presents a line of research into using Petri nets to model the behaviour of software product lines. Petri nets [30] 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. The modelling formalism we develop is *Feature Petri Nets*, or *Feature Nets* (FN) for short. Feature Nets are a Petri net extension that enables the specification of the behaviour of an entire software product line (a set of systems) in one single model. The behaviour of a FN is conditional on the features appearing in the product line. The ability to model the behaviour of a set of systems in a single model brings us closer to the goal of reasoning about multiple systems in a practical way.

We extend Petri nets in three steps. We start by guiding the execution of a Petri net based on the features that are selected. We call this model *transition-labelled Feature Nets* (FN) because it conditions the firing of transitions on the feature selection. In the second step we introduce a mechanism to dynamically update the feature selection based on the execution of the Petri net. This model is called *Dynamic Feature Nets* (DFN). In a third step we improve upon the original FN definition with the aim of supporting net composition. The improved model allows us to develop a technique for constructing larger Feature Nets from smaller ones to model the addition of new features to an SPL. The feature selection is now associated with Petri net *arcs*, hence this model is called *arc-labelled Feature Nets*. We provide correctness criteria for ensuring that the composition of arc-labelled FNs preserves the behaviour of the original model(s). Our three Feature Net models provide an elegant separation between behaviour, modelled by the underlying Petri net, and available functionality, modelled by feature sets. This special issue article extends previous publications [31,32], introducing reachability analysis of Feature Nets and exploiting analysis techniques for Feature Nets. It includes a more extensive presentation of the formalism, a comprehensive discussion and reports on practical applications, thus providing a better insight into the power of Feature Nets.

The paper is structured as follows. Section 2 illustrates the modelling challenge in SPL engineering with an example, thereby motivating the need for Feature Nets. Section 3 provides the necessary background on Petri nets. Section 4 presents *transition-labelled* Feature Nets, our original formalism for modelling SPL. Section 5 extends Feature Nets with a dynamic component that adds support for modelling dynamic SPL. In Section 6 we improve the original FN definition with the goal to better support

modular composition, presenting *arc-labelled* Feature Nets. Arc-labelled FN support a technique for constructing a larger Feature Net from smaller ones to model the addition of new features to an SPL. Section 7 discusses the encoding of Feature Nets into Petri nets. Section 8 describes an approach to model check properties of (Dynamic) Feature Nets. Section 9 surveys related work, and Section 10 concludes the paper.

2 Software Product Line Modelling Challenge

We illustrate the modelling challenge in software product line engineering using an example of a software product line of coffee vending 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 selected 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 approach 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.

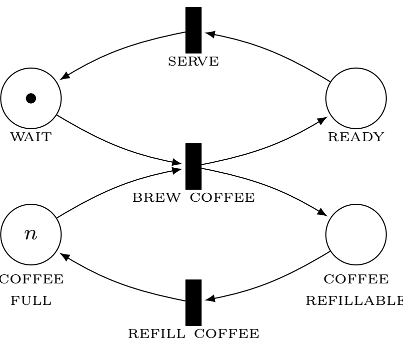


Fig. 1: Petri net model of a basic coffee machine that can only dispense coffee. Labels on places indicate states of the system; labels on transitions indicate its behaviour.

Petri nets [30] are used to specify how systems behave. Fig. 1 presents an example of a Petri net for a coffee machine. This has a capacity for n coffee servings; it can brew and dispense coffee, and refill the machine with new coffee supplies. If we now add an optional *Milk* feature, so that the machine can also add milk to a coffee serving, we need to adapt the Petri net, not only to include the functionality of adding milk, but also to be able to control whether or not this feature is present in the resulting software product.

To address the challenge of modelling a software product line with multiple features, which may or may not be included in any given product, we first introduce transition-labelled Feature Nets. Feature Net transitions are annotated with *application conditions* [34], which are propositional formula over features that reflect when the transition is enabled. Later we introduce a variation of Feature Nets in which applica-

tion conditions are placed on arcs, rather than transitions, called arc-labelled Feature Nets.

One advantage of both transition-labelled and arc-labelled Feature Nets is that they enable the *superposition* of the behaviour of the various products (given by different feature selections) in the same model.

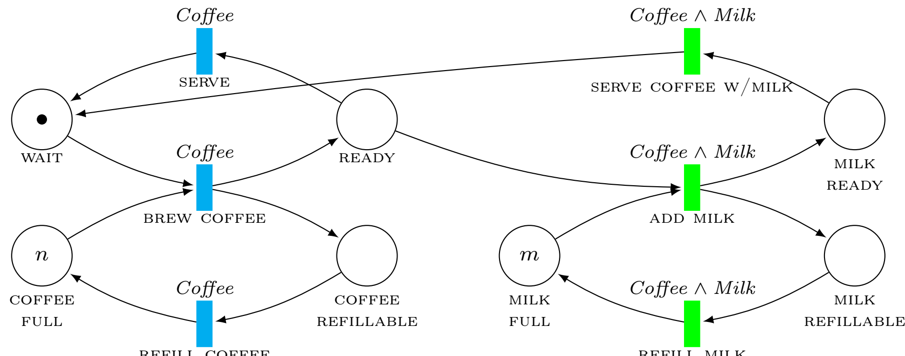


Fig. 2: Transition-labelled FN of the product line with variants $\{\{Coffee\}, \{Coffee, Milk\}\}$ showing each product in its initial state. Each transition has an application condition attached (label above transitions). Colour is used to visually group transitions by application conditions.

Fig. 2 exemplifies a transition-labelled FN of a coffee machine with a milk reservoir. It considers a product line whose products are over the set of features $\{Coffee, Milk\}$, where *Coffee* is compulsory and *Milk* is optional.[1] 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 Fig. 1, after removing unreachable places.

Fig. 3 exemplifies an arc-labelled Feature Net of a similar coffee machine SPL. The application condition above each arc reflects that the arc is present only when the condition evaluates to true. Only then does the arc affect behaviour. If the condition is false, the arc has no effect on behaviour. Consequently, the three transitions on the left-hand side can only fire when the *Coffee* feature is present; the two transitions on the right-hand side can be taken only when the feature *Milk* is present. Observe that the restriction of this example net to the transitions that can fire for feature selection $\{Coffee\}$ is, again, exactly the Petri net in Fig. 1, after removing unreachable places.

Arc-labelled Feature Nets have advantages over transition-labelled Feature Nets when it comes to supporting a modular approach to modelling. This will become clear in Section 6.1, where a composition technique for Feature Nets is proposed.

[1] This structural information is not specified by the FN. The features included in the SPL and their inter-dependencies are typically specified in a feature model. Feature Nets establish the connection to feature models by way of application conditions.

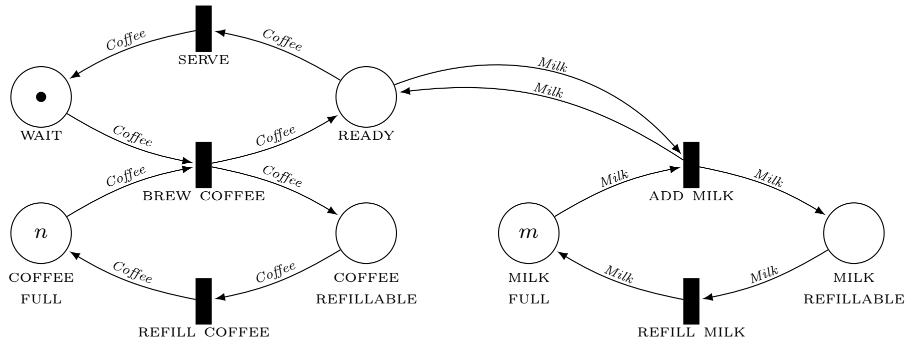


Fig. 3: Arc-labelled FN of the product line $\{\{Coffee\}, \{Coffee, Milk\}\}$. Each arc has an application condition attached.

3 Petri Nets

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, $S(x) > 0$ 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, define $M_1 \odot M_2$ as $(M_1 \odot M_2)(x) = M_1(x) \odot M_2(x)$.

To ground our theory, we recall the terminology and notation surrounding Petri nets [15].

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 $M \geq {}^\bullet t$ with ${}^\bullet t$ regarded as a multiset.

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:

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

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.

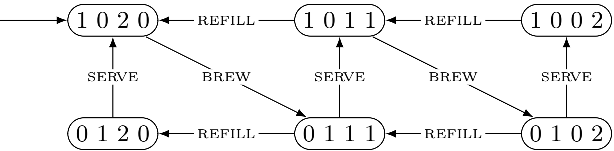
For example, the following trace is part of the behaviour of the coffee machine Petri net illustrated in Fig. 1 (the tuples represent markings of the places listed on the left).

$$\begin{array}{r} \text{WAIT} \\ \text{READY} \\ \text{COFFEE FULL} \\ \text{COFFEE REFILLABLE} \end{array} \begin{pmatrix} 1 \\ 0 \\ n \\ 0 \end{pmatrix} \xrightarrow{\text{BREW COFFEE}} \begin{pmatrix} 0 \\ 1 \\ n-1 \\ 1 \end{pmatrix} \xrightarrow{\text{SERVE}} \begin{pmatrix} 1 \\ 0 \\ n-1 \\ 1 \end{pmatrix}$$

To verify properties over a Petri net it is usually more convenient to represent all possible traces in a more compact way, using *reachability graphs*. The reachability graph of a Petri net has markings as nodes, transitions as edges, and an initial node given by the initial marking. Furthermore, only markings that can be reached from the initial marking are represented in the reachability graph. Traditional model-checkers can then be used to analyse reachability graphs.

Definition 9 (Reachability graph) Let the *reachability set* of a Petri net $N = (S, T, R, M_0)$ be the smallest set $\mathsf{Reach}(N)$ that contains M_0 and all markings M_n such that $M_0 \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} M_n$ is a trace of N, for some transitions $t_0, \ldots, t_{n-1}$. The *reachability graph* of N is the tuple $G = (\mathsf{Reach}(N), E, T, M_0)$ where $\mathsf{Reach}(N)$ are the nodes of the graph, $E \subseteq \mathsf{Reach}(N) \times T \times \mathsf{Reach}(N)$ are the edges between markings such that $(M, t, M') \in E$ iff $M \xrightarrow{t} M'$, T are the transitions of N, and M_0 is the initial state.

Traversals of the reachability graph of a Petri net N correspond to traces of N. The example from Fig. 1 has the following reachability graph when $n = 2$, where $(a\ b\ c\ d)$ represents the marking $\{\text{WAIT} \mapsto a, \text{READY} \mapsto b, \text{COFFEE FULL} \mapsto c, \text{COFFEE REFILLABLE} \mapsto d\}$.



4 Transition-Labelled Feature Nets

Transition-labelled Feature Nets are a Petri net variant used to model the behaviour of an entire software product line. For this purpose, transition-labelled FN have *application conditions* [34] 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.

Throughout this section, the term Feature Net (FN) refers to a transition-labelled Feature Net. We define Feature Nets and give their semantics. We present two semantic accounts of FN. First, when a set of features is selected, an FN *directly* models the behaviour of the product corresponding to the feature selection. Second, by *projecting* an FN 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.

Definition 10 (Application condition [34]) An *application condition* φ is a propositional formula over a set of features F, defined by the following grammar:

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

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

Definition 11 (Satisfaction of application conditions) Given an application condition $\varphi \in \Phi_F$ and a set of features $FS \subseteq F$, called a *feature selection*, we say that FS satisfies φ, written as $FS \models \varphi$, defined as follows:

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

After formally recalling Petri nets and application conditions, we are now in the position to introduce Feature Nets.

Definition 12 (Feature Net) A Feature Net is a tuple $N = (S, T, R, M_0, F, f)$, where (S, T, R, M_0) is a Petri net, F is a 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, we write φ_t. For conciseness, we say that a feature selection FS satisfies transition t whenever $FS \models \varphi_t$.

4.1 Semantics of Feature Nets

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

Definition 13 (Transition occurrence rule for FN) Given a Feature Net $N = (S, T, R, M_0, F, f)$ 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:

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

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 FN is used to define traces that describe the FN's behaviour in the same way as Petri nets.

Definition 14 (FN Trace) Given a Feature Net $N = (S, T, R, M_0, F, f)$ 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 FN, there is a set of traces representing the behaviour of the FN for each feature selection.

Definition 15 (FN behaviour for a given feature selection) Given a Feature Net $N = (S, T, R, M_0, F, f)$ 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 FN.

Definition 16 (FN Behaviour) Given a Feature Net $N = (S, T, R, M_0, F, f)$, we define $\mathrm{Beh}(N)$ to be the combined set of behaviours for all feature selections over F:

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

4.2 Projection-based Semantics of FN

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

Definition 17 (Projection) Given a Feature 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.

For example, by projecting the FN of the product line $\{\{\mathit{Coffee}\}, \{\mathit{Coffee}, \mathit{Milk}\}\}$ (Fig. 2) onto the feature selection $\{\mathit{Coffee}\}$, the application condition *Coffee* (on transitions SERVE, BREW COFFEE and REFILL COFFEE) evaluates to true, while the application condition $\mathit{Coffee} \wedge \mathit{Milk}$ (on SERVE COFEEE W/MILK, ADD MILK and REFILL MILK) evaluates to false. Hence, the latter transitions are removed, along with unreachable places. The result is the Petri net depicted in Fig. 1.

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 Net N and $FS \subseteq F$, then:*

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

By projecting $\mathrm{Beh}(N, FS)$ onto the feature selection FS, the feature selection is removed from the traces of N's behaviour.

Proof ($\subseteq$) We show that every trace $\sigma \in \mathrm{Beh}(N, FS) \downarrow FS$ 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 15, $FS \models \varphi_t$, and by Definition 17 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 17 $FS \models \varphi_t$. Hence, by Definition 15 we conclude that also $(M, FS) \xrightarrow{t} (M', FS)$. □

4.3 Reachability Analysis

The reachability graph of a Petri net represents the markings reachable from the initial marking by firing of transitions (c.f. Definition 9). In a Feature Net transitions have an associated application condition that influences their behaviour. The reachability graph of a Feature Net is therefore also extended with application conditions, into what we call a *variable reachability graph*.

Definition 18 (Variable reachability graph) Let the *reachability set* of a Feature Net $N = (S, T, R, M_0, F, f)$ be the smallest set $\mathsf{Reach}(N)$ that contains M_0 and all markings M_n such that $(M_0, FS) \xrightarrow{t_0} \cdots \xrightarrow{t_{n-1}} (M_n, FS)$ is a trace of N, for some transitions $t_0, \ldots, t_{n-1}$ and a feature selection FS. The *variable reachability graph* of N is the tuple $G = (\mathsf{Reach}(N), E, T, F, f, M_0)$ where $\mathsf{Reach}(N)$ are the nodes of the graph, $E \subseteq \mathsf{Reach}(N) \times T \times \mathsf{Reach}(N)$ are the edges between markings such that $(M, t, M') \in E$ iff there is a feature selection FS where $(M, FS) \xrightarrow{t} (M', FS)$, T is the set of transitions of N, F is the set of features of N, f associates each transition from T to an application condition over F, and M_0 is the initial state.

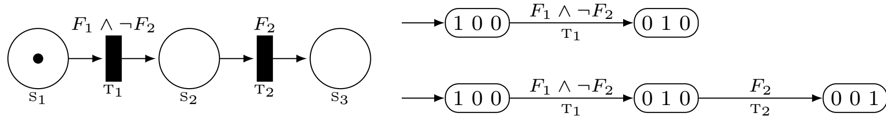


Fig. 4: A Feature Net (left) and its variable reachability graphs (right).

We will also consider a variation of this definition of a variability graph, which we call a *relaxed variable reachability graph*. This relaxed variant is intended to be easier to calculate, while still being accurate enough for model checking purposes. Fig. 4 presents the encoding of a simple example into these two variability graphs. The labels on the edges include not only the transitions but also the associated application condition, given by the function f included in the graph definitions. The top reachability graph was obtained based on Definition 18, while the second, relaxed version includes an extra node (0 0 1) that can never be reached from the initial marking if the feature selection is fixed statically (no feature selection can simultaneously satisfy the application conditions of T_1 and T_2). Relaxed variability graphs consider all feature selections at any given state, therefore including some states that are unreachable when the feature selection is fixed a priori. A relaxed variability graph has two main advantages. First, it is simpler to build, by ignoring the feature selections and just including satisfiable application conditions. Second, it includes transitions and states that are reachable in a more dynamic environment where the feature selection can change at runtime, which will be exploited in the next section.

Definition 19 (Relaxed variable reachability graph) Let the *relaxed reachable set* of a Feature Net $N = (S, T, R, M_0, F, f)$ be the smallest set $\mathsf{Reach}_\mathsf{R}(N)$ such that $M_0 \in \mathsf{Reach}_\mathsf{R}(N)$, and if $M \in \mathsf{Reach}_\mathsf{R}(N)$ and $(M, FS) \xrightarrow{t} (M', FS)$ for some M', FS and t, then $M' \in \mathsf{Reach}_\mathsf{R}(N)$. The *relaxed variable reachability graph* of N is the tuple G_R built in the same way as the variable reachability graph of N (cf. Definition 18), using $\mathsf{Reach}_\mathsf{R}(N)$ instead of $\mathsf{Reach}(N)$ as the set of nodes.

The results regarding the semantics-preserving projection of Feature Nets onto Petri nets are now lifted to reachability graphs. Intuitively, projecting a variable reachability graph of a Feature Net N onto a feature selection FS yields a reachability graph. The same graph is obtained based on the projection of N onto FS. The definition of the projection of reachability graphs ensures that the above equality always holds.

Definition 20 (Graph projection) Given a variable reachability graph (relaxed or not) $G = (Ns, E, T, F, f, M_0)$ and a feature selection $FS \subseteq F$, the *projection* of G onto FS, denoted $G{\downarrow}FS$, is a reachability graph (Ns, E', T', M_0), with $E' = \{(n_1, t, n_2) \in E \mid FS \models \varphi_t\}$, and $T' = \{t \in T \mid FS \models \varphi_t\}$.

As it stands, projecting a Feature Net and deriving its reachability graph is not necessarily the same as deriving the variable reachability graph of the Feature Net and projecting it with the same feature selection. Let $G_{\pi \to r}$ be the reachability graph from the first approach, and $G_{r \to \pi}$ the one from the second approach. These only differ in the set of unreachable nodes and transitions: $G_{r \to \pi}$ has only a minimal set of nodes and transitions, while $G_{\pi \to r}$ can have some nodes and transitions that are not connected

to the initial node. These unconnected elements stem from the way the reachability graph is projected, that is, by discarding unused transitions.

However, $G_{r\to\pi}$ and $G_{\pi\to r}$ are *behaviourally equivalent*, that is, the set of possible traces from the initial node is the same in both graphs. This result holds for both variants of variable reachability graphs. For the purpose of model checking, for example, this is enough. We do not show this equivalence between $G_{r\to\pi}$ and $G_{\pi\to r}$, which follows a reasoning similar to Theorem 1.

Model checking Feature Nets using Petri nets. We identify three different approaches for model checking a Feature Net N: (1) a naive approach based on Petri nets, (2) an optimised one based on variable reachability graphs, and (3) a more relaxed one based on modal transition systems. The former approach consists of encoding N into a traditional Petri net, where a variety of tools and techniques for their analysis exist [30]. This encoding is discussed in more detail in Section 7, and a concrete approach for analysing Petri nets is discuss in Section 8.1. The basic idea of this encoding is that each feature can be specified as a Petri net with two places denoting the presence of the feature, and connected to the Petri net describing the system behaviour. The main disadvantage of this approach is the need to create a potentially large number of Petri nets—one for each selection of features—and the potential need to expand the number of transitions, which will be discussed when presenting the encoding into Petri nets. The combined Petri net is in turn used for checking the desirable properties.

Model checking Feature Nets using featured transition systems. The second approach relies on analysing relaxed variable reachability graphs. A relaxed variable reachability graph can be built using the following simple algorithm. Denote the initial marking as the first node of the graph. Add nodes corresponding to all markings reachable in one step from the initial marking (by firing any one transition, ignoring the satisfaction condition). Include edges corresponding to the fired transitions and annotate them with the respective application conditions. Repeat above steps for the added nodes until no new markings are found. The resulting graph can be seen as a *featured transition system* (FTS) [10], that is, a labelled transition system whose transitions are labelled by application conditions. This can be checked using dedicated FTS model checkers such as SNIP/ProVeLines [8]. The problem of producing and analysing a reachability graph from a Feature Net, seen as an FTS, is addressed with the mCRL2 toolset [12] in Section 8.2.

Model checking Feature Nets using modal transition systems. The third approach to model check properties of feature nets uses *modal transition systems* (MTS). An MTS can be seen as a special kind of labelled transition system where edges are marked either as *required* or as *optional*. In the context of software product lines, MTS have been used to represent families of systems, one for each choice of optional edges [18, 16]. In our case, MTS can be used to model check Feature Nets by generating special reachability graphs (Definition 9) where edges are marked as required or optional, and use MTS tools to infer properties over it. For example, one can generate an MTS from a Feature Net and a given set PL of desired feature selections; in our running example, $PL = \{\{\mathit{Coffee}\}, \{\mathit{Coffee}, \mathit{Milk}\}\}$. The MTS is obtained by calculating the reachability graph of the Feature Net without the application conditions, and by annotating each transition t of the Feature Net as required if $\forall FS \in PL \cdot FS \models \varphi_t$, as optional if $\exists FS \in PL \cdot FS \models \varphi_t$, and discarded otherwise. When compared to the previous

two approaches, MTS convey less information because they abstract away the valid combinations of optional edges. Consequently, verifying that a path exists in the MTS does not guarantee the existence of the same path for the reachability graph projected to some feature selection. Interesting properties than can still be verified using MTS include the guarantees that certain sequences of actions are possible in all considered feature selections, or the unreachability of certain markings.

5 Dynamic Feature Nets

Dynamic Software Product Lines (DSPL) is an area of research concerned with runtime variability of systems [23]. DSPL is an umbrella concept that addresses dynamic reconfiguration of products (i.e. features are added and removed 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.

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

5.1 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 enabled/disabled temporarily based on the connected hardware modules. 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.

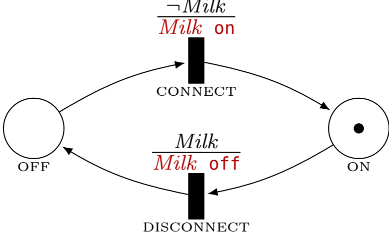


Fig. 5: DFN modelling the ability to connect/disconnect a feature at runtime.

In our example, switching the *Milk* feature on and off can be modelled by the DFN in Fig. 5, as an independent addition to the model in Fig. 2. Associated to the DISCONNECT transition is the *update* expression "`Milk off`". By firing the DISCONNECT

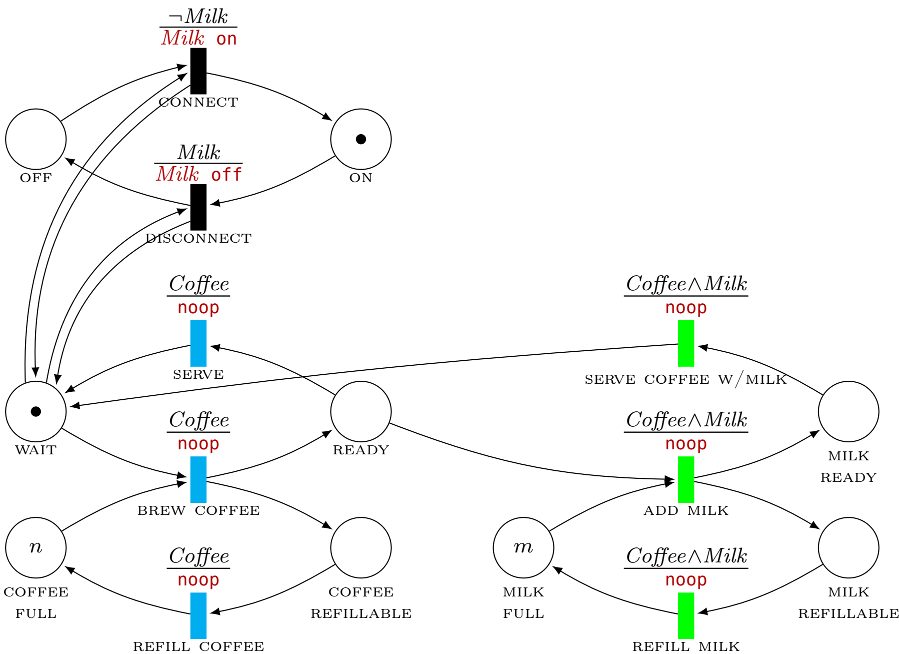


Fig. 6: DFN (initial state) of a dynamically reconfigurable product line. Whenever transition DISCONNECT fires, feature *Milk* is switched off, disabling all transitions that are conditioned on *Milk*. It is enabled again by firing CONNECT.

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 Fig. 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 of the coffee machine. Alternatively, we can assume that the reconfiguration of features depends on the functional model. Fig. 6 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.

5.2 Definition

We extend the definition of Feature 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 Nets* (DFN). DFN extend Feature 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 enables a more concise description of SPLs, without adding expressive power with respect to Petri nets (see Section 7 for a justification of this claim). We now define update expressions before formalising DFN.

Definition 21 (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 22 (Dynamic Feature Net) A DFN is a tuple $N = (S, T, R, M_0, FS_0, F, f, u)$, where (S, T, R, M_0, F, f) is an FN, $FS_0 \subseteq F$ is the initial feature selection and u is a function $T \to U_F$, associating each transition with an update from U_F.

The initial marking M_0 together with the initial feature selection FS_0 define the initial state of the DFN. We write u_t to denote the update expression $u(t)$ associated with a transition t.

5.3 Semantics

Definition 23 (DFN transition occurrence) Given a Dynamic Feature Net $N = (S, T, R, M_0, FS_0, F, f, u)$, 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 24 (DFN trace) Given a DFN $N = (S, T, R, M_0, FS_0, F, f, u)$, the behaviour the net exhibits by assuming a sequence of states $(M_0, FS_0) \ldots (M_n, FS_n)$, 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 FN.

Definition 25 (DFN Behaviour) Given a DFN $N = (S, T, R, M_0, FS_0, F, f, u)$, we define $\text{Beh}(N)$ to be the set of all traces starting with the initial state (M_0, FS_0).

For example, the following trace is an element of the behaviour of the DFN illustrated in Fig. 6 (tuples represent markings and the sets below are feature selections).

$$\begin{array}{r}\text{OFF}\\ \text{ON}\\ \text{WAIT}\\ \text{READY}\\ \text{M. READY}\\ \text{C. FULL}\\ \text{C. REFILLABLE}\\ \text{M. FULL}\\ \text{M. REFILLABLE}\end{array} \underset{\{\mathit{Coffee}, \mathit{Milk}\}}{\begin{pmatrix}0\\1\\1\\0\\0\\n\\0\\m\\0\end{pmatrix}} \xrightarrow{\text{DISCONNECT}} \underset{\{\mathit{Coffee}\}}{\begin{pmatrix}1\\0\\1\\0\\0\\n\\0\\m\\0\end{pmatrix}} \xrightarrow{\text{CONNECT}} \underset{\{\mathit{Coffee}, \mathit{Milk}\}}{\begin{pmatrix}0\\1\\1\\0\\0\\n\\0\\m\\0\end{pmatrix}}$$

5.4 Reachability Analysis

DFN incorporate a feature selection that can change dynamically when transitions fire. Thus, when building the reachability graph of a DFN, each node (which represents a state of the DFN) will also have a feature selection associated with it.

Definition 26 (Dynamic reachability graph) Let the *dynamically reachable set* of a Dynamic Feature Net $N = (S, T, R, M_0, FS_0, F, f, u)$ be the smallest set $\mathsf{Reach}(N)$ such that $(M_0, FS_0) \in \mathsf{Reach}(N)$, and $(M_i, FS_i) \in \mathsf{Reach}(N)$ if $(M_0, FS_0) \xrightarrow{t_0} \cdots \xrightarrow{t_i} (M_i, FS_i)$. The *dynamic reachability graph* of N is the tuple $G = (\mathsf{Reach}(N), E, T, M_0, FS_0, F, f)$ where $\mathsf{Reach}(N)$ are the nodes of the graph, $E \subseteq \mathsf{Reach}(N) \times T \times \mathsf{Reach}(N)$ are the graph's edges, T is the set of transitions of N, (M_0, FS_0) is the initial state of N, F is the set of features of N, and $f : E \to \Phi_F$ is a function associating each edge with an application condition over F.

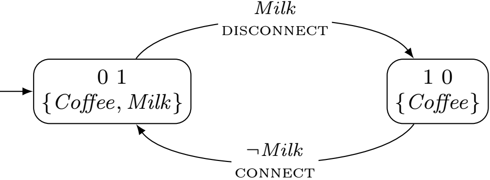


Fig. 7: Dynamic reachability graph of the DFN depicted in Fig. 5.

For example, the DFN from Fig. 5 has the dynamic reachability graph depicted in Fig. 7. To model check DFNs one can use a traditional model checker, such as SPIN [24] or mCRL2 [12]. This is explained in detail in the end of this paper (Section 8.3), where the reachability graph of the DNF from Fig. 6 is calculated and analysed. In a nutshell, states can be encoded as tuples of integers, each representing a marking together with

the feature selection, and labels as transitions. Note that the approach based on variable reachability graphs (c.f. Section 4.3) cannot be used for Dynamic Feature Nets. Even though a dynamic reachability graph can be seen as a featured transition system [10] by discarding the information regarding feature selection updates, the semantics would be different since the feature selections cannot be modified during the execution of the system.

6 Arc-Labelled Feature Nets

A Feature Net (FN) is a Petri net variant used to model the behaviour of an entire software product line. Arc-labelled Feature Nets are a FN variant that have application conditions attached to their *arcs*. As defined in Section 4, an application condition is a propositional logical formula over a set of features. For arc-labelled Feature Nets, it describes the feature combinations to which the arc applies. If the application condition is false for a given feature selection, it is as if the arc were not present. Arc-labelled Feature Nets allow a technique for constructing larger Feature Nets from smaller ones to model the addition of new features to an SPL. Along with presenting the composition technique, we provide correctness criteria for ensuring that the resulting composition preserves the behaviour of the original model(s). Arc-labelled Feature Nets are as expressive as transition-labelled FNs, as will be shown in Section 7. The main difference is that they allow a finer grained association with features, which often results in more concise models.

Throughout this section, whenever the term Feature Net is used, it refers to the arc-labelled variant. We define arc-labelled Feature Nets and their behaviour by adapting the definition of Feature Nets described in Section 4, where application conditions apply to transitions instead of arcs.

Definition 27 (Feature Net) A Feature Net is a tuple $N = (S, T, R, M_0, F, f)$, 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*. The elements of R are called *arcs*. Finally, F is set of features and $f : R \to \Phi_F$ is a function associating each arc with an application condition from Φ_F. Note that f is different from the function f that associates transitions with application conditions in transition-labelled Feature Nets (Definition 12).

Without f and F, a Feature Net is just a Petri net. Sometimes we omit the initial marking M_0. The function f determines a node's pre- and post-set, defined below.

Definition 28 (Marking of a Feature Net) A marking M of an FN (S, T, R, F, f) is a multiset over S. A place $s \in S$ is *marked* iff $M(s) > 0$.

The pre- and post-sets of arc-labelled FNs depend on the feature selection FS, which determines whether an arc is present or not. The following definition takes this into account. Note that this is different from transition-labelled Feature Nets, where arcs are fixed.

Definition 29 (Pre-sets and post-sets) Given a node x of a Feature Net and a feature selection FS, the set ${}^{(FS)}x = \{y \mid (y, x) \in R, FS \models f(y, x)\}$ is the *pre-set* of x and the set $x^{(FS)} = \{y \mid (x, y) \in R, FS \models f(x, y)\}$ is the *post-set* of x.

Definition 30 (Enabling) Given a feature selection FS, a marking M *enables* a transition $t \in T$ if it marks every place in ${}^{(FS)}t$, that is, if $M \geq {}^{(FS)}t$.

We now define the behaviour of Feature Nets for a given feature selection.

Definition 31 (Transition occurrence) Let $N = (S, T, R, M_0, F, f)$ be a Feature Net and $FS \subseteq F$ a feature selection. A transition $t \in T$ *occurs*, leading from a state with marking M_i to a state with marking M_{i+1}, denoted $M_i \xrightarrow{t,FS} M_{i+1}$, iff the following two conditions are met:

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

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

The transition rule for FN is used to define traces that describe the FN's behaviour. We now define the semantics of a Feature Net by projecting it onto a Petri net for a given feature selection.

Definition 32 (Projection) Given a Feature 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 $R' = \{(x, y) \mid (x, y) \in R, FS \models f(x, y)\}$.

One *projects* N onto a feature selection FS by evaluating all application conditions $f(x, y)$ with respect to FS for all arcs $(x, y) \in R$. If FS does not satisfy $f(x, y)$, then arc (x, y) is removed from the Petri net.

The behaviour of a Feature Net is the union of the behaviour of the Petri nets obtained by projecting all possible feature selections. The behaviour of a Petri net $N = (S, T, R, M_0)$ is given by the set of all of its traces [20], written $\mathrm{Beh}(N) = \{M_0 \xrightarrow{t_1} \cdots \xrightarrow{t_s} M_n \mid M_i \subseteq S, i \in 1..n, M_{i-1} \xrightarrow{t_i} M_i\}$, and does not include application conditions nor feature selections.

Definition 33 (FN Behaviour) Given an FN $N = (S, T, R, M_0, F, f)$, we define $\mathrm{Beh}(N)$ as follows:

$$\mathrm{Beh}(N) = \bigcup_{FS \subseteq F} \mathrm{Beh}(N \downarrow FS).$$

6.1 Modular Modelling

For a modelling formalism to be useful in practice, it needs to facilitate modular development techniques. This is especially important for modelling software product lines: a single SPL model combines the behaviour of a set of different systems, which are often too complex to develop simultaneously.

Modular approaches include top-down techniques, where initially an abstract model is sketched and more details are added incrementally, and bottom-up approaches, where subsystems are modelled separately and later plugged together to a global model. Petri nets support both approaches [20]. In the following we propose a bottom-up composition technique for Feature Nets. It is based on the idea of modelling features of the system individually and then combining them to obtain a model of the entire SPL. Our approach starts by building a model of the *core* system that is the behaviour which is common to all products of the SPL. Optional features are modelled as separate

nets, which also specify how they interact with the core through an *interface*. Core and additional features are then composed stepwise, by incrementally applying each feature to the core. We show how this technique can be applied to modularly specify a coffee machine product line from the three features *Coffee*, *Payment* and *Milk*.

Feature Net Composition

We devise a modular modelling approach in which features (or parts thereof) are first expressed as separate FNs. A feature's interaction with the rest of the system (the *core*) is modelled using an *interface*. Features are modelled separately in such a way that they can be attached to the core, in order to incrementally build a larger model. The interface simulates the behaviour of the core that the features are designed to be plugged into. A feature modelled using this technique can be seen as a partially specified model of the entire SPL, where the feature's behaviour is fully specified, whereas everything else is underspecified. Composition then amounts to connecting the interface to the core to obtain a specification of the combined system. We call a feature net with such an interface a *delta feature net*, as it provides a behavioural delta to the core (i.e., it adds or removes behaviour).

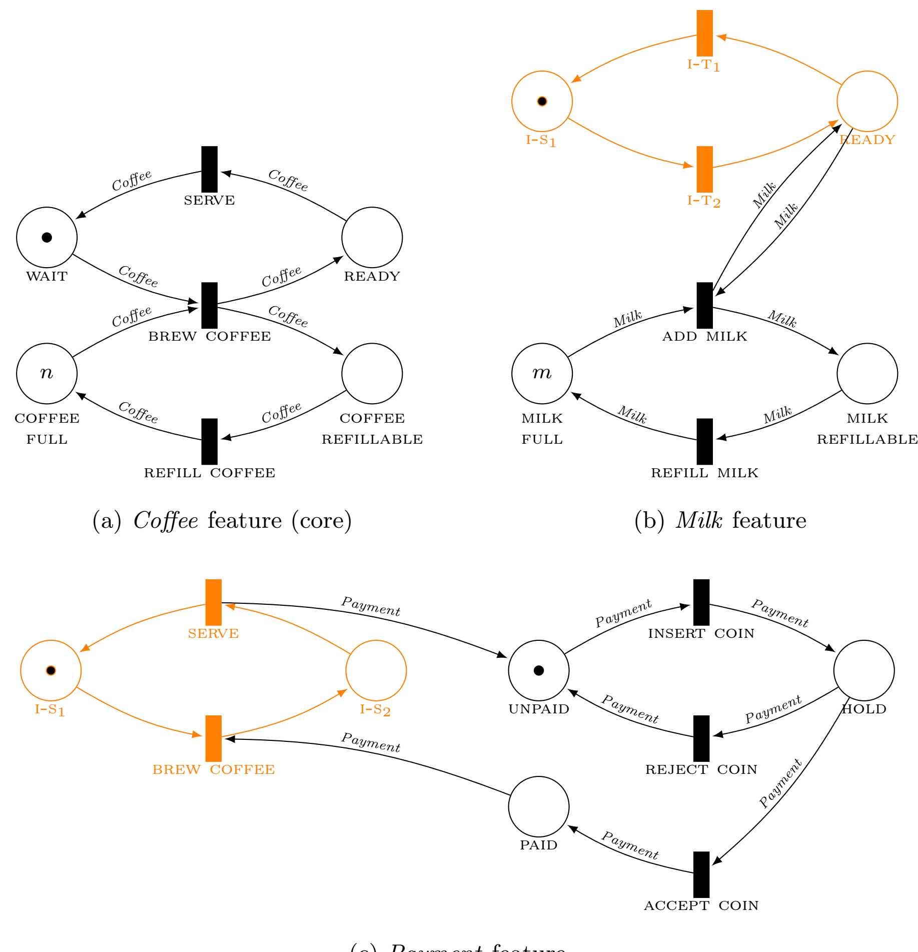
The three features of our example coffee machine are modelled as separate FNs (Fig. 8). Apart from when a feature's behaviour is self-contained (such as the *Coffee* net in Fig. 8a) it will typically interact with other features that are part of the larger system. To faithfully model such interactions we include an interface. Interfaces (highlighted in orange in Fig. 8) abstract the behaviour of the core. The interface will also be used to show that the individual net exposes the same behaviour as it does when it is part of the combined system. For example, the model of *Milk* in Fig. 8b reflects the fact that adding milk depends on a state of the system in which a cup of fresh coffee is available. The larger system is represented abstractly by the highlighted interface, which models the availability of coffee in the place READY; a token in this place would denote a state in which a freshly brewed cup of coffee is available. Similarly, Fig. 8c models the fact that after a payment has been accepted, the overall system is able to BREW COFFEE, and after serving the coffee, the system goes back to an UNPAID state. Note that interfaces are in general not limited to the composition with a particular core, but can be attached to any core that they are applicable to.

Constructing a model of the whole SPL is done by stepwise applying the delta nets of each feature to a core model. The intuition behind delta net application is that each interface is replaced with a more complex Feature Net. Nodes (transitions and places) are identified by their names. Therefore, nodes with the same name that appear in different nets are considered the same and serve as reference when replacing the interface with the core net. In our example, the first step could be to refine *Payment*'s interface by replacing it with the Feature Net for *Coffee*. In a second step, the feature *Milk* is refined by replacing its interface with the net obtained in the previous step.

We now formally introduce the application of a delta net to a core net.

Definition 34 (Delta Feature Net) A delta Feature Net N is a FN with a designated interface (S_I, T_I), denoted $N = (S, T, R, F, f, S_I, T_I)$, where $S_I \subseteq S$ and $T_I \subseteq T$.

Delta Feature Nets specify the behaviour of features designed to be added to a larger system. A sequence of delta FN is combined with a stand-alone FN, the *core*, by sequentially *applying* each delta net to the core. Delta nets include an interface,



(a) *Coffee* feature (core)

(b) *Milk* feature

(c) *Payment* feature

Fig. 8: Individual Feature Nets modelling the features *Coffee*, *Milk* and *Payment* of a product line of coffee machines. Interfaces are highlighted in orange. Arcs without labels have the application condition *true*.

which models interactions with the core. Such interactions are modelled by transitions or places common to both core and delta net.

Definition 35 (Delta Net Application) Let $N = (S, T, R, F, f)$ be a Feature Net and $D = (S_d, T_d, R_d, F_d, f_d, S_I, T_I)$ a delta Feature Net with $S \cap S_d \neq \emptyset$. The *application* of D to N results in a net $N' = (S', T', R', F', f')$, written as $N \oplus D$, where

$$
\begin{aligned}
S' &= (S_d \setminus S_I) \cup S \\
T' &= (T_d \setminus T_I) \cup T \\
R' &= \{(s,t) \in (R \cup R_d) \mid s \in S', t \in T'\} \\
&\quad \cup \{(t,s) \in (R \cup R_d) \mid t \in T', s \in S'\} \\
F' &= F \cup F_d \\
f' &= f \oslash f_d
\end{aligned}
$$

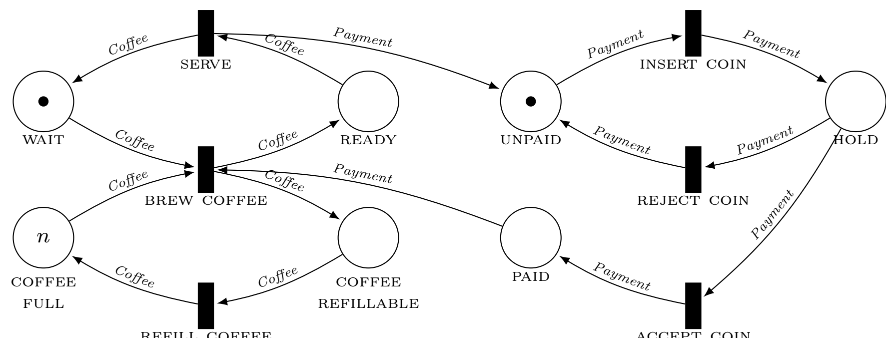


Fig. 9: A software product line over feature set $\{Coffee, Payment\}$ obtained by applying the delta net *Payment* (Fig. 8c) to the core net modelling *Coffee* (Fig. 8a).

and

$$(g \oslash h)(arc) = \begin{cases} g(arc) & \text{if} \quad arc \in \mathrm{dom}(g) \wedge arc \notin \mathrm{dom}(h) \\ h(arc) & \text{if} \quad arc \notin \mathrm{dom}(g) \wedge arc \in \mathrm{dom}(h) \\ g(arc) \wedge h(arc) & \text{if} \quad arc \in \mathrm{dom}(g) \cap \mathrm{dom}(h). \end{cases}$$

When applying a delta net to a core, the interface is dropped and the two nets are "fused" along their common nodes. The arcs that previously connected the delta net interface now connect the core. The applicability of a delta net is limited to certain cores. Let S_B and T_B represent the border of the interface, that is, $S_B = \{s \in S_I \mid \exists t \in T_d \setminus T_I : (s,t) \in R'\}$ and $T_B = \{t \in T_I \mid \exists s \in S_d \setminus S_I : (t,s) \in R'\}$. A delta net is applicable to a core net if the border of the interface is preserved, that is, if $S \cap S_d = S_B$ and $T \cap T_d = T_B$.

We show how delta net application is used to build a model of the example coffee machines SPL. Starting with the separate sub-models in Fig. 8, delta nets are applied stepwise to a growing core. First, a model with the two features *Coffee* and *Payment* is composed by applying the delta net from Fig. 8c to the core shown in Fig. 8a. These nets have the two transitions SERVE and BREW COFFEE in common. The result after applying the delta Feature Net is the new core Feature Net shown in Fig. 9. In a second step, we add the *Milk* behaviour by applying the Feature Net in Fig. 8b to the core obtained in the previous step. These two nets have the place READY in common. The result after delta net application is the model shown in Fig. 10. Note that the order in which we apply the two delta nets does not matter in this case, because neither feature (*Milk* or *Payment*) depends on the other. In general, features can depend on other features. This would be reflected by the design of their interfaces, effectively restricting the applicability and ensuring that the delta nets can only be applied in a valid order. As a consequence, delta net application is not commutative.

6.2 Behaviour Preservation

When is the application of a delta net D to a core net N *correct*? We consider this application correct if the traces of N and D are in some way the same as the traces of $N \oplus D$, introduced in Definition 35, after projecting onto the transitions of N and D. However, there are various ways to compare these traces. We can consider only the

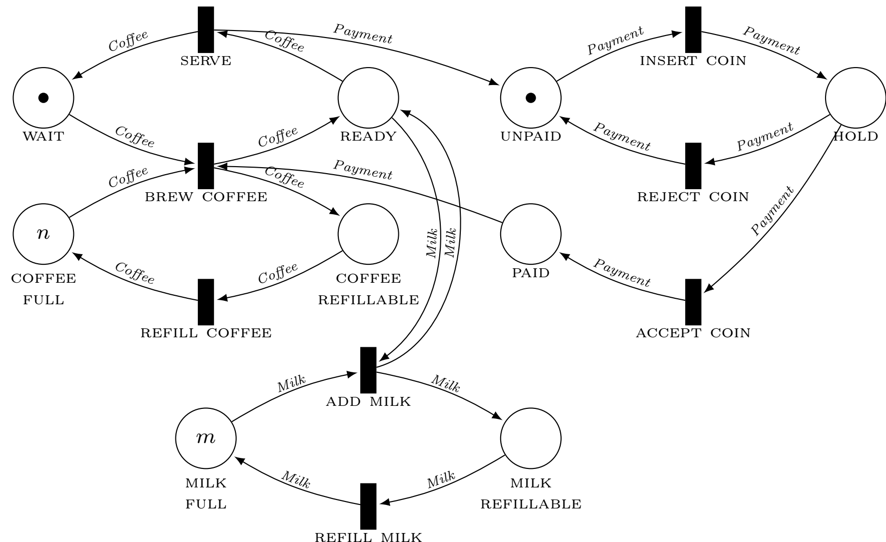


Fig. 10: FN model of an SPL over the feature set $\{Coffee, Payment, Milk\}$ obtained by sequential application of the delta nets for the features *Payment* (Fig. 8c) and *Milk* (Fig. 8b).

features used by the original nets (F_N or F_D) or the features used by the combined net ($F_N \cup F_D$). We can consider the core net N or the delta net D. Finally, we can consider the inclusion of traces of the original net in the combined net or also the inclusion of traces of the combined net in the original net. The three dimensions are summarised as:

- **Original** vs. **combined** features. When comparing the behaviour of one of the original nets with the combined net, we can either consider the combined features in the final net or just the features in one of the original nets.
- **Core** vs. **delta**. We can evaluate the correctness of the core or delta net behaviour, always in comparison to the combined net's behaviour.
- **Liveness**, **safety**, or **both**. Preserving liveness means that a net cannot inhibit behaviour in the other net, while preserving safety means that a net cannot introduce new behaviour to the other net. For example, we say a delta application is safe with respect to the core net N if the traces of the combined net are included in the traces of N, when considering the common transitions.

By choosing different parameters along these dimensions we obtain different notions of correctness. We formulate a parametrised notion of correctness for the application of delta net D to a core net N as follows:

$$\forall FS \subseteq \Theta_F : \mathrm{Beh}(\Theta_N {\downarrow} FS) \;\; \Theta_R \;\; \mathrm{Beh}((N \oplus D){\downarrow} FS) \qquad \text{(param. correctness)}$$

$$\begin{aligned} \Theta_F &\in \{F_N, F_D, F_N \cup F_D\} \\ \Theta_N &\in \{N, D\} \\ \Theta_R &\in \{\subseteq_{Ts}, \supseteq_{Ts}, =_{Ts}\} \end{aligned}$$

Θ_F can be either the full set of features, or the features of either the net N or D; Θ_N can be either the core or the delta net; and Θ_R is a superset, set inclusion or set equivalence relation between the two sets of traces, with respect to a given set of relevant transitions Ts. This set of transitions will be explained in Section 6.3 and made concrete in Sections 6.4, 6.5, and 6.6. When Θ_R is a superset relation, it represents *safety*, since no new traces can be introduced by combining the two nets. On the other hand, a subset relation represents *liveness*, since all traces in the original net are still valid traces in the combined net. When we have both safety and liveness assurances, we say that the behaviour is *preserved*, and instantiate Θ_R to be the equality of the traces with respect to the common transitions.

Not all combinations of these dimensions are desirable in all cases. For example, sometimes we might want to inhibit or extend the behaviour of a core net with respect to the combined set of features, breaking the liveness or safety criteria. However, it seems desirable to preserve this behaviour with respect to the features of the core net. In fact, it is open to debate which combination of these dimensions are ideal. In this paper, we provide sufficient conditions to guarantee:

1. *Preservation* of the behaviour of N with respect to the *original* features ($\Theta_F = F_N$; $\Theta_N = N$; $\Theta_R =$ '$=_{Ts}$')
2. *Preservation* of the behaviour of D with respect to the *combined* features ($\Theta_F = F_N \cup F_D$; $\Theta_N = D$; $\Theta_R =$ '$=_{Ts}$')
3. *Safety* of the behaviour of N with respect to the *combined* features ($\Theta_F = F_N \cup F_D$; $\Theta_N = N$; $\Theta_R =$ '$\supseteq_{Ts}$')

6.3 Mathematical Preliminaries

We defined liveness and safety as inclusion of traces with respect to a relevant set of traces. We formalise this concept below.

Definition 36 (Behaviour inclusion $\subseteq_{Ts}$) Let $N_i = (S_i, T_i, R_i)$ be a pair of Petri nets, for $i \in 1..2$, and Ts be a set of transitions. We say that the behaviour of N_1 is included by the behaviour of N_2 with respect to Ts, written $\text{Beh}(N_1) \subseteq_{Ts} \text{Beh}(N_2)$, if $\text{Beh}(N_1) \upharpoonright Ts \subseteq \text{Beh}(N_2) \upharpoonright Ts$, where $\text{Beh}(N) \upharpoonright Ts = \{tr \upharpoonright Ts \mid tr \in \text{Beh}(N)\}$ and:

$$M \upharpoonright Ts = \varepsilon \qquad (M \xrightarrow{t} tr) \upharpoonright Ts = \begin{cases} t \cdot (tr \upharpoonright Ts) & \text{if } t \in Ts \\ tr \upharpoonright Ts & \text{otherwise.} \end{cases}$$

Similarly, we write $\supseteq_{Ts}$ and $=_{Ts}$ to represent superset inclusion and equality for the transitions in Ts.

Behaviour inclusion between two nets N_1 and N_2 is defined by comparing the transition sequences that both nets are able to perform. The transition sequences of both nets are restricted to transitions from a given set T_s. If the transition sequences of N_1 are a subset of the transition sequences of N_2, we say that the behaviour of N_1 is included in the behaviour of N_2.

We now define *weak bisimulation* between two Feature Nets, which we will use to relate the interface of a delta net with the net to which the delta is applied to, based on the notion of bisimulation described by Schnoebelen and Sidorova [36].

Definition 37 (Weak bisimulation) Let $N_i = (S_i, T_i, R_i, M_{0i}, F_i, f_i)$ be two Feature Nets, for $i \in 1..2$, $\mathcal{M}_i$ the set of markings of N_i, and $\mathcal{B} \subseteq (\mathcal{M}_1 \times \mathcal{M}_2) \cup (T_1 \times T_2)$

a relation over markings and transitions. Recall also the notion of occurrence of transitions introduced in Definition 31. In the following we write $t \in \mathcal{B}$ to denote that t is in the domain or codomain of $\mathcal{B}$. $\mathcal{B}$ is a weak bisimulation if, for any feature selection FS:

1. $M_{01}\ \mathcal{B}\ M_{02}$
2. $\forall (M_1, M_2) \in \mathcal{B}$, if $M_1 \xrightarrow{t_1, FS} M_1'$ and $t_1 \notin \mathcal{B}$, then $M_1'\ \mathcal{B}\ M_2$;
3. $\forall (M_1, M_2) \in \mathcal{B}$, if $M_1 \xrightarrow{t_1, FS} M_1'$ and $t_1 \in \mathcal{B}$, then there exists $t_2 \in T_2$ and M_2' such that $M_2 \xrightarrow{t_2, FS}_{\mathcal{B}} M_2'$, $M_1'\ \mathcal{B}\ M_2'$, and $t_1\ \mathcal{B}\ t_2$;
4. conditions (2) and (3) also hold for $\mathcal{B}^{-1}$;

where $M \xrightarrow{t, FS}_{\mathcal{B}} M'$ denotes that there are n transitions $t_1 \dots t_n$ such that $M \xrightarrow{t_1, FS} \cdots \xrightarrow{t_n, FS} M_n \xrightarrow{t, FS} M'$ and $\forall j \in 1..n : t_j \notin \mathcal{B}$.

If a weak bisimulation exists between N_1 and N_2 we say that they are weakly bisimilar, written $N_1 \approx N_2$.

Example Let C be the Feature Net for the *Coffee* feature (Fig. 8a), and P the delta net dealing with *Payment* (Fig. 8c). Let the interface of P be seen as a Feature Net P_I. It holds that $C \approx P_I$. Furthermore, there exists a weak bisimulation $\mathcal{B}$ that relates the transitions with the same name of the two nets, namely SERVE and BREW COFFEE. More specifically, the relation $\mathcal{B}$ below is a bisimulation, where we write $\mathcal{M}_C$ and $\mathcal{M}_{P_I}$ to denote all the markings of C and P_I, respectively.

$$\{(M, M') \mid M \in \mathcal{M}_C, M' \in \mathcal{M}_{P_I}, M(\textsc{wait}) = 1, M'(\textsc{i-s}_1) = 1\} \cup$$
$$\{(M, M') \mid M \in \mathcal{M}_C, M' \in \mathcal{M}_{P_I}, M(\textsc{wait}) = 0, M'(\textsc{i-s}_1) = 0\} \cup$$
$$\{(\textsc{serve}, \textsc{serve}), (\textsc{brew coffee}, \textsc{brew coffee})\}$$

6.4 Preservation of the core behaviour for the original features

Our first criterion compares the core net with the combined net, considering only the features originally present in the core net. If we require the behaviour of the core net to be *preserved* in the combined net, then their traces must coincide with respect to the transitions in the core net. We formalise this criterion as follows.

Criterion 1 (preservation/core/original) *Let $N = (S, T, R, F, M_0, f)$ be a core net and D a delta net. We say that $N \oplus D$ preserves the behaviour of N for the features in F iff*

$$\forall FS \subseteq F\ :\ \mathrm{Beh}(N{\downarrow}FS) =_T \mathrm{Beh}(N \oplus D{\downarrow}FS).$$

To verify that a delta net application obeys the above correctness criteria, it is sufficient (although not necessary) to verify the following condition. Check that the arcs between the interface and the non-interface nodes of D require at least one 'new' feature to be present. By new feature we mean a feature that is not in F. This syntactic check ensures that, when considering only the features from the core net, the arcs connecting it to the delta net will never be active.

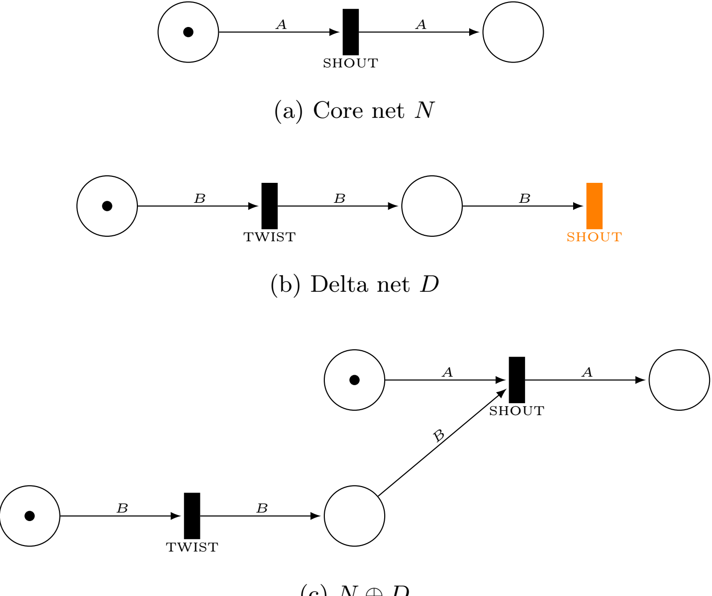
Theorem 2 *Let $D = (S_d, T_d, R_d, M_{0d}, F_d, f_d, S_I, T_I)$ be a delta net, $N = (S, T, R, M_0, F, f)$ a Feature Net, and $R_I \subseteq R_d$ be the set of arcs connecting interface nodes*

$(S_I \cup T_I)$ to non-interface nodes. The behaviour of N is preserved by $N \oplus D$ for the features in F (Criterion 1) if:

$$\forall (x,y) \in R_I \;:\; \forall FS \in F_d \cup F \;:\; FS \models f(x,y) \quad \mapsto \quad FS \cap (F_d \backslash F) \neq \emptyset. \tag{1}$$

Proof Assume that Equation (1) holds for $N \oplus D = N'$. We show that the core behaviour is preserved, i.e., that $\forall FS \subseteq F \cdot \text{Beh}(N \downarrow FS) =_T \text{Beh}(N' \downarrow FS)$. Observe that, for every $FS \subseteq F$, $FS \cap (F_d \backslash F) = \emptyset$. Hence, by assuming Equation (1) we conclude that for every arc $(x,y) \in R_I$, $FS \not\models f(x,y)$. Therefore the traces $t \in \text{Beh}(N \downarrow FS)$ coincide with the traces of $\text{Beh}(N' \downarrow FS)$ with respect to the transitions of N. □

In both our examples of delta application, that is, adding payment to a coffee machine and adding milk to the resulting net, the condition in Equation (1) holds. The intuition is that, for example, when the *Payment* feature is not available, the *Coffee* Feature Net is detached from the *Payment* Feature Net in the combined net. Hence its behaviour is not affected by the *Payment* net and is preserved.



(c) $N \oplus D$

Fig. 11: Example of an FN composition that is correct w.r.t. Criterion 1.

Example The following simple example illustrates this criterion. Fig. 11 shows (a) a core net N with feature set {A}, (b) a delta net D with feature set {B} and (c) the combined net $N \oplus D$ obtained by applying the delta net to the core. Criterion 1 verifies that the net $N \oplus D$ preserves the behaviour of N for the feature selection {A} by using

Definition 36 to compare the behaviour of the two nets $N{\downarrow}\{A\}$ and $N \oplus D{\downarrow}\{A\}$:

$$\begin{aligned} \mathrm{Beh}(N{\downarrow}\{A\}) \upharpoonright \{\textsc{twist}\} &= \{\textsc{twist}\} \\ \mathrm{Beh}(N \oplus D{\downarrow}\{A\}) \upharpoonright \{\textsc{twist}\} &= \{\textsc{twist}\} \\ \implies \mathrm{Beh}(N{\downarrow}\{A\}) &=_{\{\textsc{twist}\}} \mathrm{Beh}(N \oplus D{\downarrow}\{A\}). \end{aligned}$$

To check Criterion 1 we can also use Theorem 2 and simply observe that the application condition on the arc between the interface and the non-interface nodes of D requires the (additional) presence of feature B.

6.5 Preservation of the delta behaviour for the combined features

Our second correctness criterion compares the combined net with the delta net, considering all features in the combined net. In a nutshell, this criterion holds when the delta net behave the same after its interface is replaced by the core net, hence the comparison is based only on the transitions from the delta that are not on the interface. The formal definition follows.

Criterion 2 (preservation/delta/combined) *Let $N = (S, T, R, M_0, F, f)$ be a core net and $D = (S_d, T_d, R_d, M_{0d}, F_d, f_d, S_I, T_I)$ a delta net. We say that $N \oplus D$ preserves the behaviour of D with respect to features from the combined net iff*

$$\forall FS \subseteq F \cup F_d \;:\; \mathrm{Beh}(D{\downarrow}FS) =_{T_d \setminus T_I} \mathrm{Beh}(N \oplus D{\downarrow}FS).$$

As with the correctness Criterion 1, we present a sufficient condition that guarantees the preservation of the Criterion 2. However, as opposed to the previous case, this condition is based on a *semantic property* of the interface and the core net. More specifically, this condition relates the interface of the delta net and the core net that will replace this interface, and imposes extra constraints on their behaviour.

Theorem 3 *Let $D = (S_d, T_d, R_d, M_{0d}, F_d, f_d, S_I, T_I)$ be a delta net, $N_I = (S_I, T_I, R_I, M_{0D}, F_d, f_d)$ be the interface of D, $N = (S, T, R, F, f)$ a (core) Feature Net, and $R_B \subseteq R_d$ denote the arcs connecting interface to non-interface nodes. The behaviour of D is preserved by $N \oplus D$ (Criterion 2) if $N \approx N_I$ and there is a specific weak bisimulation $\mathcal{B}$ between N and N_I such that:*

$$\{(t,t) \mid t \in T \cap T_I\} \subseteq \mathcal{B}, \tag{2}$$

$$\forall s \in S \cap S_I, (M, M') \in \mathcal{B} : M(s) = M'(s), \tag{3}$$

$$\forall (s,t) \in R_B, s \in S_I, (M, M') \in \mathcal{B} \;:\; (M - \{s \mapsto 1\})\; \mathcal{B}\; (M' - \{s \mapsto 1\}) \tag{4}$$

$$\forall (t,s) \in R_B, s \in S_I, (M, M') \in \mathcal{B} \;:\; (M + \{s \mapsto 1\})\; \mathcal{B}\; (M' + \{s \mapsto 1\}) \tag{5}$$

For Equation (4) we assume that, if $M(s) = M'(s) = 0$, then subtracting $\{s \mapsto 1\}$ does not change the markings.

Proof Let $FS \subseteq F \cup F_d$, and $\mathcal{B}$ the bisimulation described by Theorem 3. We write M^N, M^D, and M^I to denote the markings M restricted to the places of N, D, and the interface of D, respectively.

We prove safety, while liveness can be shown in an analogous way, because $\mathcal{B}$ is symmetric. We show by induction the following property. Assume that $tr = t_1 \cdots t_n$

is a trace both in $\text{Beh}(N \oplus D{\downarrow}FS) \upharpoonright (T_d \backslash T_I)$ and in $\text{Beh}(D{\downarrow}FS) \upharpoonright (T_d \backslash T_I)$, ending in marking M and L respectively, and $M^N \ \mathcal{B} \ L^I$. Then for every transition t such that $(tr \cdot t) \in \text{Beh}(N \oplus D{\downarrow}FS) \upharpoonright (T_d \backslash T_I)$ ending in marking M', it also holds that $(tr \cdot t) \in \text{Beh}(D{\downarrow}FS) \upharpoonright (T_d \backslash T_I)$ ending in marking L' and $M'^N \ \mathcal{B} \ L'^I$. We now distinguish three scenarios for t.

1. $t \in T_d \backslash T_I$. t can also be performed by D. The possible problem is when places in ${}^{(FS)}t$ or $t^{(FS)}$ are in the interface. However, the property of $\mathcal{B}$ described by Theorem 3 guarantees that the shared markings between N and the interface of D have the same tokens for the shared places. The final marking in this case will still be in the bisimulation, i.e., the final marking L' from D will preserve the number of tokens in the M'^N and L'^I. Hence, by Equation (4) we conclude $M'^N \ \mathcal{B} \ L'^I$, and using induction hypothesis we conclude also that $(tr \cdot t) \in \text{Beh}(D{\downarrow}FS) \upharpoonright (T_D \backslash T_I)$.
2. $t \in T \cap T_I$. t is a transition from N and from D. Then the bisimulation gives that $t \ \mathcal{B} \ t$ and $L^I \xrightarrow{t,FS} L'^I$ is a firing from D, where $M^N \ \mathcal{B} \ L'^I$. Using the induction hypothesis we conclude that also $(tr \cdot t) \in \text{Beh}(D{\downarrow}FS) \upharpoonright (T_D \backslash T_I)$.
3. $t \in T \backslash T_I$. t is a transition from N but not from D. Since $\mathcal{B}$ is a weak bisimulation and $M^N \ \mathcal{B} \ L^I$, then the interface of D can perform zero or more transitions in T_I (hence not visible when restricting to $T_D \backslash T_I$) until a transition L'^I such that $M'^N \ \mathcal{B} \ L'^I$. Again, the induction hypothesis allows us to conclude that $(tr \cdot t) \in \text{Beh}(D{\downarrow}FS) \upharpoonright (T_D \backslash T_I)$.

By (1), (2), and (3), and because the empty trace is always globally safe, we conclude by induction that any trace in $\text{Beh}(N \oplus D{\downarrow}FS)$ is also in $\text{Beh}(D{\downarrow}FS)$, when restricted to $T_d \backslash T_I$. □

Example Recall our running examples. As explained in the end of Section 6.3, there is a weak bisimulation $\mathcal{B}$ between the interface P_I of the delta net for payment and the core net C for coffee. This bisimulation obeys Equation (2) because the shared transitions are related by $\mathcal{B}$, Equation (3) because the places of C and P_I are disjoint, and Equations (4) and (5) because, in this case, $\text{dom}(R_B) \cap S_I = \emptyset$, i.e., there is no place in the interface connected to a transition outside the interface. Consequently the composition $CP = C \oplus P_I$ is correct with respect to Criterion 2.

Consider now the application of the delta net for milk M to the previously obtained core CP. A possible weak bisimulation between CP and the interface of M relates equal markings of the places READY in CP and READY in M, as well as of the places WAIT and I-S$_1$. Note that, in order to use Theorem 3, we need to include markings for any number of tokens in READY, because of Equations (4) and (5). Furthermore, Equation (2) trivially holds, and our specific bisimulation relation $\mathcal{B}$ (which obeys Equations (2)–(5)) also captures Equation (3). We conclude that the composition $CP \oplus M$ is also correct with respect to Criterion 2.

6.6 Safety of the core behaviour for the combined features

Our last correctness criterion compares the core net with the combined net with respect to all features, as opposed to the first criterion that only considered the features of the core net. When including the features in the delta net, we consider it *safe* to inhibit traces that were initially possible, provided that no new traces are introduced. We formalise safety using trace inclusion.

Criterion 3 (safety/core/combined) *Let $N = (S, T, R, M_0,\ F, f)$ be a core net and $D = (S_d, T_d, R_d, M_{0d}, F_d, f_d, S_I, T_I)$ a delta net. We say that $N \oplus D$ is safe with respect to N and to the combined features iff*

$$\forall FS \subseteq F \cup F_d \ :\ \text{Beh}(N{\downarrow}FS) \supseteq_T \text{Beh}(N \oplus D{\downarrow}FS).$$

We claim that, when applying a delta net connecting only places from the interface to the rest of the delta, the delta net application is safe with respect to N and the combined features.

Theorem 4 *When applying a delta net $D = (S_d, T_d, R_d, M_{0d}, F_d, f_d, S_I, T_I)$ to a core net N, $N \oplus D$ is safe with respect to N and the combined features if:*

$$\forall s \in S_I, t \in T_d \backslash T_I \ :\ (t, s) \notin R_d \quad \wedge \quad \forall t \in T_I, s \in S_d \backslash S_I \ :\ (s, t) \notin R_d. \tag{6}$$

The theorem is easily justified by the fact that, after the application, the core net will only be connected to the delta net through arcs pointing from the core to the delta net. These arcs can only further restrict when core transitions can fire.

Example We exemplify the application of two delta nets in this paper: the *Payment* and the *Milk* nets (Fig. 8c and 8b). The first net obeys Equation (6) in Theorem 4, hence the correctness Criterion 3 holds. The second delta net has arcs connecting places from the interface to a non-interface transition, invalidating Equation (6). However, in this case the safety criterion is nevertheless preserved, because a token that exits the core when firing ADD MILK is transported back to its origin in the same step.

7 Encoding Feature Nets into Petri Nets

Petri nets are a general modelling formalism, proposed for a wide variety of applications. Feature Nets and Dynamic Feature Nets leverage the power of Petri nets for modelling static and dynamic software product lines. They combine the behaviour of a set of Petri nets in a single model, thus offering conciseness and convenience when modelling entire software families.

Theorem 1 shows that a transition-labelled Feature Net is equivalent in behaviour to a set of Petri nets, one for each product defined by the SPL. Arc-labelled Feature Nets also do not exceed the expressive power of Petri nets. This is indicated by the fact that an arc-labelled FN can be first encoded as a transition-labelled FN, whose behaviour can be described using a set of regular Petri nets. We now provide a deeper insight to this claim, by providing brief guidelines on how to encode arc-labelled, transition-labelled, and dynamic Feature Nets into Petri nets.

The first encoding from arc-labelled FN to transition-labelled FN replaces each transition attached to n arcs in R by 2^n transitions, one for each possible combination of the possible arcs. This is illustrated by an example in Fig. 12.

The second encoding, that is, from transition-labelled Feature Nets to Petri nets can be achieved by encoding the application condition of FN transition occurrences (cf. Definition 13) by considering, for each feature F, two places—F ON and F OFF—marked in mutual exclusion depending on whether the feature is selected or not. We illustrate this idea in Fig. 13. The place MILK ON is associated to the presence of the feature *Milk*. When there is a token in this place, the transitions $T_1 \ldots T_n$ are enabled.

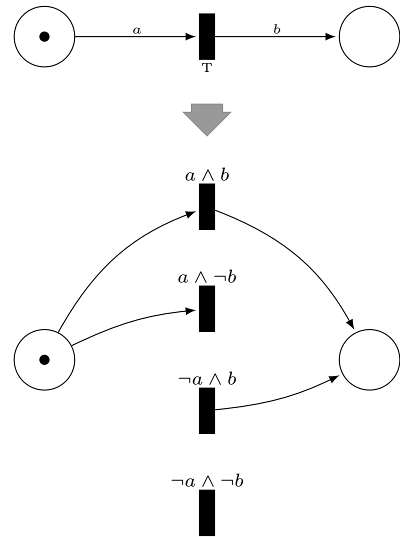


Fig. 12: Encoding an arc-labelled FN (top) into a transition-labelled FN (bottom).

These n transitions correspond to those transitions of the FN that require the feature *Milk* to be present. Hence they are allowed to occur only when there is a token in the place Milk on. The double arcs between Milk on and each T_i guarantee that this token remains in the same place. The transitions connect and disconnect capture the selection of features—other variations of encodings are also possible, e.g., where the feature selections cannot be modified during the execution of the net.

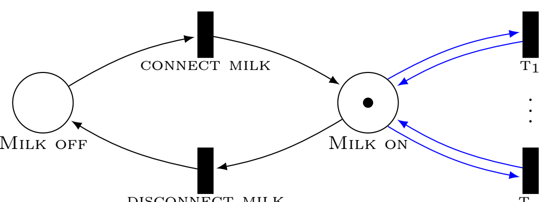


Fig. 13: Encoding a feature selection as a Petri net marking.

This approach to encode transition-labelled Feature Nets requires the introduction of only a pair of transitions and a pair of places for each feature. However, this example assumes that the original transitions are labelled with simple application conditions either with a single feature (such as *Milk*), with a negated feature, or with a conjunction of (negated) features. In the presence of more complex application conditions these have to be manipulated. More specifically, they have to be converted into a disjunctive normal form, which effectively means that all solutions to the condition have to be found. For example, a transition labelled with $a \wedge (b \vee c)$ has to be split into 2 transitions: one with $a \wedge b$ and another with $a \wedge c$; one for each solution of the condition. The former will be connected to the place denoting that a is present and to the one denoting that b

is present, and similarly for the latter with respect to a and c. Summarising, encoding transition-labelled Feature Nets into Petri nets can be a complex operation and can produce a larger number of transitions when the application conditions are complex.

A similar approach to the encoding from transition-labelled Feature Nets can be used to convert Dynamic Feature Nets into Petri nets. Additionally, the encoding of Dynamic Feature Nets also needs to include the update instructions for features triggered by transitions. This idea is illustrated by exemplifying the encoding of a transition that is active when *Coffee* is selected and that turns the *Milk* feature on (cf. Fig. 14). The encoding assumes the initial feature selection to be {*Coffee*}. As before, features are explicitly represented by pairs of places, denoting their presence or absence. Furthermore, the application conditions are also represented by double arcs (cf. blue arcs connected to COFFEE ON), connecting the transitions to the places denoting the availability of features. However, the original transition is split into two mutual exclusive transitions: one assumes that *Milk* is selected (T_M), and the other that *Milk* is not selected ($\mathrm{T}_{\neg M}$). This assumption is explicit by their pre-sets, which include red arcs from MILK OFF or MILK ON. Whenever either T_M or $\mathrm{T}_{\neg M}$ is fired, a token is placed in MILK ON, and no token is left in MILK OFF, hence modelling the update expression *Milk on*.

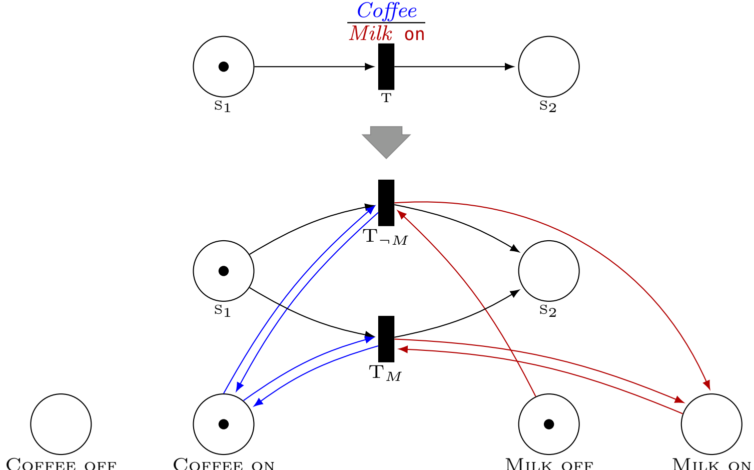


Fig. 14: Encoding a Dynamic Feature Net (top) into a Petri net (bottom).

These encodings show that, even though Feature Nets offer a more concise way to describe the systems in an SPL than Petri nets, their expressive power is still equivalent. This also means that analysis techniques for Petri nets still apply to Feature Nets. However, due to the complexity of the encoding, we have discussed alternative approaches to model check Feature Nets that do not rely on this encoding (Section 4.3 and 5.4).

DFN provides support for dynamic SPLs, by allowing explicit modelling of feature reconfiguration as part of the behavioural model. By adding update expressions to Feature Nets, Dynamic Feature 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 present Feature 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 practical applicability of our modular modelling workflow could arise. These are subject to future research. In addition, many analysis techniques that exist to determine the behavioural correctness of a Petri net design [30] could be adopted for Feature Nets.

8 Model Checking Feature Nets

This section provides a concrete example of how to model check properties of (Dynamic) Feature Nets. Our approach uses the mCRL2 toolset [12] to model check properties of (1) Petri nets, (2) transition-labelled Feature Nets, and (3) Dynamic Feature Nets. One can formally specify and analyse system behaviour in mCRL2 using its own specification language based on process algebra. A Petri net can be encoded as a mCRL2 process that describes its reachability graph (Definition 9); we exemplify this in Section 8.1. We incrementally extend the encoding of Petri nets, first by adding application conditions (Section 8.2), then by adding feature updates (Section 8.3). We show how to prove properties of these mCRL2 models in Section 8.4.

8.1 Modelling Petri Nets with mCRL2

The reachability graph of the Petri net from Figure 1 is modelled as a mCRL2 process `St(...)` defined in Listing 1. The first line declares the actions (transition names), and the **`proc`** clause defines the process `St`, representing a state of the graph parametrised on a tuple defining the current marking. These parameters are underlined to highlight their role as the state of the process (and of the reachability graph produced later). The definition of the process follows the pattern `t1+t2+t3`, denoting a choice of 3 possible transitions. In turn, each transition is defined as `(cond)->(action.newState)`: upon execution, if the condition `cond` is met, `action` is performed, causing the process to evolve into `newState`. The new state is built using the auxiliary functions `pre` and `suc`, which calculate the predecessor and successor of a number, respectively. The last line defines the initial state, where a single token is placed in the places WAIT (`wt`), and two tokens are in COFFEE FULL (`cf`).

```
act serve,brew,refill;

proc St(wt:Nat,rd:Nat,cf:Nat,ce:Nat) =
  ( wt>0 && cf>0 ) -> ( brew   . St(pre(wt),suc(rd),pre(cf),suc(ce)))  +
  ( rd>0 )         -> ( serve  . St(suc(wt),pre(rd),cf,ce))            +
  ( ce>0 )         -> ( refill . St(wt,rd,suc(cf),pre(ce)))            ;

init St(1,0,2,0) ;
```

Listing 1: Process in mCRL2 that models the reachability graph of Fig. 1.

From this process algebra description, the mCRL2 tools can directly produce a graph that describes its behaviour: each node represents a reachable instance of the

process St, and labels are the actions of the process algebra **serve**, **brew**, or **refill**. This graph corresponds exactly to the reachability graph of the Petri net in Figure 1 depicted at the end of Section 3, with 6 states and 9 edges. By substituting the initial state with (1,0,50,0) (i.e., $n = 50$) the reachability graph grows to 102 states and 201 edges.

The general encoding from a Petri net into an mCRL process algebra can be done fully automatic, but we omit here its formalisation because we only aim at exemplifying how one could verify Feature Nets and Dynamic Feature Nets. Intuitively, a Petri net can be encoded into a process (of a process algebra) by defining a single process St parametrised by several arguments, one per each place in the Petri net. This process is then defined by a sum of choices, one for each transition. Each of these choices is an implication that checks if the condition is enabled (as in Definition 6), and if so it performs an action that identifies the transition and behaves as a process St after updating the parameters based on the transition's destination arcs.

8.2 Modelling Transition-Labelled Feature Nets with mCRL2

The transition-labelled Feature Net in Figure 2 models a coffee machine with an optional *Milk* feature. Its reachability graph can be modelled in mCRL2 by extending Listing 1 with the additional places and transitions, as well as application conditions, as shown in Listing 2. In this example we introduce a boolean variable milk, set to **true** in the second line. For brevity the *Coffee* feature is not explicitly included in the specification. The conditions guarding the *Milk*-related transitions are then extended with the application condition, written with a highlighted background as in milk. The expressivity of these conditions is that of mCRL2's language, including the traditional boolean operators.

```
map milk:Bool;
eqn milk=true;

act serve,brew,refill,add_milk,serve_w_milk,refill_milk;

proc St(wt:Nat,rd:Nat,cf:Nat,ce:Nat,mr:Nat,mf:Nat,me:Nat) =
  ( wt>0 && cf>0 ) -> ( brew   . St(pre(wt),suc(rd),pre(cf),suc(ce),mr,mf,me))   +
  ( rd>0 )         -> ( serve  . St(suc(wt),pre(rd),cf,ce,mr,mf,me))             +
  ( ce>0 )         -> ( refill . St(wt,rd,suc(cf),pre(ce),mr,mf,me))             +
  ( milk && rd>0 && mf>0 )
                   -> ( add_milk . St(wt,pre(rd),cf,ce,suc(mr),pre(mf),suc(me))) +
  ( milk && mr>0 ) -> ( serve_w_milk . St(suc(wt),rd,cf,ce,pre(mr),mf,me))       +
  ( milk && me>0 ) -> ( refill_milk  . St(wt,rd,cf,ce,mr,suc(mf),pre(me)))       ;

init St(1,0,2,0,0,2,0) ;
```

Listing 2: Process in mCRL2 that models the reachability graph of Fig. 2.

A general encoding from a transition-labelled Feature Net can be defined by adapting the encoding from Petri nets intuitively described in the previous section. More concretely, a Feature Net is encoded as a single process St with a choice for each transition, as before. However, each of these choices now includes also the application condition together with the enabling conditions. These application conditions are specified as logical formulas, using global variables to specify the features values.

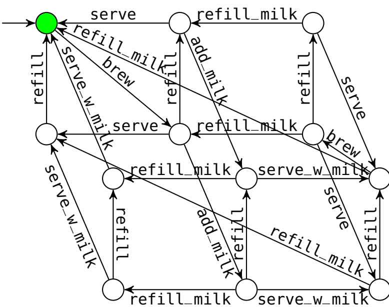


Fig. 15: Reachability graph of the Feature Net from Figure 2 ($n = m = 1$).

The transition system built from the specification in Listing 2 using the mCRL2 tool has 27 states and 66 edges. The same process with an initial state where $n = m = 1$ (i.e., only one refill of coffee and one of milk), produces a smaller transition system with 12 states and 24 edges, depicted in Figure 15. When increasing this number to $n = m = 50$, for example, the number of states and edges go up to 7803 and 25602, respectively.

This approach for modelling Feature Nets allows the easy configuration of desired features, and the verification of individual features and feature combinations. This constitutes a *product-based* analysis approach. However, it does not provide the means to automatically verify all valid combinations of features, as an *SPL-based* analysis would demand. Recent research towards efficient SPL-based analysis of behavioural properties modelled with mCRL2 [5] appears promising and we will follow this objective in future work.

An alternative approach to model checking FNs would be to use the dedicated tools for featured transition systems [10] instead of mCRL2. This could be achieved by building the variable reachability graph (as described in Section 4.3), which already represents an FTS. Note that this is not a trivial construction. A possible way to generate this variable reachability graph is to use the mCRL2 specification to produce the transition system by dropping all application conditions, and later to add these application conditions to the resulting transition system. Hence in our example the size of the featured transition system is the same as the size of reachability graph when the milk feature is selected.

8.3 Modelling Dynamic Feature Nets with mCRL2

This subsection further extends the previous example by showing how to model the Dynamic Feature Net in Figure 6 with mCRL2. The main differences to the previous example are the possibility to turn the *Milk* feature on and off during the execution of the DFN, and the inclusion of new places and transitions that trigger these updates. Hence the *Milk* feature is no longer modelled as a static boolean variable, but included in the state. The process is shown in Listing 3, highlighting both the application conditions and the updates with a darker background.

The resulting reachability graph has 45 states and 111 edges. Figure 16 shows the smaller reachability graph when $n = m = 1$ (only one refill of coffee and milk), which

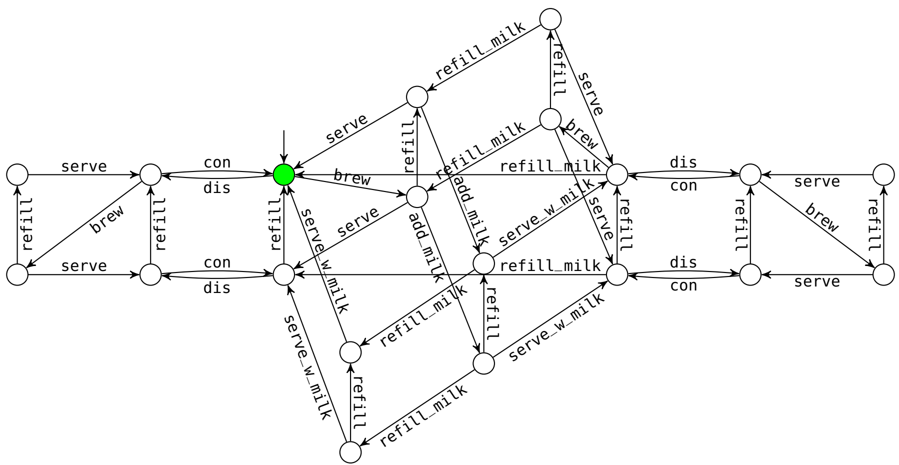


Fig. 16: Reachability graph of the Dynamic Feature Net from Fig. 6 ($n = m = 1$).

has 20 states and 42 edges. When increasing this number to 50, for example, the number of states and edges go up to 13005 and 41055, respectively.

```
act serve,brew,refill,serve_w_milk,add_milk,refill_milk,con,dis;

proc St(wt:Nat,rd:Nat,cf:Nat,ce:Nat,mr:Nat,mf:Nat,me:Nat,on:Nat,off:Nat,milk:Bool) =
  ( wt>0 && cf > 0 ) ->
    ( brew   . St(pre(wt),suc(rd),pre(cf),suc(ce),mr,mf,me,on,off,milk))       +
  ( rd>0 ) ->
    ( serve  . St(suc(wt),pre(rd),cf,ce,mr,mf,me,on,off,milk))                 +
  ( ce>0 ) ->
    ( refill . St(wt,rd,suc(cf),pre(ce),mr,mf,me,on,off,milk))                 +
  ( milk && rd>0 && mf>0 ) ->
    ( add_milk     . St(wt,pre(rd),cf,ce,suc(mr),pre(mf),suc(me),on,off,milk)) +
  ( milk && mr>0 ) ->
    ( serve_w_milk . St(suc(wt),rd,cf,ce,pre(mr),mf,me,on,off,milk))           +
  ( milk && me>0 ) ->
    ( refill_milk  . St(wt,rd,cf,ce,mr,suc(mf),pre(me),on,off,milk))           +
  ( wt>0 && off>0 ) ->
    ( con . St(wt,rd,cf,ce,mr,mf,me,suc(on),pre(off),true))                    +
  ( wt>0 && on>0 ) ->
    ( dis . St(wt,rd,cf,ce,mr,mf,me,pre(on),suc(off),false))                   ;

init St(1,0,2,0,0,2,0,1,0,true) ;
```

Listing 3: Process in mCRL2 that models the dynamic reachability graph of Fig. 6.

8.4 Verification of Feature Nets

It is possible to analyse the behaviour of any of the processes described above using mCRL2's modal logic [12], a first-order modal μ-calculus. Details of the mCRL2 lan-

guage and tools can be found online.[2] The following properties refer to the Dynamic Feature Net of Listing 3.

[true*]<true> true (deadlock freedom)

[true*.brew]<(!brew)*.(serve + serve_w_milk)> true (brew must be served)

[true*.dis.(!con)*.add_milk] false (no *Milk*, no add_milk)

[true*.brew.(!brew && !serve)*]<(!brew)*.serve> true (serve before 2 brews)

All formulas but the last hold. The first formula stipulates that after any sequence of actions another action can execute. The second formula says that all paths that reach **brew** must eventually perform **serve** or **serve_w_milk** without performing another **brew** action. The third formula says that there can be no path that reaches a **dis** and performs a **add_milk** before **con** is performed. Finally, the last formula says that a **serve** must be performed before executing 2 **brew**s – this is only true when there are no milk capabilities, otherwise **serve_w_milk** can be executed instead.

Size of the reachability graph. In general, the reachability graph can be arbitrarily large. For instance, a Petri net (or Feature Net) with a transition that has a single arc pointing to a place will have an infinitely large reachability graph: this transition has an empty pre-set and is able to send an infinite amount of tokens to its connected place. Furthermore, the reachability problem of Petri nets was shown to be EXPSPACE-hard [28]. However, if one can either show that the number of tokens in each place is bounded by a maximum value k, or if one disallows places with more than k tokens (such nets are known as k-*safe*), then the maximum number of markings is bounded to $(k+1)^{|P|}$, where $|P|$ is the number of places. This maximum number of markings is also the maximum number of states in the reachability graph. In our running example we have 7 places, and it is possible to show that the number of tokens per place is bounded by k when the initial process is St(1,0,k,0,0,k,0). Hence, when $k \in \{1, 2, 50\}$ the maximum number of markings is 128, 2187, or around 900 billions. The actual number of states is much smaller than this upper bound because only a small number of combinations of markings is actually possible: the places WAIT, READY, and MILK READY always have exactly 1 token combined; the places COFFEE FULL and COFFEE REFILLABLE can have exactly k tokens combined; and the same holds for the remaining 2 places. Hence the number of states is $3 * (k+1) * (k+1)$ (which gives the number of token combinations in each place). Therefore, when $k \in \{1, 2, 50\}$, the number of markings is 12, 27, and 7803. Observe that the verification of logical properties in mCRL2 does not require creating the full reachability graph—the processes and the formulas are analysed until the property is either refuted or proven, which may not require building the full state space.

9 Related Work

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

[2] http://www.mcrl2.org

9.1 Petri Net Extensions

Petri net composition and decomposition strategies that preserve some properties of the initial net(s) have been studied thoroughly [6, 37, 36, 20].

In *Open Petri Nets* [4], places designated as open represent an interface towards the environment. Open nets are composed by fusing common open places, and the composition operation is shown to preserve behaviour with respect to an inverse decomposition operation. Our Petri net model uses a similar notion of interface, which includes an abstraction of the net that will be matched during application. We use an incremental approach using application of deltas instead of a symmetric composition operation, guided by the intuition that larger systems are built by extending more fundamental systems. The main focus of open Petri nets is the study of properties in a category of nets, while we have a more practical focus on the incremental development of nets.

Inhibitor arc Petri nets [1] 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 [13] 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 FN 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* [38], 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 [19] are similar to self-modifying Petri nets, 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 DFN.

Using *net rewriting systems* [29], dynamic changes in the configuration of a Petri net are described using a rewriting rule that 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 DFN 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)FN 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.

9.2 Behavioural SPL Models

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. A survey of formal methods for software product lines has recently been published [7].

UML activity diagrams have been used to model the behaviour of SPL by superimposing several such diagrams in a single model [14]. Attached to the activity diagram's elements are “presence expressions,” which are similar to application conditions. 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. [17]).

Gruler et al. extended Milner's CCS with a product line variant operator that allows an alternative choice between two processes [22]. The *PL-CCS* calculus includes information about variability: by defining dependencies between features, one can control the set of valid configurations [21].

Variability is often modelled using transition systems enhanced with product-related information. *Modal transition systems* (MTS) [27] allow optional transitions, lending themselves as a tool for modelling a set of behaviours at once [18]. Generalised extended MTS [16] introduce cardinality-based variability operators and propose to use temporal logic formulas to associate related variation points. Asirelli et al. reason about MTS using propositional deontic logic, which is able to express constraints on variable behaviour [2,3].

Modal I/O automata [26] are a behavioural formalism for describing the variability of components based on MTS and I/O automata. Mechanisms for component composition are provided to support a product line theory. These approaches do not relate, or, in the case of generalized extended MTS, only partially relate behaviour to elements of a structural variability model.

Featured transition systems (FTS) [10] are an extension of labeled transition systems. Similar to Feature Nets, transitions are explicitly labeled with boolean expressions that refer to a feature model, and a feature selection determines the subset of active transitions. FTS are a purely annotative [25] approach, which requires a pre-existing understanding of what the system looks like. While FNs also employ annotations to assign features to behaviour, arc-labelled FNs support a compositional approach, where features are specified in separate modules (delta nets), which are later combined to model the complete system. FNs can be seen as a higher level modelling language, whose semantics can be expressed using FTS, as explained in Section 4.3 and reinforced in Section 8.2. To verify properties of FTS, Classen et al. developed the variability-aware model checker SNIP and have recently proposed symbolic model checking algorithms [9] to counter the state space explosion caused by the large number of different behaviours of an SPL. Our model checking approach relies on mCRL2, and although still in an early development stage, we can take advantage of the recent progress towards modular, family-level verification of SPLs with mCRL2 [5].

Delta-oriented programming (DOP) [35] is an approach for implementing software product lines that organises the SPL code base into a set of delta modules that comprise modifications of object-oriented programs. Delta feature nets are conceptually similar to delta modules to the extent that both encapsulate certain behavioural modifications that can be plugged into a larger system. However, our notion of delta application enables the modular specification of an SPL, whereas in DOP it is a mechanism of deriving software products.

With regard to *dynamic SPLs*, dynamic Feature Nets as introduced in [31] is, to the best of our knowledge, the first formal specification and analysis framework for dynamic SPL behaviour. The DFN formalism's strength is simplicity, making it easy to understand and use. Cordy et al.'s work [11] goes in a similar direction by extending FTS with labels that update the feature selection. To reduce the space of runtime configurations, adaptive FTS distinguish between fixed features (defined at compile time), adaptable features (which the program can chose to modify at runtime), and environment features (which reflect variability of the environment).

10 Conclusion

This paper proposes a formal framework for modelling systems with a high degree of variability, addressing an important challenge in software product line engineering. The modelling formalism used is Feature Nets, a lightweight Petri net extension, of which we present two variants. In transition-labelled FNs, the firing of transitions is conditional on the presence of certain features through *application conditions*. Arc-labelled FNs place application conditions on the arcs, effectively determining their presence or absence. For arc-labelled FNs we present an approach to composing behavioural models from separately engineered models of individual features. Three correctness criteria for such compositions are also presented.

The Dynamic FN model extends transition-labelled FNs with the ability to express dynamic variability. *Update expressions* associated with DFN 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.

Future work will further explore the possibilities of analysis and verification, and investigate the practical applicability of the proposed modular techniques. The usage of the mCRL2 toolset to analyse and verify Feature Nets (Section 8) provide a simple and elegant starting point. The verification of more complex behavioural properties of Petri-nets [30] can also be investigated, after adjusting its adequacy for the domain of products and product lines. The practical applicability of the modular modelling FN techniques that we propose also needs closer examination, especially with respect to scalability. This could be accomplished by a case study involving more substantial SPL models.

References

1. Agerwala, T., Flynn, M.: Comments on capabilities, limitations and "correctness" of Petri nets. In: 1st annual symposium on Computer architecture Proceedings, ISCA '73, pp. 81–86. ACM Press (1973)
2. Asirelli, P., Beek, M., Fantechi, A., Gnesi, S.: A logical framework to deal with variability. In: Integrated Formal Methods, *LNCS*, vol. 6396, pp. 43–58. Springer (2010)
3. Asirelli, P., Beek, M., Fantechi, A., Gnesi, S.: A compositional framework to derive product line behavioural descriptions. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, *LNCS*, vol. 7609, pp. 146–161. Springer (2012)
4. Baldan, P., Corradini, A., Ehrig, H., Heckel, R.: Compositional semantics for open Petri nets based on deterministic processes. Mathematical Structures in Computer Science **15**(01), 1–35 (2005)

5. ter Beek, M.H., de Vink, E.P.: Towards modular verification of software product lines with mCRL2. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, *LNCS*, vol. 8802, pp. 368–385. Springer (2014)
6. Berthelot, G.: Transformations and decompositions of nets. Petri Nets: Central Models and Their Properties pp. 359–376 (1987)
7. Clarke, D.: Quality Assurance for Diverse Systems, chap. 5, pp. 27–37 (2011). Deliverable 1.2 of the EternalS Coordination Action (FP7-247758), supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme
8. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Model checking software product lines with SNIP. Journal on Software Tools for Technology Transfer **14**(5), 589–612 (2012)
9. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Formal semantics, modular specification, and symbolic verification of product-line behaviour. Science of Computer Programming **80**, 416–439 (2014)
10. Classen, A., Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., Raskin, J.F.: Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. IEEE Transactions on Software Engineering **39**(8), 1069–1089 (2013)
11. Cordy, M., Classen, A., Heymans, P., Legay, A., Schobbens, P.Y.: Model checking adaptive software with featured transition systems. In: Assurances for Self-Adaptive Systems, *LNCS*, vol. 7740, pp. 1–29. Springer (2013)
12. Cranen, S., Groote, J.F., Keiren, J.J.A., Stappers, F.P.M., de Vink, E.P., Wesselink, W., Willemse, T.A.C.: An overview of the mCRL2 toolset and its recent advances. In: N. Piterman, S.A. Smolka (eds.) TACAS, *Lecture Notes in Computer Science*, vol. 7795, pp. 199–213. Springer (2013)
13. Ţiplea, F.L.: On conditional grammars and conditional Petri nets, pp. 431–455. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1994)
14. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: Generative Programming and Component Engineering, *LNCS*, vol. 3676, pp. 422–437. Springer (2005)
15. Desel, J., Esparza, J.: Free choice Petri nets. Cambridge University Press, New York, NY, USA (1995)
16. Fantechi, A., Gnesi, S.: Formal modeling for product families engineering. In: International Software Product Line Conference, SPLC '08, pp. 193–202. IEEE Press (2008)
17. Farooq, U., Lam, C.P., Li, H.: Transformation methodology for UML 2.0 activity diagram into colored Petri nets. In: Advances in Computer Science and Technology, pp. 128–133. ACTA Press (2007)
18. Fischbein, D., Uchitel, S., Braberman, V.: A foundation for behavioural conformance in software product line architectures. In: International Workshop on the Role of Software Architecture in Analysis and Testing, pp. 39–48. ACM Press (2006)
19. Ghabri, M.K., Ladet, P.: Dynamic Petri nets and their applications. In: International Conference on Computer Integrated Manufacturing and Automation Technology, pp. 93–98 (1994)
20. Girault, C., Valk, R.: Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications. Springer, Secaucus, NJ, USA (2001)
21. Gruler, A., Leucker, M., Scheidemann, K.: Calculating and modeling common parts of software product lines. In: International Software Product Line Conference, SPLC '08, pp. 203–212. IEEE Press (2008)
22. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and model checking software product lines. In: International Conference on Formal Methods for Open Object-based Distributed Systems, *LNCS*, vol. 5051, pp. 113–131. Springer (2008)
23. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. IEEE Computer **41**(4), 93–95 (2008)
24. Holzmann, G.J.: The SPIN Model Checker - primer and reference manual. Addison-Wesley (2004)
25. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: ICSE '08: Proceedings of the 30th international conference on Software engineering, pp. 311–320. ACM Press (2008)
26. Larsen, K., Nyman, U., Wąsowski, A.: Modal I/O automata for interface and product line theories. In: Programming Languages and Systems, *LNCS*, vol. 4421, pp. 64–79. Springer (2007)

27. Larsen, K., Thomsen, B.: A modal process logic. In: Third Annual Symposium on Logic in Computer Science, pp. 203–210. IEEE Press (1988)
28. Lipton, R.: The reachability problem requires exponential space. Tech. Rep. 62, Yale University (1976)
29. Llorens, M., Oliver, J.: Structural and dynamic changes in concurrent systems: reconfigurable Petri nets. IEEE Transactions on Computers **53**(9), 1147–1158 (2004)
30. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
31. Muschevici, R., Clarke, D., Proença, J.: Feature Petri Nets. In: Workshop on Formal Methods and Analysis in Software Product Line Engineering, *SPLC '10*, vol. 2, pp. 99–106. Lancaster University (2010)
32. Muschevici, R., Proença, J., Clarke, D.: Modular modelling of software product lines with Feature Nets. In: Software Engineering and Formal Methods, *LNCS*, vol. 7041, pp. 318–333. Springer (2011)
33. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Springer (2005)
34. Schaefer, I.: Variability modelling for model-driven development of software product lines. In: D. Benavides, D.S. Batory, P. Grünbacher (eds.) International Workshop on Variability Modelling of Software-Intensive Systems, vol. 37, pp. 85–92. Universität Duisburg-Essen, Linz, Austria (2010)
35. Schaefer, I., Bettini, L., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: International Software Product Line Conference, SPLC '10, pp. 77–91. Springer (2010)
36. Schnoebelen, P., Sidorova, N.: Bisimulation and the reduction of Petri nets. In: Application and Theory of Petri Nets, *LNCS*, vol. 1825, pp. 409–423. Springer (2000)
37. Souissi, Y., Memmi, G.: Composition of nets via a communication medium. In: Advances in Petri Nets, *LNCS*, vol. 483, pp. 457–470. Springer (1991)
38. Valk, R.: Self-modifying nets, a natural extension of Petri nets. Automata, Languages and Programming pp. 464–476 (1978)