Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Master's Degree en Formal Methods in Computer Science

Trabajo Fin de Máster

Analysis of Concurrent Product Lines using Temporal Logic

Autor: Dejian Kuang
Tutor(a): María Elena Gómez Martínez, José Ignacio Requeno Jarabo

September 2025

Este Trabajo Fin de Máster se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Master's Degree
Master's Degree en Formal Methods in Computer Science

Título: Analysis of Concurrent Product Lines using Temporal Logic

September 2025

Autor: Dejian Kuang
Tutores: María Elena Gómez Martínez, José Ignacio Requeno Jarabo
Facultad de Informática
Universidad Complutense de Madrid

Resumen

Las líneas de productos de redes de Petri (PNPL) proporcionan un formalismo potente y compacto para modelar y analizar la variabilidad del comportamiento en sistemas concurrentes. Al permitir anotaciones basadas en características sobre transiciones, lugares y arcos, PNPL permite la especificación unificada de múltiples variantes de productos en un único modelo del 150 % de redes de Petri. Sin embargo, la comprobación de propiedades dinámicas como la alcanzabilidad, la ausencia de bloqueo y la vivacidad en todas las configuraciones sigue siendo un reto importante. Las técnicas tradicionales de comprobación de modelos no suelen ser escalables o carecen de soporte para una semántica que tenga en cuenta la variabilidad. La lógica CTL (del inglés, Computational Tree Logic) con características (fCTL) se ha introducido para razonar formalmente sobre propiedades de comportamiento en Líneas de Producto de Software (SPL) modeladas utilizando Sistemas de Transición con características (FTS^+). Mientras que fCTL es muy adecuado para la comprobación de modelos basados en características, su aplicación a modelos PNPL no se ha explorado sistemáticamente.

En este trabajo investigamos la aplicabilidad e idoneidad de fCTL para PNPL proponiendo un proceso de transformación estructurado de PNPL a FTS^+ a través de un gráfico de alcanzabilidad anotado por características. Nuestro enfoque consta de tres pasos principales:

(i) Construir un RG anotado por características a partir del modelo PNPL, preservando explícitamente la información de variabilidad de la red original;

(ii) Convertir este RG en una representación FTS^+ correspondiente mediante la definición de mapeos formales para estados, transiciones, guardias de características y proposiciones atómicas;

(iii) Aplicar especificaciones fCTL al FTS^+ resultante para verificar propiedades como la alcanzabilidad, la libertad de bloqueo y la vivacidad a través de configuraciones de características válidas.

Con tal fin, evaluamos nuestra metodología a través de casos de estudio ilustrativos, incluyendo modelos PNPL representativos con comportamiento multi-token y restricciones de características complejas. Nuestros resultados experimentales muestran que el enfoque propuesto captura eficientemente la variabilidad tanto estructural como de comportamiento de PNPL a la vez que permite una verificación escalable basada en fCTL. La transformación también admite estrategias de poda de características para eliminar configuraciones no válidas causadas por

la convergencia de múltiples características.

Este trabajo demuestra la viabilidad de unir PNPL y fCTL a través de un modelo de comportamiento intermedio que preserva las características. Aporta una vía formal y práctica para aplicar la verificación lógica temporal en sistemas concurrentes configurables, ofreciendo una perspectiva novedosa para el análisis formal de líneas de productos de sistemas.

Palabras clave: Líneas de productos de redes de Petri, lógica de árbol computacional destacada, sistemas de transición de características, gráfico de alcanzabilidad.

Abstract

Petri Net Product Lines (PNPL) provide a powerful and compact formalism for modelling and analysing behavioural variability in concurrent systems. By allowing feature-based annotations on transitions, places, and arcs, PNPL enables the unified specification of multiple product variants in a single 150% Petri net model. However, verifying dynamic properties such as reachability, deadlock-freedom, and liveness across all configurations remains a significant challenge. Featured Computation Tree Logic (fCTL), an extension of Computation Tree Logic (CTL), has been introduced to formally reason about behavioural properties in Software Product Lines (SPLs) modelled using Featured Transition Systems (FTS^+). While fCTL is well-suited for feature-aware model checking, its application to PNPL models has not been systematically explored.

In this work, we investigate the applicability and suitability of fCTL to PNPL by proposing a structured transformation pipeline from PNPL to FTS^+ through an feature-annotated Reachability Graph. Our approach involves three main steps:

(i) Constructing a feature-annotated RG from the PNPL model, explicitly preserving variability information from the original net;

(ii) Converting this RG into a corresponding FTS^+ representation by defining formal mappings for states, transitions, feature guards, and atomic propositions;

(iii) Applying fCTL specifications to the resulting FTS^+ for verifying properties such as reachability, deadlock-freedom, and liveness across valid feature configurations.

We evaluate our methodology through illustrative case studies, including representative PNPL models with multi-token behaviour and complex feature constraints. Our experimental results show that the proposed approach efficiently captures both the behavioural and structural variability of PNPL while enabling scalable fCTL-based verification. The transformation also supports feature pruning strategies to eliminate invalid configurations caused by multiple-feature convergence.

This work demonstrates the feasibility of bridging PNPL and fCTL through an intermediate feature-preserving behavioural model. It contributes a formal and practical pathway for applying temporal logic verification in configurable concurrent systems, offering a novel perspective for the formal analysis of system

product lines.

Keywords: Petri Net Product Lines, Featured Computation Tree Logic, feature transition systems, Reachability Graph

Contents

Índice de figuras

Capítulo 1

Introduction

Petri nets are a popular formalism to describe and analyse concurrent systems [1], offering a robust mathematical and graphical framework to represent states, transitions, and concurrency. Their ability to capture complex system dynamics makes them a cornerstone in domains such as distributed systems, manufacturing, and workflow management. Petri Net Product Lines (PNPLs) [2] extend classical Petri nets with feature-based variability, enabling the modelling of concurrent systems with configurable behaviour. In other words, Petri Net Product Lines (PNPLs) permit the compact definition of a system family by means of a Petri net where all variants are superimposed, and from which specific variants can be obtained by slicing. This simplifies the management of the system family because just one artefact (the PNPL) suffices to define all variants of the family in a unified way [3].

Despite the advantages of PNPLs, analysing their dynamic properties, particularly reachability, remains a significant challenge. Traditional reachability analysis techniques, designed for classical Petri nets, are not directly applicable to PNPLs because they do not account for feature-based variability, which introduces additional complexity in state space exploration. The state space of a PNPL is inherently larger and more intricate due to the combinatorial nature of feature configurations, necessitating specialized methods to handle variability while preserving the correctness of the analysis. Furthermore, verifying dynamic properties of PNPLs, such as safety, liveness, or temporal constraints, requires a formalism capable of reasoning about feature variability in a temporal context.

1.1. Motivation

With the growing complexity of modern concurrent systems, which increasingly incorporate variability to meet diverse functional, performance, and market requirements. In domains such as automotive systems, telecommunications, and configurable manufacturing, systems must support multiple configurations while adhering to stringent safety, reliability, and performance constraints. For example, in an automotive control system, features like adaptive cruise control or lane-keeping assistance may be optional, yet their interactions with the

core system must be verified to ensure safety across all possible configurations. PNPLs provide a powerful framework for modelling such systems, but the lack of effective verification methods limits their practical adoption. Verifying each system variant individually is computationally infeasible due to the exponential number of configurations, and traditional Petri net analysis tools do not account for feature variability, leading to incomplete or inefficient verification processes.

Featured Computation Tree Logic (fCTL) [4] is a dialect of Computation Tree Logic (CTL) [5] adapted for specifying and verifying properties in software product lines. fCTL has been successfully applied to Featured Transition Systems (FTS^+), which are a generalization of Labelled Transition Systems (LTS) with feature annotations. Although fCTL is well-suited for FTS^+, its applicability to PNPL has not been thoroughly explored.

The application of fCTL to PNPLs offers a promising solution to these challenges. By leveraging fCTL's ability to reason about feature-dependent temporal properties, we can verify properties across all variants of a system family in a single analysis, significantly reducing verification effort. For instance, fCTL can express properties such as "in all configurations with feature X enabled, the system never reaches a deadlock state," enabling comprehensive verification without enumerating all variants. However, the structural mismatch between PNPLs and FTS, the native model for fCTL, necessitates a robust transformation methodology. This work is motivated by the need to bridge this gap, enabling the seamless integration of PNPL modelling with fCTL based verification. Such integration not only enhances the practical utility of PNPLs but also aligns with the broader goals of software product line engineering, where efficient verification of configurable systems is critical to reducing development costs and time-to-market.

Furthermore, the proposed research has significant implications for both theoretical and practical advancements. Theoretically, it establishes a formal connection between Petri net-based modelling and transition system-based verification, contributing to the unification of modelling and analysis paradigms in software engineering. Practically, it enables the use of existing fCTL-based model-checking tools, such as those developed for FTS, to verify PNPLs, eliminating the need for specialized PNPL verification frameworks. This is particularly valuable in safety critical domains, where formal verification is essential to ensure compliance with regulatory standards. Additionally, the transformation pathway proposed in this work addresses the scalability challenges of state space explosion by leveraging the compact representation of PNPLs, making it feasible to analyse large-scale systems with extensive variability. By addressing these challenges, this research paves the way for the broader adoption of PNPLs in real-world applications, from embedded systems to software-defined networks, where concurrency and configurability are paramount.

The objective of this paper is to evaluate the suitability of fCTL for verifying PNPL properties and to propose a formal transformation pathway from PNPLs to FTS^+. This pathway involves two critical steps: first, transforming a PNPL into a feature-enriched Reachability Graph (fRG) that captures both the dynamic behaviour and the variability of the system; second, converting the feature-enriched

RG into an FTS^+ compatible with fCTL. By enabling fCTL-based verification, this approach facilitates the analysis of temporal properties across all variants of a system family in a unified manner, ensuring that properties such as reachability, deadlock-freedom, and feature-dependent behaviours can be systematically verified.

The significance of this research lies in its potential to unify the modeling and verification of concurrent systems with variability. By leveraging the expressive power of fCTL, our approach enables the verification of complex properties that vary across system configurations, reducing the verification effort compared to analyzing each variant independently. This is particularly valuable in industries where system families must support multiple configurations while adhering to strict safety and performance requirements. Additionally, the transformation from PNPLs to FTS^+ allows the reuse of existing fCTL-based model-checking tools, avoiding the need for specialized PNPL verification frameworks. This work also contributes to the theoretical foundations of software product line engineering by establishing a formal connection between Petri net-based modeling and transition system-based verification, paving the way for further advancements in the analysis of configurable concurrent systems.

1.2. Objectives

The goal of this work is to bridge this gap by assessing the suitability of fCTL for PNPL and proposing a formal transformation from PNPL to FTS^+ for property verification. To achieve this, we propose a transformation pathway: first, convert PNPL into feature-enriched RG, which captures the dynamic behaviour and variability of PNPL; then, transform the RG into FTS^+, aligning with the structural requirements of fCTL. Finally, apply fCTL on the FTS^+ to verify properties. The main contributions of this paper are as follows:

- A formalized transformation process for converting PNPL models into feature-enriched Reachability Graphs (fRG) while preserving feature annotations and ensuring that the dynamic behaviour of all variants is accurately captured.
- A systematic method for constructing Featured Transition Systems (FTS^+) from feature-enriched RGs, aligning the structural and semantic requirements of fCTL for property verification.
- The application of FCTL on FTS^+ to verify critical system properties.

1.3. Document structure

The paper is structured as follows. We start with foundational knowledge in Chapter 2, covering Petri nets, PNPLs, reachability analysis, and fCTL, with references for further reading. Chapter 3 presents the transformation framework: converting PNPLs to feature-enriched RGs, illustrated with three case study and pruning techniques. Chapter 4 introduces transformation from RG with features

to FTS^+ for advanced fCTL properties. Chapter 5 applies fCTL to FTS^+, detailing verification strategies with examples. Chapter 6 presents some related works. Chapter 7 concludes with key findings and future directions.

Capítulo 2

Background

This chapter presents the theoretical background and existing research relevant to this work, focusing on Petri Nets, Product Line Extensions, Reachability Analysis, Featured Transition Systems (FTS^+), and Featured Computation Tree Logic (fCTL). These foundations are crucial for understanding the transformation process from PNPL to FTS^+ and the application of fCTL.

2.1. Petri Net

Petri nets (PN) are a formal modelling language widely used for describing and analysing concurrent systems [6]. Their ability to capture concurrency and synchronization has established them as a foundational tool in fields such as distributed computing, workflow management, and real-time systems. As a graphical tool, Petri nets can be used as a visual communication aid similar to flow charts, block diagrams, and networks. In addition, tokens are used in these nets to simulate the dynamic and concurrent activities of systems [7].

2.1.1. Classification of PN

Petri nets can be categorized into the following main types according to their extended functionality and application scenarios.

Place/Transition Net (P/T Net) [1]: The Place/Transition Net, also known as P/T Net, is a fundamental type of Petri net consisting of places, transitions, and arcs. It is used to model and analyse concurrent systems by representing states (places) and events (transitions) with token flow, providing a basic framework for dynamic system behaviour.

Coloured Petri Net (CPN) [8]: The Coloured Petri Net (CPN) extends the basic Petri net by introducing Coloured tokens, which carry data or attributes. This allows for more expressive modelling of complex systems, distinguishing different token types, and enabling detailed analysis of system behaviour.

Timed Petri Net [9]: The Timed Petri Net incorporates time parameters, such as delays or durations, into transitions or places. This extension is particularly useful for modelling real-time systems where timing constraints play a critical role in system dynamics.

Stochastic Petri Net (SPN) [10]: The Stochastic Petri Net (SPN) integrates random time distributions, typically exponential, into transitions. It is widely used for performance evaluation and reliability analysis of systems with probabilistic behaviours.

Generalized Stochastic Petri Net (GSPN) [11]: The Generalized Stochastic Petri Net (GSPN) is an extension of SPN, allowing a combination of immediate transitions (with zero delay) and stochastic transitions. This enhances the modelling of systems with both deterministic and probabilistic elements.

Hierarchical Petri Net [12]: The Hierarchical Petri Net organizes Petri nets in a layered structure, making it suitable for modelling large and complex systems. This approach facilitates the decomposition and management of system components at different levels of abstraction.

Predicate/Transition Net (PrT Net) [13]: The Predicate/Transition Net (PrT Net) enhances the P/T Net by incorporating predicates and functions, increasing its expressive power. It is designed for advanced modelling of systems where logical conditions and data manipulations are essential.

Object-Oriented Petri Net [14]: The Object-Oriented Petri Net integrates object-oriented principles into the Petri net framework, making it suitable for modelling distributed and object-driven systems. This approach allows for encapsulation, inheritance, and polymorphism, enhancing the modularity and reusability of system models.

2.1.2. Place/Transition nets

In this work, we focus mainly on Place/Transition nets (P/T nets). Thereby, a Petri net is defined as:

Definition 1 (Petri net). A Petri net is a tuple PN = (P, T, A, W), where:

- P: A finite set of places, representing the states or conditions of the system.
- T: A finite set of transitions, representing events or actions that cause a change in the system's state.
- $A \subseteq (P \times T) \cup (T \times P)$ is the set of arcs that connect places to transitions or vice versa.
- W: arc weight that is a non-negative integer associated with an arc, indicating the number of tokens that are consumed (for input arcs from places to transitions) or produced (for output arcs from transitions to places) when a transition fires. If no weight is specified, it is assumed to be 1 by default.

In a Petri net, markings are fundamental because they determine which transitions are enabled and serve as a state descriptor and drive the execution semantics of

the Petri net. By tracking changes in markings over time, one can analyse the possible states and behaviours of the system through the reachability graph.

Definition 2 (Marking of Petri net). Given a Petri net N = (P, T, A, W), a marking is a function:

$$M : P \to \mathbb{N}$$

that assigns to each place $p \in P$ a non-negative integer $M(p)$, representing the number of tokens currently residing in p.

The marking M describes the state of the Petri net at a given point in time. The initial marking is denoted by M_0 which assigns an initial number of tokens to each place, representing the initial state of the system.

Example. The behaviour of a Petri net is defined by the firing of transitions, which consume tokens from input places and produce tokens in output places according to the arc weights.

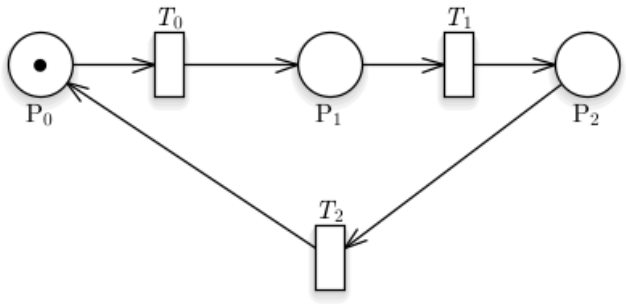


Figura 2.1: Example of Petri Net

The Petri net illustrated in Figure 2.1 consists of :

$P(places) = \{P_0, P_1, P_2\}$,

$T(transitions) = \{T_0, T_1, T_2\}$,

$A(arcs) = \{(P_0, T_0), (T_0, p1), (P_1, T_1), (T_1, P_2), (P_2, t2), (T_2, P_0)\}$,

$M_0(initialmarking) = (P_0, P_1, P_2) = (1, 0, 0)$.

Here, P_0 has one token initially and T_0 is enabled, so firing T_0 removes the token from P_0 and adds one token to P_1, then firing T_1 removes the token from P_1 and adds one token to P_2 and so on, modelling a process that consumes a resource and produces one output.

As for the markings of this Petri net in Figure 2.1, we can see that there is only one token in P_0 and three places (P_0, P_1, P_2) in total. Additionally, in conjunction with the firing of transitions and movement of tokens in places, the overall makings are $M_0(P_0, P_1, P_2) = (1, 0, 0), M_1(P_0, P_1, P_2) = (0, 1, 0), M_2(P_0, P_1, P_2) = (0, 0, 1)$.

2.2. Petri Net Product Line

2.2.1. Association from PN to PNPL

Petri Net are widely used due to their rich body of theoretical results enabling analysis, but in reality, there are many scenarios that require modelling (possibly a large set of) variants of similar systems. In this case, the designer needs to build many variations of a base model.

Software Product Line (SPL) [15] consists of a collection of software products that share common core characteristics while also varying in specific, clearly defined ways. The products of an SPL are defined and implemented in terms of features, which are subsequently combined in specific ways to obtain the final software products [7].

Based on the relevance and portability of appeal PN and SPL, [2] proposed Petri Net Product Line (PNPL) which is a formal model that combines Petri nets with software product line (SPL) engineering to support modelling and analysis of product lines in concurrent systems. It extends classical Petri nets to support variability modelling in software product lines.

PNPLs are based on the notion of a feature model that defines the variability space of possible configurations [16]. The core idea is to create a 150 % Petri net model that contains all behavioural elements of all product variants, annotated with Boolean feature expressions. With this, it enables the modelling of concurrent systems with variability, allowing the analysis of different product configurations within a single model.

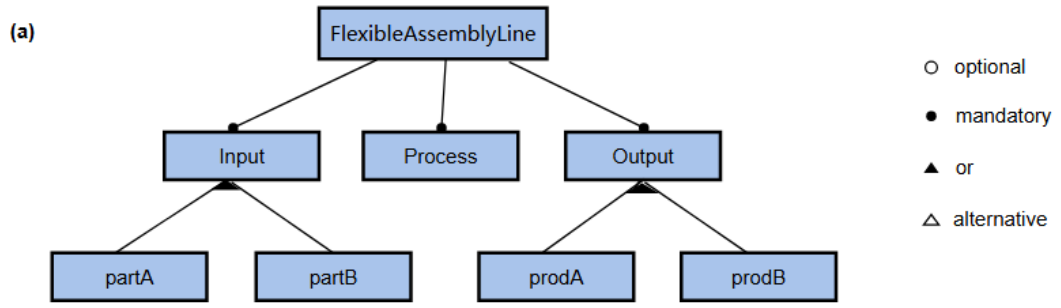
2.2.2. Feature Model

A feature model is used to explicitly represent the variability of the system. It defines a set of features that describe the common and variable aspects of the product line. These features are typically organized hierarchically and may include relationships such as mandatory, optional, alternative (XOR), or or-groups (OR). Formally, a Feature Model (FM) can be defined as follows:

Definition 3 (Feature model). A feature model $FM = (F, \Psi)$ consists of a set of propositional variables $F = \{f1, ..., fn\}$ called features and a propositional formula Ψ over the variables in F.

The feature model enables a systematic and scalable way to manage variability and generate different Petri net variants corresponding to specific product configurations.

As an illustration, the flexible assembly line depicted in Figure 2.2 is a highly configurable system designed to accommodate a versatile production process. The main purpose is to model all such possible configurations in a compact way and analyse properties of all configurations efficiently. Figure 2.2(a) shows the feature model using a diagrammatic notation and Figure 2.2(b) using Definition 3.



(b) **FM**=({FlexibleAssemblyLine, Input, Process, Output, partA, partB, prodA, prodB}, {FlexibleAssemblyLineInput /\ Process /\ Output /\ (partA \/ partB) /\ (prodA \/ prodB)})

Figura 2.2: Feature model for the flexible assembly line using (a) the diagrammatic notation, and (b) definition 3 as logical proposition

This FM can be customized to accept one or two distinct types of input parts, namely PartA and PartB, allowing flexible handling of raw materials based on production requirements. Following the input stage, the assembly line incorporates a robust quality control process to ensure the integrity and precision of the manufacturing workflow. Subsequently, it is capable of producing one or two types of finished product: ProdA and ProdB, enabling it to adapt to diverse market demands or customer specifications. This reconfigurability underscores the potential of the system for optimizing efficiency and versatility in a concurrent production environment.

2.2.3. Feature Configuration

Feature configuration is used to obtain specific Petri net from PNPL. Formally:

Definition 4 (Feature Configuration). A valid feature configuration $\rho \subseteq F$ of a feature model $FM = (F, \Psi)$ is a subset of its features satisfying Ψ. $P(FM) = \{\rho_i\}$ for the set of all valid feature configurations of PNPL.

The approach is that $\Psi[true/\rho, false/(F \setminus \rho)]$ evaluates to true when each $f \in \rho$ is substituted by true, and each $f \in F \setminus \rho$ is substituted by false.

For example, Figure 2.2 admits 9 feature configurations, Since (FlexibleAssemblyLine, Input, Process, Output) are mandatory in FM, they must be included in Feature configuration. (partA, partB) are optional on Input, (prodA, prodB) are optional on Output. And propositional formula Ψ = FlexibleAssemblyLine $\wedge$ Input $\wedge$ Process $\wedge$ Output $\wedge$ (partA $\vee$ partB) $\wedge$ (prodA $\vee$ prodB). Hence,

$\{\rho_1\}$: {FlexibleAssemblyLine, Input, Process, Output, partA, prodA}

$\{\rho_2\}$: {FlexibleAssemblyLine, Input, Process, Output, partA, prodB}

$\{\rho_3\}$: {FlexibleAssemblyLine, Input, Process, Output, partB, prodA}

$\{\rho_4\}$: {FlexibleAssemblyLine, Input, Process, Output, partB, prodB}

$\{\rho_5\}$: {FlexibleAssemblyLine, Input, Process, Output, partA, partB, prodA}

$\{\rho_6\}$: {FlexibleAssemblyLine, Input, Process, Output, partA, partB, prodB}

$\{\rho_7\}$: {FlexibleAssemblyLine, Input, Process, Output, partA, prodA, prodB}

$\{\rho_8\}$: {FlexibleAssemblyLine, Input, Process, Output, partB, prodA, prodB}

$\{\rho_9\}$: {FlexibleAssemblyLine, Input, Process, Output, partA, partB, prodA, prodB}

$P(FM) = \{\rho_i\} = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5, \rho_6, \rho_7, \rho_8, \rho_9\}$

2.2.4. PNPL Definition

A PNPL is a 150% Petri net whose elements can be annotated with boolean formulae, having as variables the features of the feature model [2], it allows modelling the variability space using a feature model, and automatically producing specific Petri nets from given feature configurations [17]. Formally:

Definition 5 (Petri net product line). A PNPL = (FM,PN,Φ), where:

- FM: a feature model $FM = (F, \Psi)$ consists of a set of propositional variables $F = \{f1, ..., fn\}$ called features and a propositional formula Ψ over the variables in F.
- PN: a Petri net PN (called the 150% Petri net).
- Φ: a mapping that consists of pairs $< x, \Phi_x >$ mapping an element $x \in P \cup T \cup A$ to a propositional formula Φ_x (called the presence condition (PC) of x) over the features in FM.

PNPL is well-formed if $\forall a \in A : (\Phi_a \rightarrow \Phi_{a_0}) \wedge (\Phi_a \rightarrow \Phi_{a_1})$ is true [2]. The well-formedness condition requires the PC of an arc (Φ_a) to be stronger than the PC of its source (Φ_{a_0}) and target (Φ_{a_1}) elements, which ensures that if the arc is present in a product Petri net, its source and target elements will be present as well.

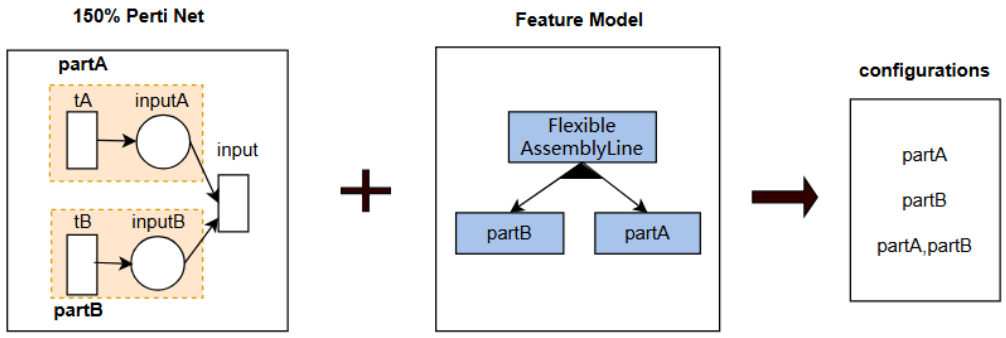


Figura 2.3: Elements of a PNPL

As illustrated in Figure 2.3, a PNPL is based on several core components. First, the variability space is captured through a feature model, which specifies the available features (i.e., partA, partB) and superimpose all possible net variants into a single unified Petri net, known as the 150 % Petri net. Each element in this net is annotated with a presence condition, which is a Boolean formula defined over the features in the feature model. To obtain a specific net variant, users select a configuration, a subset of features from the feature model. In this configuration, selected features are replaced by true and unselected ones by false. These substitutions are then applied to the presence conditions, which are evaluated as either true or false. Elements whose presence conditions are evaluated to false are removed from the 150 % net. The remaining elements form the Petri net corresponding to the selected configuration.

2.3. Reachability Graph

PNPLs inherently model systems with variability across multiple product configurations, defined by a Feature Model (FM), and their dynamic behaviour must be analysed across all valid feature combinations. However, research on the dynamic properties of PNPL is relatively limited. Hence, we propose to extend some methods for dynamic analysis of Petri nets.

2.3.1. Choice of Dynamic Analysis

Here are some methods for the dynamic analysis of Petri nets.

Reachability Graph Analysis [7]: This method involves generating all possible states of a Petri net (reachability graph, RG) to analyse dynamic properties such as reachability and liveness. Although effective for small-scale systems, it may encounter the state explosion problem.

Linear Algebraic Techniques [18]: Using state equations or P-invariants and T-invariants to analyse dynamic properties like boundedness or conservativeness, the results are suitable for structured verification.

Simulation-Based Analysis [19]: This approach evaluates dynamic behaviour by simulating the execution trajectories of a Petri net, particularly useful in Timed Petri Nets or Stochastic Petri Nets for performance analysis.

Model Checking [20]: A formal method using temporal logic to verify dynamic properties of Petri nets, ideal for analysing deadlock-freeness or liveness in complex systems.

Transition Invariant Analysis [21]: This method examines transition invariants to check the cyclic behaviour and dynamic stability of a Petri net.

In this work, we choose Reachability Graph Analysis. Here are some reasons:

1. The RG, generated from the base Petri net by exploring all possible markings through transition firings, provides a comprehensive state space representation that captures the system's concurrency and asynchronous

nature. This allows for the identification of dynamic properties such as reachability, deadlock, and liveness, which are critical for ensuring system correctness.

2. Extending the RG with feature annotations addresses the variability aspect of PNPLs and reflects how different product configurations influence state transitions. This ensures that dynamic properties are evaluated contextually, considering only the transitions enabled by a given feature combination, thus aligning the analysis with the SPL paradigm.

3. The RG serves as a bridge to advanced verification techniques, such as conversion to a Featured Transition System (FTS$^+$), which enhances scalability and supports model checking by using formal logics like fCTL. The analysis can systematically explore the state space while preserving variability information, making it an efficient and principled approach for dynamic property verification in complex, configurable systems like PNPLs.

2.3.2. Introduction to RG

Reachability Graph can be formally defined as follows:

Definition 6 (Reachability graph). Given a marked Petri Net $N = (P, T, A, W)$ and its initial marking M_0, the Reachability Graph (RG) is defined as: $RG(N) = (V, E)$, where

- V: the set of nodes, each representing a reachable marking M of the Petri Net.
- E: represents a transition firing that leads from one marking to another.

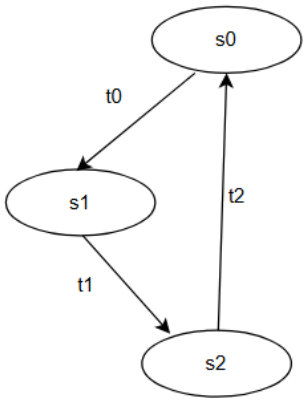


Figura 2.4: Reachability Graph

As illustrated in Figure 2.4, a Reachability Graph has V (a set of states) which is s0, s1, s2 with each state representing a reachable marking, and E (a set of transitions) which is t0, t1, t2 with each transition denoting a state change from one state to another state. The overall process is starting from s0 which is

also the initial marking M_0, and then the system enters state s1 by triggering transition t0, the transition t1 is triggered from s1 and the system enters state s2, the transition t2 is triggered from s2 and the system returns to state s0.

2.4. Featured Transition Systems (FTS$^+$)

2.4.1. Software Product Line

In Software Product Line (SPL) development, systems are designed as families with the goal of making economies of scale through systematic reuse of development artifacts [22]. The various system variants, referred to as "products", are identified in advance and a model is developed that captures their differences and shared characteristics.

Formal modelling and verification of SPL behaviour is vital for quality assurance [4], and the model checking problem involves determining whether a system adheres to a specified property expressed in temporal logic which is very suitable for SPL behaviour. In the context of Transition Systems (TS)—a widely used formalism to represent system behaviour—there are numerous model checking algorithms available. The specific features and performance of these algorithms largely depend on the type of temporal logic used to describe the desired properties.

Therefore, behaviours of individual SPL products can be represented by Transition Systems (TS) [5].

2.4.2. Transition Systems

A TS is a directed graph whose transitions are labelled with actions, and whose states are labelled with atomic propositions [4]. Formally:

Definition 7 (Transition Systems). A Transition Systems (TS) is a tuple $ts = (S, Act, trans, I, AP, L)$, where:

- S is a set of states,
- Act is a set of actions,
- trans is a set of transitions,
- I is a set of initial states,
- AP is a set of atomic propositions,
- L is a labelling function,$S \to 2^{AP}$

2.4.3. Featured Transition Systems

Featured Transition Systems (FTS$^+$) [4] extends from Transition Systems (TS) which is a common mathematical representation of system behaviour to generalize a compact model for representing the behaviours of all the products of

an software product line (SPL), where each transition is associated with a feature condition that determines its applicability. It is particularly suited for systems derived from Petri Net Product Lines (PNPL), where the feature-annotated Reachability Graph(fRG) is converted into a more abstract and efficient representation.

Definition 8 (Featured Transition Systems). An Featured Transition Systems is a tuple FTS$^+$ = (S, Act, trans,I, AP, L, d, γ), where:

- S is a set of states,
- Act is a set of actions,
- trans is a set of transitions,
- I is a set of initial states,
- AP is a set of atomic propositions,
- L is a labelling function,$S \rightarrow 2^{AP}$
- d is a feature model,
- γ is a total function, labelling each transition with a feature expression.

The advantage of FTS$^+$ is its optimized structure and compatibility with advanced verification tools (e.g., NuSMV [23]), enabling efficient symbolic analysis using BDDs (Binary Decision Diagrams) [24]). This makes FTS$^+$ particularly valuable for verifying complex properties, such as safety and liveness, across all valid configurations.

2.4.4. Derivation Example

The following example illustrates the transition from a feature model in Software Product Line (SPL) to a product-specific transition system (TS), and finally to a unified Feature Transition Systems (FTS$^+$), capturing the behaviour of a software product line (SPL) and its variants.

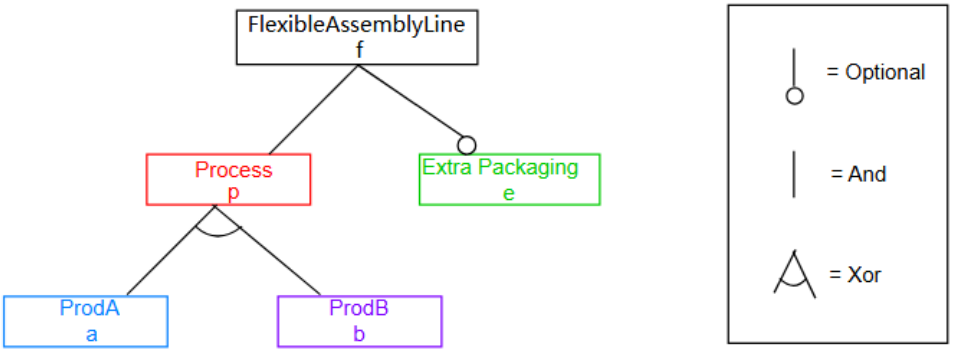


Figura 2.5: FD of a Flexible Assembly Line SPL

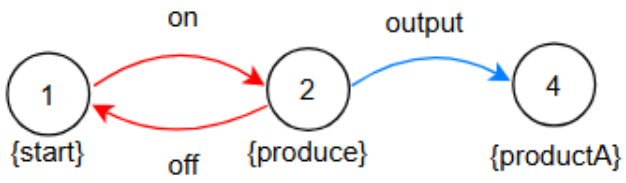


Figura 2.6: TS of product {f, p, a}

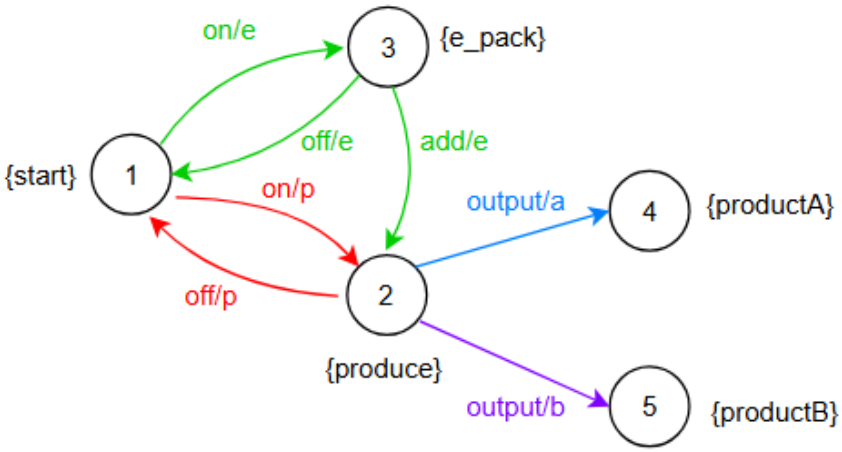


Figura 2.7: Flexible Assembly Line FTS+

Figure 2.5 depicts the feature diagram (FD) of a software product line called Flexible Assembly Line. A Feature Diagram (FD) is a concise representation for the valid products of an SPL. Boxes denote features and edges denote decomposition of features. The root feature is Flexible Assembly Line, which includes two sub-features: Process (mandatory), which has an exclusive-or (XOR) group comprising ProdA and ProdB. This means that only one of the two products can be selected in any valid configuration. Extra Packaging (optional), which can be added to any product configuration. The diagram uses standard feature modelling notation: mandatory features are connected with solid lines, optional features with empty circles, and XOR groups are denoted by an arc with a triangle.

Figure 2.6 shows the Transition System (TS) of a concrete product derived from the feature configuration f, p, a, which includes the root Flexible Assembly Line (f), the Process (p), and product ProdA (a) and it does not include ProdB or Extra Packaging. The TS captures the runtime behaviour of the selected product: The initial state (state 1) is labelled start. From state 1, the system can transition to state 2 using action on, or back to itself via off. State 2 is labelled produce and can transition to state 4 with action output, producing produc-

tA. This transition system reflects the exact behaviour of the selected feature combination and omits any behaviour related to unselected features. It is important to note that colours are used consistently across the figures to indicate the same features. For example, the red-coloured arcs in Figure 2.6 correspond to transitions associated with the Process feature (denoted as p) in the feature diagram shown in Figure 2.5.

Figure 2.7 presents the corresponding Feature Transition System Plus (FTS^+). Unlike a standard TS, the FTS^+ integrates the behaviour of all possible products in the SPL in a single transition system. Each transition is annotated with a presence condition, a Boolean expression over features, indicating in which feature combinations the transition is valid. For example: The transition on/p from state 1 to state 2 is valid only when the Process feature is included. The transition output/a from state 2 to state 4 is associated with feature ProdA. The transition add/e to state 3 models the packaging behaviour, conditioned on the presence of ExtraPackaging. By encoding feature conditions directly in the transition system, the FTS^+ provides a compact yet expressive way to support family-based model checking, enabling the verification of temporal properties over multiple configurations simultaneously.

This example clearly demonstrates the workflow from feature modelling to behavioural modelling in SPLs, showing how individual products can be derived and how their combined behaviour can be abstracted and analysed through FTS^+.

2.5. Feature Computation Tree Logic (fCTL)

The model checking problem involves determining whether a system fulfils a specified temporal logic property. For Transition Systems (TS), a widely used mathematical representation of system behaviour, various model checking algorithms exist, with their characteristics largely determined by the logic employed to define the properties. In the literature, the most prevalent logics include Linear Temporal Logic (LTL), which enables reasoning over individual paths, and Computation Tree Logic (CTL) [5], which quantifies across sets of paths.

2.5.1. Introduction to CTL

CTL is a branching-time temporal logic for expressing properties over a kripke structure [25]/ transition system [5], which formally model system behavior as a set of states with transitions and atomic propositions. These structures are foundational to model checking and temporal logic semantics. The syntax and semantics of CTL are defined over a computation tree, which represents all possible execution paths starting from an initial state.

Definition 9 (Syntax of CTL). CTL formulas are constructed from atomic propositions, logical connectives (Boolean operators), and temporal operators, which include path quantifiers and state quantifiers. The components are as follows:

- Atomic Propositions (AP): Basic truth assignments, such as a, representing properties of system states.

- Logical Connectives: $\neg$ (negation), $\wedge$ (conjunction).
- Path Quantifiers: **A** Holds on all paths, **E** Holds on at least one path
- State Quantifiers: **X**: Next state. **F**: Eventually. **G**: Globally. **U**: Until.

CTL quantifies over all possible execution paths in a computation tree, making it well-suited for describing the behaviour of concurrent systems. A CTL algorithm determines the collection of states that fulfil the property under examination.

However, when verifying Software Product Lines (SPLs), the concept of satisfaction is contingent upon the specific products within the SPL. A property is considered valid for a given state only with respect to a particular subset of those products. Therefore, in [4], Featured Computation Tree Logic (fCTL) [4] was proposed as an extension of Computation Tree Logic (CTL) tailored for the formal verification of properties in Software Product Lines (SPLs) and systems with feature variability.

Introduced as a response to the need to reason about dynamic behaviour across different product variants, fCTL integrates the branching-time semantics of CTL with the concept of features—specific configurable elements or options within a product line. This makes it particularly suitable for analysing systems modelled using Featured Transition Systems (FTS), which represent the behaviour of SPLs by incorporating feature expressions. The logic enables the specification and verification of properties such as reachability, liveness, and safety, tailored to specific feature configurations, making it a powerful tool for modern software engineering.

2.5.2. Syntax and Semantics of fCTL

fCTL is obtained by augmenting CTL formulae with product quantifiers, also called guards. Formulae without a product quantifier (and thus pure CTL) implicitly range over all products [4]. Its formal definition is built upon a computation tree derived from a Featured Transition System (FTS$^+$). Below is a detailed breakdown of fCTL's syntax and semantics.

The syntax of fCTL is given as follows:

Definition 10 (Syntax of feature Computation Tree Logic). A fCTL formula is defined as:

- a if $a \in AP$;
- $\neg\phi$, $\phi \wedge \psi$ if ϕ and ψ are fCTL formula expression;
- $[\chi_{px}]\mathbf{AX}\phi$ | $[\chi_{px}]\mathbf{EX}\phi$ | $[\chi_{px}]\mathbf{AF}\phi$ | $[\chi_{px}]\mathbf{EF}\phi$ | $[\chi_{px}]\mathbf{AG}\phi$ | $[\chi_{px}]\mathbf{EG}\phi$ | $[\chi_{px}]\mathbf{A}[\phi\mathbf{U}\psi]$ | $[\chi_{px}]\mathbf{E}[\phi\mathbf{U}\psi]$ if $px \subseteq \mathcal{P}(N)$.

Where:

- $N = \{f_1, f_2, \ldots, f_n\}$ denote the set of features in a SPL.

- $\mathcal{P}(N)$ refer to the valid product set.

- $\chi_{px}(x_1, \ldots, x_n)$ is a Boolean function over the set of features N, which we have $\chi_{px} : \{0,1\}^{|N|} \to \{0,1\}$, is a symbolic encoding of a set of products $px \subseteq \mathcal{P}(N)$.

- $[\chi_{px}]$ indicates a symbol-encoded product set such that

$$[\chi_{px}] \triangleq \{p \in px \mid \chi_{px}(x_1, \ldots, x_n) = 1 \, , x_i = 1 \iff f_i \in p \text{ for } i \in [1, n]\}.$$

In general, the $[\chi_{px}]$ acts as a guard, ensuring the enclosed formula is evaluated only in configurations where $px \subseteq \mathcal{P}(N)$.

Theoretically, the feature set N can generate 2^n feature subsets via binary choices. However, due to constraints imposed by the feature model (e.g., dependencies, mutual exclusions), only a subset of these theoretical combinations correspond to deployable products, which is $\mathcal{P}(N)$.

Example. From Figure 2.5 and Figure 2.7, we extract the following features:

- f: FlexibleAssemblyLine (mandatory)

- p: Process (core manufacturing process)

- e: Extra Packaging (optional packaging)

- a: ProdA (Product A, tied to p, ProdA and ProdB are exlusive)

- b: ProdB (Product B, tied to p, ProdA and ProdB are exlusive)

Using these constraints, we list all valid feature subsets (products) in $\mathcal{P}(N)$:

1. **Empty set** (no features selected): $\varnothing$

2. **Process + Product A** (no packaging): $\{f, p, a\}$

3. **Process + Product A + Extra Packaging**: $\{f, p, a, e\}$

4. **Process + Product B** (no packaging): $\{f, p, b\}$

5. **Process + Product B + Extra Packaging**: $\{f, p, b, e\}$

Thus,

$$N = \{f, p, a, b, e\}$$

$$\mathcal{P}(N) = \{\varnothing,\ \{f, p, a\},\ \{f, p, a, e\},\ \{f, p, b\},\ \{f, p, b, e\}\}$$

Now, if we want to varify property in Product B (b), then

$\implies$ $[b]$ = $[\chi_{px}]$

$\implies$ $\chi_{px}(f, p, a, b, e)$

$\implies$ $\chi_{px}(x_1, x_2, x_3, x_4, x_5)$ with $f = x_1, p = x_2, a = x_3, b = x_4, e = x_5$

$\implies$ $\chi_{px}(x_1, x_2, x_3, x_4, x_5) = x_1 \wedge x_4 = 1$ as x_1 is mandatory

$\implies$ $\{f, p, a\} : x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 0, x_5 = 0 \to x_1 \wedge x_4 = 1 \wedge 0 = 0$ unsatisfied;
$\{f, p, a, e\} : x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 0, x_1 = 0 \to x_1 \wedge x_4 = 1 \wedge 0 = 0$ unsatisfied;
$\{f, p, b\} : x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1, x_5 = 0 \to x_1 \wedge x_4 = 1 \wedge 0 = 0$ satisfied;
$\{f, p, b, e\} : x_1 = 1, x_2 = 1, x_3 = 0, x_4 = 1, x_5 = 1 \to x_1 \wedge x_4 = 1 \wedge 1 = 1$ satisfied;

$\implies [\chi_{px}] = \{\{f, p, b\},\ \{f, p, b, e\}\}$

Hence,

$[b]\mathbf{AX}\phi$ means in all next states, ϕ holds when $[b]$ is satisfied.

$[b]\mathbf{EX}\phi$ means there exists a next state where ϕ holds when $[b]$ is satisfied.

$[b]\mathbf{AF}\phi$ means along all paths, ϕ eventually holds when $[b]$ is satisfied.

$[b]\mathbf{EF}\phi$ means there exists a path where ϕ eventually holds when $[b]$ is satisfied.

$[b]\mathbf{AG}\phi$ means along all paths, ϕ always holds (globally) when $[b]$ is satisfied.

$[b]\mathbf{EG}\phi$ means there exists a path where ϕ always holds when $[b]$ is satisfied.

$[b]\mathbf{A}[\phi\mathbf{U}\psi]$ means On all paths, ϕ holds until ψ holds when $[b]$ is satisfied.

$[b]\mathbf{E}[\phi\mathbf{U}\psi]$ means There exists a path where ϕ holds until ψ holds when $[b]$ is satisfied.

The semantics of fCTL are defined over a Featured Transition System (FTS$^+$), Its semantics is defined as follows.

Definition 11 (Semantics of an *fCTL* formula over FTS$^+$). wrt. an FTS Fts, a state $s \in S$, and a set of products $px \subseteq \mathcal{P}(N)$.

$$s, px \models \phi \quad \Leftrightarrow \quad \forall p \in px,\ s, p \models \phi$$

Where $p \in px$ is a product and satisfaction of a formula in a state s and a product p:

- $(Fts, s, p) \models a$ iff $a \in L(s)$.
- $(Fts, s, p) \models \neg\phi$ iff $(Fts, s, p) \not\models \phi$.
- $(Fts, s, p) \models \phi \wedge \psi$ iff $(Fts, s, p) \models \phi$ and $(Fts, s, p) \models \psi$.
- $(Fts, s, p) \models [\chi_{px}]\mathbf{EX}\phi$ iff $p \in [\chi_{px}]$ and there exists a path $\pi = s_0, s_1, \ldots$ (where $s_0 = s$) such that $(Fts, s_1, p) \models \phi$.
- $(Fts, s, p) \models [\chi_{px}]\mathbf{AX}\phi$ iff $p \in [\chi_{px}]$ and for all paths $\pi = s_0, s_1, \ldots$ (where $s_0 = s$), $(Fts, s_1, p) \models \phi$.
- $(Fts, s, p) \models [\chi_{px}]\mathbf{EF}\phi$ iff $p \in [\chi_{px}]$ and there exists a path $\pi = s_0, s_1, s_2, \ldots$ (where $s_0 = s$) such that there exists some $i \geq 0$ where $(Fts, s_i, p) \models \phi$.
- $(Fts, s, p) \models [\chi_{px}]\mathbf{AF}\phi$ iff $p \in [\chi_{px}]$ and for all paths $\pi = s_0, s_1, s_2, \ldots$ (where $s_0 = s$), there exists some $i \geq 0$ where $(Fts, s_i, p) \models \phi$.
- $(Fts, s, p) \models [\chi_{px}]\mathbf{EG}\phi$ iff $p \in [\chi_{px}]$ and there exists a path $\pi = s_0, s_1, s_2, \ldots$ (where $s_0 = s$) such that for all $i \geq 0$, $(Fts, s_i, p) \models \phi$.
- $(Fts, s, p) \models [\chi_{px}]\mathbf{AG}\phi$ iff $p \in [\chi_{px}]$ and for all paths $\pi = s_0, s_1, s_2, \ldots$ (where $s_0 = s$), for all $i \geq 0$, $(Fts, s_i, p) \models \phi$.
- $(Fts, s, p) \models [\chi_{px}]\mathbf{E}[\phi\mathbf{U}\psi]$ iff $p \in [\chi_{px}]$ and there exists a path $\pi = s_0, s_1, s_2, \ldots$ (where $s_0 = s$) such that there exists some $i \geq 0$ where $(Fts, s_i, p) \models \psi$, and for all $0 \leq j < i$, $(Fts, s_j, c) \models \phi$.

- $(Fts, s, p) \models [\chi_{px}]\mathbf{A}[\phi \mathbf{U} \psi]$ iff $p \in [\chi_{px}]$ and for all paths $\pi = s_0, s_1, s_2, \ldots$ (where $s_0 = s$), there exists some $i \geq 0$ where $(Fts, s_i, p) \models \psi$, and for all $0 \leq j < i$, $(Fts, s_j, p) \models \phi$.

Example. An example property for Flexible Assembly Line FTS$^+$ in Figure 2.7 is

$$[\neg b]\mathbf{AG}\neg productB$$

which expresses that "in absence of the ProdB feature (b), it is impossible to activate productB". This property is satisfied by the model, since the transition leading to the productB state is not present if the ProdB is not included. In other words, for every product without feature (b), all states could not satisfy expression "productB".

Capítulo 3

Convert PNPL into feature-annotated RG

This chapter presents the formal process of translating a Petri Net Product Line (PNPL) into a reachability graph (RG) enriched with feature annotations. The feature-annotated RG (fRG) serves as a bridge between the variability-aware Petri net model and the subsequent construction of a Featured Transition System (FTS$^+$), which is used for fCTL property verification.

3.1. Conversion Idea

The Reachability Graph (RG) of a Petri Net is a state-space representation that captures all possible markings reachable from the initial marking through firing transitions. Each node in the graph represents a reachable marking, and each edge corresponds to the firing of a transition that leads from one marking to another.

In Petri Net Product Line (PNPL), transitions, places, or arcs can be associated with feature expressions that determine their activation under different product configurations. To perform dynamic property verification (e.g., liveness, safety) in the presence of variability, it is necessary to generate a comprehensive behavioural representation of the PNPL.

When extending this concept to a Petri Net Product Line (PNPL), where elements of the net are annotated with presence conditions over features, the construction of the RG must account for variability. In this case, we annotate each transition in the RG with its corresponding feature condition, resulting in a feature-annotated RG.

To generate this extended RG, the following steps should be taken:

1. Begin with the 150 % Petri Net, along with its initial marking.
2. For each enabled transition $t \in T$ in a marking M, compute the successor marking M' as usual.

3. Annotate the edge $(M \xrightarrow{t} M')$ in the RG with a presence condition χ, derived from the conjunction of all presence conditions involved in the firing (including those of places, arcs, and the transition).

4. Repeat the above steps recursively for all reachable markings from M_0.

3.2. RG Generation

Before dealing with those processes above, we need to understand the process of constructing a reachability graph (RG) from a Petri Net (PN). Hence, we present a simplified case study and this example will serve as the foundation for understanding how the PNPL-to-RG transformation extends the basic Petri net semantics with variability.

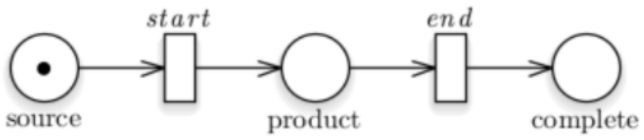


Figura 3.1: Petri Net of simple product line

The Petri net in Figure 3.1 consists of the following components:

- Places: source, product, complete
- Transitions: start, end
- Arcs: All arcs have weight = 1

Now, we construct Reachability Graph, the basic construction rule is that the RG is constructed by exploring all markings reachable from the initial marking M^0 through enabled transitions. In this example, M^0 = [source = 1, product = 0, complete = 0], indicating that the source place starts with one token, while product and complete are initially empty.

For transition firing rules, a transition t is enabled in a marking M if: $\forall p \in pre(t), M(p) \geq W(p,t)$. Upon firing, the new marking M' is computed by: Consuming tokens from input places based on arc weights, producing tokens into output places based on arc weights.

Here is the specific transformation steps:

Start Marking	**Transition**	**Enabled**	**Next Marking**
M^0 = [1, 0, 0]	start	Yes	M^1 = [0, 1, 0]
M^1 = [0, 1, 0]	end	Yes	M^2 = [0, 0, 1]

Since there is no more enabled transition, we get the final reachability graph as Figure 3.2.

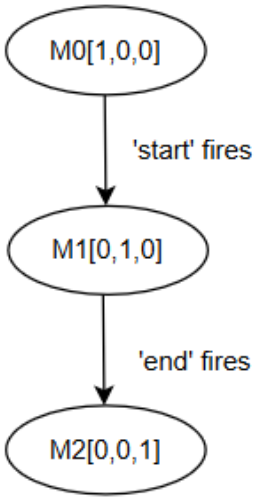


Figura 3.2: Reachability Graph of a simple product line

This RG is the base model that, once extended with feature annotations, enables reasoning about software product lines as shown in PNPL models.

3.3. Adding Feature Annotations to RG

In order to verify conversion idea, the translation process considers three scenarios of increasing complexity:

- Case 1: Single token and 1-weighted arc
- Case 2: Multiple tokens and 1-weighted arc
- Case 3: Multiple tokens and arbitrary weighted arc

3.3.1. Case 1: Single token and 1-weighted arc

This is the simplest case. Each place in the PNPL holds at most one token, and all input/output arcs have a weight of 1. The behaviour closely resembles classical reachability analysis in safe Petri nets.

To illustrate the Case 1 conversion, we consider a simplified product line scenario where a system is capable of producing two types of products: Product A and Product B. This scenario is modelled as a PNPL in Figure 3.3 where:

- There is a shared “source” place, which initially holds one token.
- Two production paths are available: The upper path is guarded by feature A, producing Product A; The lower path is guarded by feature B, producing Product B.
- Two feature-annotated branches: Branch A: [A] $\rightarrow$ startA $\rightarrow$ productA $\rightarrow$ endA $\rightarrow$ complete; Branch B: [B] $\rightarrow$ startB $\rightarrow$ productB $\rightarrow$ endB $\rightarrow$ complete.

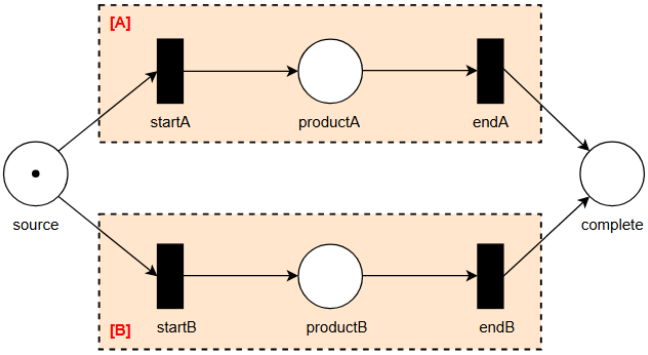


Figura 3.3: Petri Net Product Line of Single token and 1-weighted arc

- Each path contains three transitions: start, produce, and end.
- All arcs in the PNPL model have weight = 1.

In this example, the initial marking is M^0 = [source = 1, productA = 0, productB = 0, complete = 0], for simplify, we can also mark it as M^0 = [1,0,0,0].

First, we consider the PNPL as a complete Petri net, ignoring its features. Then convert this Petri net into reachability graph following the previous idea of PN to RG. After that, we label the corresponding transition in that reachability graph with the features to which it belongs. Then we get the feature-annotated reachability graph as Figure 3.4.

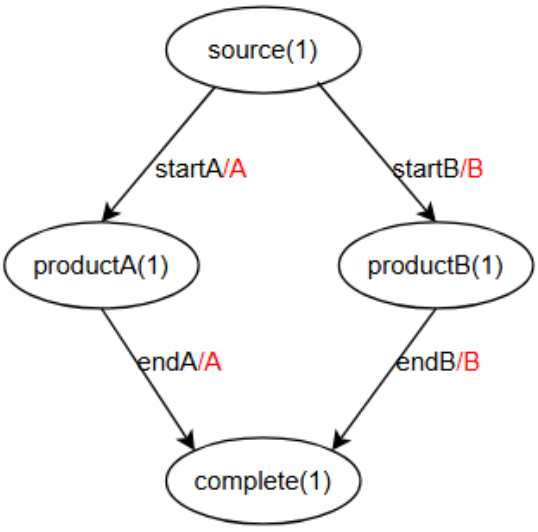


Figura 3.4: Feature-annotated RG of Single token and 1-weighted arc

The marking naming convention in Figure 3.4 is as follows:

source(1) = [source = 1, productA = 0, productB = 0, complete = 0] = [1,0,0,0].

productA(1) = [source = 0, productA = 1, productB = 0, complete = 0] = [0,1,0,0].

productB(1) = [source = 0, productA = 0, productB = 1, complete = 0] = [0,0,1,0].

complete(1) = [source = 0, productA = 0, productB = 0, complete = 1] = [0,0,0,1].

In Figure 3.4, feature-annotated RG has two path: source(1) → productA(1) → complete(1), source(1) → productB(1) → complete(1). Among them, the number 1 in parentheses after place means that there is currently 1 token in this place. There also has two transitions (startA and endA) in the productA path and two transitions (startB and endB) in the productB path. Additionally, Features are labelled in red next to the relative transition, for instance, Feature A is is labelled in red next to the transition startA and endA.

Observing from the feature-annotated RG, we get some conclusions that is the system can non-deterministically choose to produce A or B first, depending on which transition is fired first and since the initial marking has one tokens in the source place, both products A and B can be produced independently. Transitions are labelled with features [A] or [B], preserving the variability semantics.

3.3.2. Case 2: Multiple tokens and 1-weighted arc

To illustrate the Case 2 transformation, we extend previous product line example, the only difference is that there are two tokens in the source place which is shown as Figure 3.5.

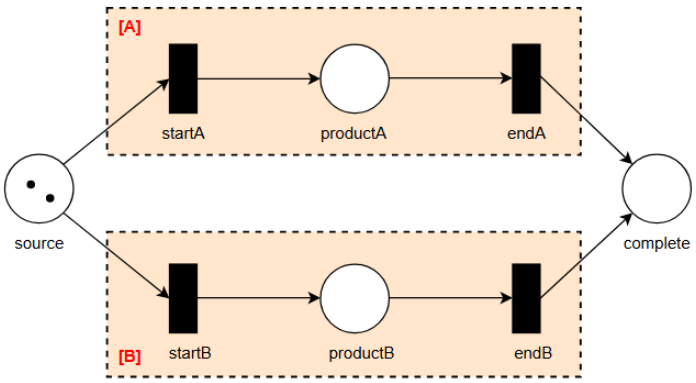


Figura 3.5: Petri Net Product Line of Multiple tokens and 1-weighted arc

Still, we consider the PNPL as a complete Petri net, ignoring its features. Then convert this Petri net into reachability graph following the previous idea of PN to RG. After that, we label the corresponding transition in that reachability graph

with the features to which it belongs. The final feature-annotated RG is as Figure 3.6.

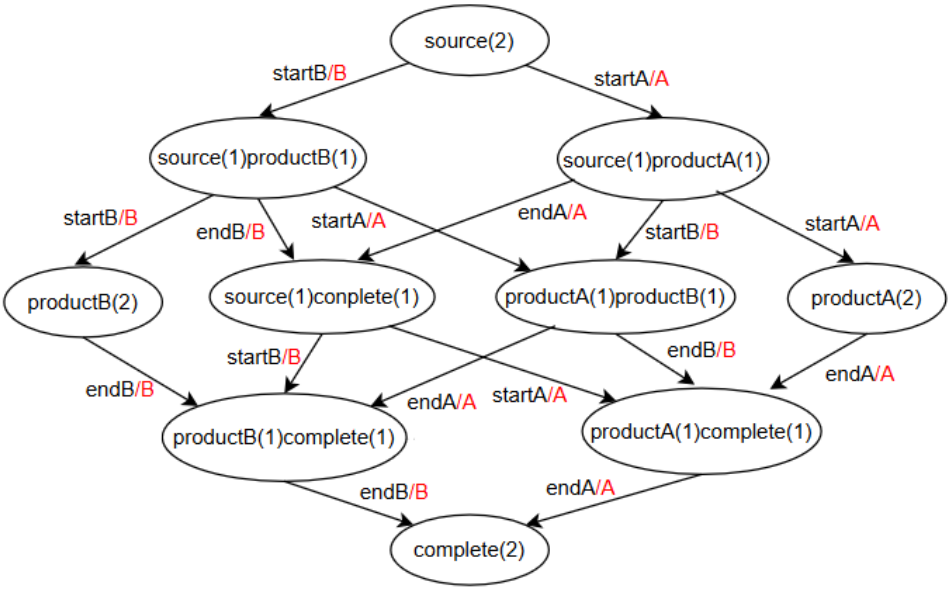


Figura 3.6: Feature-annotated RG of Multiple tokens and 1-weighted arc

The marking naming convention in Figure 3.6 is as follows:

source(2) = [source = 2, productA = 0, productB = 0, complete = 0] = [2,0,0,0].

source(1)productB(1) = [source = 1, productA = 0, productB = 1, complete = 0] = [1,0,1,0].

source(1)productA(1) = [source = 1, productA = 1, productB = 0, complete = 0] = [1,1,0,0].

productB(2) = [source = 1, productA = 0, productB = 2, complete = 0] = [0,0,2,0].

source(1)complete(1) = [source = 1, productA = 0, productB = 0, complete = 1] = [1,0,0,1].

productA(1)productB(1) = [source = 0, productA = 1, productB = 1, complete = 0] = [0,1,1,0].

productA(2) = [source = 0, productA = 2, productB = 0, complete = 0] = [1,0,0,0].

productB(1)complete(1) = [source = 0, productA = 0, productB = 1, complete = 1] = [0,0,1,1].

productA(1)complete(1) = [source = 0, productA = 1, productB = 0, complete = 1] = [0,1,0,1].

complete(2) = [source = 0, productA = 0, productB = 0, complete = 2]
= [0,0,0,2].

In this example, we can observe that states and paths have increased a lot compared to the previous example simply because of the addition of a token in the "source", which could have caused a state space explosion if more tokens were added. Due to the well-known problem of state-space explosion [1], generation of the reachability set and reachability graph with the known approaches often becomes intractable even for moderately sized nets [26].

There are some key observations from the feature-annotated RG: The system can choose to produce A or B first non-deterministically, depending on which transition is fired first. Since the initial marking has 2 tokens in the source place, both products A and B can be produced independently or in sequence.

The state space includes all combinations of intermediate markings: States where only A is produced and States where only B is produced and States where both are produced in various orders. Transitions are labelled with features [A] or [B], preserving the variability semantics. In this paper, in order to prevent the state space problem, some invalid arcs/states should be removed. These arcs/states present invalid behaviour (i.e., moving to ProductA when we have selected B products).

3.3.3. Case 3: Multiple tokens and arbitrary weighted arc

In Case 3 transformation, except using previous product line example, we also add five tokens in the source place and make arc weight from source to productA is equal to 2 and arc weight from source to productB is equal to 3 separately, which is shown as Figure 3.7.

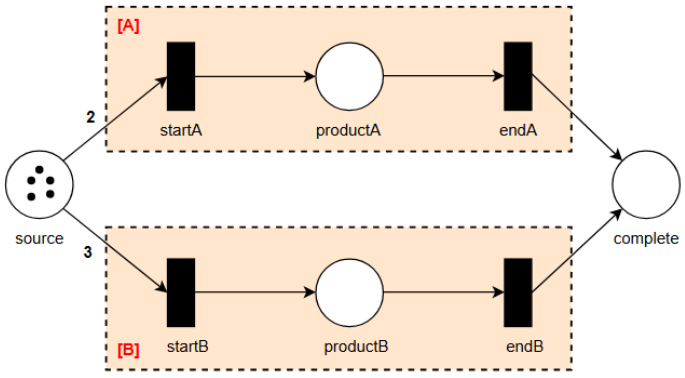


Figura 3.7: Petri Net Product Line of Multiple tokens and arbitrary weighted arc.

Following the previous method, we get feature-annotated RG which is shown as Figure 3.8.

The marking naming convention in Figure 3.8 is as follows:

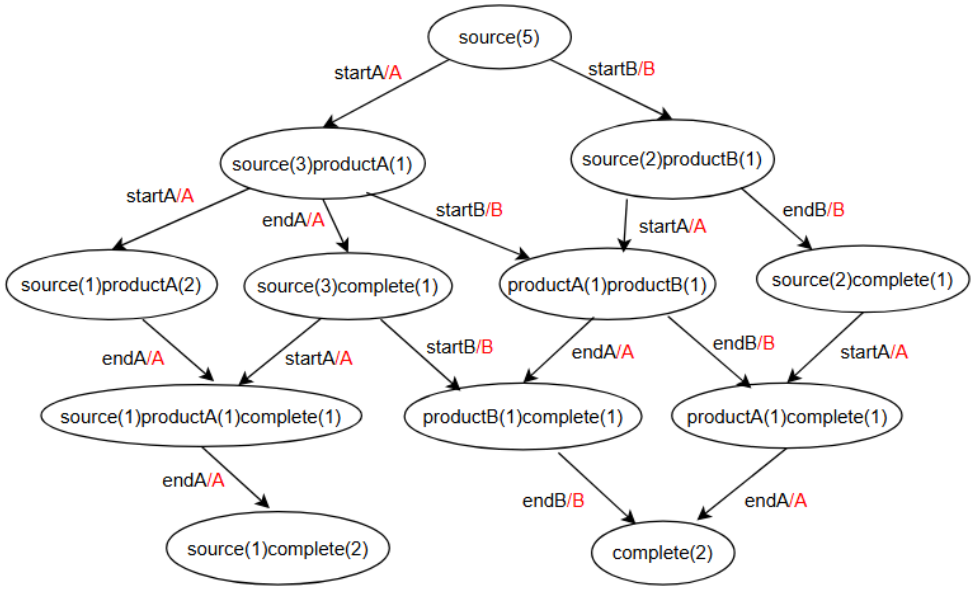


Figura 3.8: Feature-annotated RG of Multiple tokens and arbitrary weighted arc

source(5) = [source = 5, productA = 0, productB = 0, complete = 0] = [5,0,0,0].

source(3)productA(1) = [source = 3, productA = 1, productB = 0, complete = 0] = [3,1,0,0].

source(2)productB(1) = [source = 2, productA = 0, productB = 1, complete = 0] = [2,0,1,0].

source(1)productA(2) = [source = 1, productA = 2, productB = 0, complete = 0] = [1,2,0,0].

source(3)complete(1) = [source = 3, productA = 0, productB = 0, complete = 1] = [3,0,0,1].

productA(1)productB(1) = [source = 0, productA = 1, productB = 1, complete = 0] = [0,1,1,0].

source(2)complete(1) = [source = 2, productA = 0, productB = 0, complete = 1] = [2,0,0,1].

source(1)productA(1)complete(1) = [source = 1, productA = 1, productB = 0, complete = 1] = [1,1,0,1].

productB(1)complete(1) = [source = 0, productA = 0, productB = 1, complete = 1] = [0,0,1,1].

productA(1)complete(1) = [source = 0, productA = 1, productB = 0, complete = 1] = [0,1,0,1].

source(1)complete(2) = [source = 1, productA = 0, productB = 0, complete = 2] = [1,0,0,2].

complete(2) = [source = 0, productA = 0, productB = 0, complete = 2] = [0,0,0,2].

In this figure, we can also see that the states and paths have increased exponentially, so the number of states and paths depends on the number of tokens and arc weight in PNPL. For example, for the same token, the smaller the weight, the more states and the more complex the feature-annotated RG; for the same weight, the more tokens, the more states and the more complex the RG too.

On the other hand, with the increment of states and transitions, it will lead to state space explosion. Hence, in response to this problem, we thought deeply and got a suitable strategy to verify it, that is, pruning strategies.

3.4. Adding pruning strategies

In the process of constructing a feature-annotated Reachability Graph (fRG) from a PNPL model, special care must be taken to handle the correctness of state generation, particularly under configurations with multiple tokens and overlapping transitions. It is possible that a single marking enables multiple transitions that belong to distinct feature expressions. If these transitions are independently enabled by different tokens, the resulting marking M' may implicitly combine features that are mutually exclusive. This leads to semantically invalid states when verifying fCTL over the model. For instance, the markings marked in orange in Figure 3.9, their input arcs have two different exclusive features, then we can not fire transition startB if we have previously selected Feature A, likewise, we cannot fire transition startA if we have previously selected Feature B.

To address this, we introduce a conflict detection filter during feature-annotated RG construction. The filter prevents the generation of markings that are the result of executing transitions from conflicting features. For example, Let M $\rightarrow$ M' be a transition path. If M' is derived from applying two or more transitions in parallel or sequence that originate from incompatible feature expressions, then M' must be excluded from the state space.

The states and transitions labelled in orange in the Figure 3.9 are the ones that contain multiple features, and it is clear from the appeal strategy that this part needs to be prune away.

This approach preserves the semantic integrity of the RG and ensures that the resulting FTS^+ is compatible with well-defined product configurations in the feature model. It avoids false positives in fCTL verification caused by invalid mixed-feature states. What is more, it reduces the number of states, thus optimizing the structure and making the problem of state explosion somewhat alleviated.

Now we propose a formally definition of feature-annotated Reachability Graph(fRG) as following:

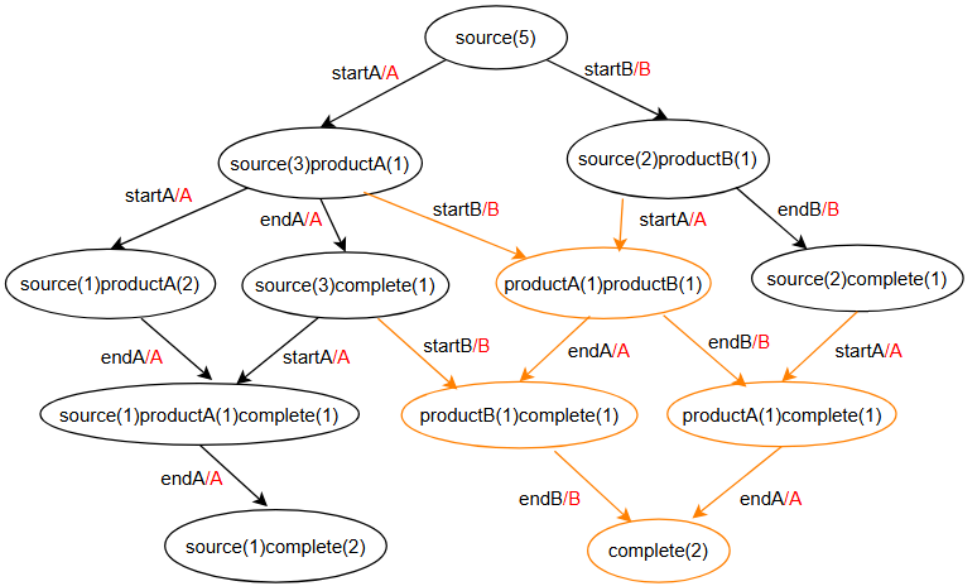


Figura 3.9: Feature-annotated RG of pruning strategies

Definition 12 (feature-annotated Reachability Graph). Given a PNPL = (FM,PN,Φ), feature-annotated Reachability Graph (fRG) is defined as: $fRG\,(PNPL) = (V, E, f)$, where

- V, E are defined as in Definition 6
- f: A subset of Φ in PNPL, but only assigns features (defined by the Feature Model FM) to each transition $t \in E$

3.5. Algorithm

3.5.1. Algorithm Construction

Having the fundamentals covered, we can proceed to the algorithmic foundation for transforming Petri Net Product Lines (PNPL) into Reachability Graphs with features. We begin by revisiting the classical method for constructing reachability graphs from basic Petri nets, and then extend this technique to handle feature variability as defined in PNPLs. The transformation is illustrated progressively through three incremental cases, reflecting increasing complexity. Finally, we introduce a pruning strategy to eliminate semantically invalid markings that arise from incompatible feature interactions.

To ground the transformation process, we first describe the standard algorithm for generating a reachability graph from a basic Petri net.

Initialization : Start with the initial marking M^0, which assigns a number of tokens to each place in the Petri net. Add M^0 to the state set S and to a processing queue.

Main Loop (Repeat until no new markings are found): First, select a marking M from the queue, then for each transition t in the Petri net, check whether t is enabled at M, which is for every input place p of t, confirm that: $M(p) \geq W(p,t)$. If not, t cannot fire. If enabled, compute the resulting marking M', that is subtract tokens from input places according to input arc weights and add tokens to output places according to output arc weights. After that, Add an edge (M, t, M') to the graph. If M' is new (not yet in S), add it to the state set and queue.

Termination : The algorithm terminates when all reachable markings have been explored. The resulting RG is a directed graph (V, E), which V = reachable markings (states) and E = transitions between markings.

To adapt the above method to PNPL models, we introduce transformation algorithm of three cases. From that three cases, we can know that regardless of the number of tokens and the size of arc weight in PNPL, as long as it satisfies $M(p) \geq W(p,t)$, then firing the transition can be performed. Therefore, in this algorithm, we continue to follow the appeal algorithm, and the difference is that we need to add a set of feature inputs, and then in the main loop to defines the feature acquisition function that labels the transitions in the edges with features when the edges are added.

For the pruning strategies, we know that certain markings may emerge where multiple transitions with mutually exclusive feature annotations are enabled in the same state. This leads to unreachable or invalid system variants. Hence, the prune rule should be at any marking M, if two enabled transitions t1 and t2 originate from distinct, disjoint feature expressions (e.g., [A] and [B] where $A \wedge B$ is unsatisfiable), their resulting merged markings are pruned. So we insert a conflict check before enqueuing any new marking to satisfy pruning strategies, maintain a trace of features used along each path, and discard markings with conflicting feature lineage.

3.5.2. Algorithm Description

Let us now go through the complete algorithm 1 to give more details.

Algorithm 1 Algorithm for transforming PNPL into feature-annotated RG

```
Inputs: initial marking(M0),transitions(ts),arcs(as),places(p),
features(fs)
Outputs: A marking set(V) and transition set(T) of
feature-annotated reachability graph(fRG)
1: Vt := null, P := null, fRG := null
2: P.append(M0)
3: Vt.add(M0)
4: fRG.V.add(M0)
5: while P != null {
6:    M = P.poll()
7:    for each t in ts do
8:       ia := t.getinputArc(as)
```

```
9:        oa := t.getoutputArc(as)
10:        ip := t.getinputArc(as).getPlace(p)
11:        op := t.getoutputArc(as)getPlace(p)
12:        if M.getToken(ip)>= ia.getWeight() then {
13:           M′ := M
14:           M′.decreaseToken(ip, ia.getWeight())
15:           M′.increaseToken(op, ia.getWeight())
16:           if M′.inputfeature(fs) = 1 then {
17:              fRG.T.add(new TransitionEdge(M,t,t.getfeature,M′))
18:              if (!Vt.contains(M′)) then {
19:                   fRG.V.add(M′)
20:                   Vt.add(M′)
21                    P.add(M′) }
22:           }
23:        }
24: }
25: return fRG
```

Algorithm 1 accepts the input of PNPL elements such as initial marking, transitions, arcs, places and features. Then output feature-annotated reachability graph, which is made of a set of paths through intern process. The internal process is illustrated as Figure 3.10.

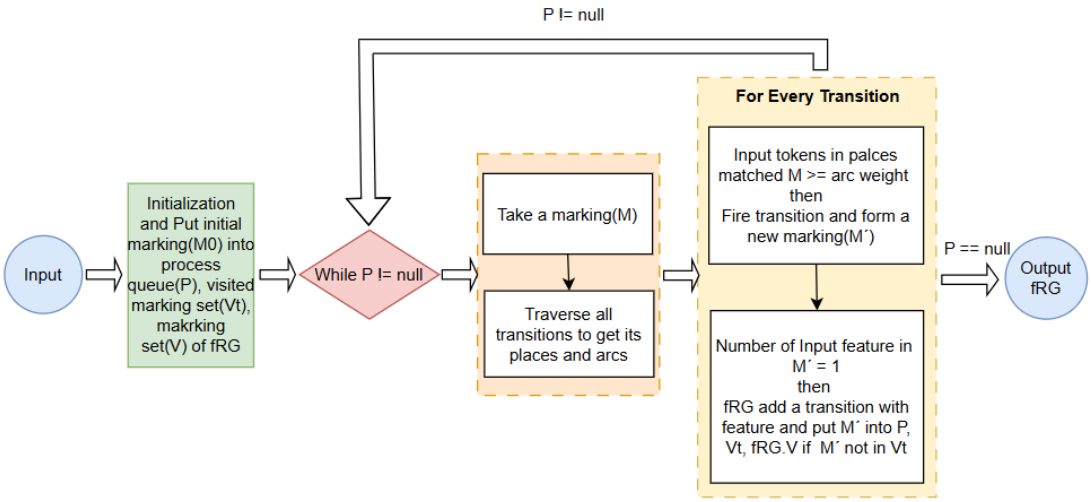


Figura 3.10: Process in Algorithm 1

Program explanation.

Firstly, this algorithm initializes the data structures to manage visited marking set (Vt), the processing queue (P), and a feature-annotated reachability graph (fRG)(line 1). It begins with the initial marking M0 and adds M0 to the processing queue (P), marking set (Vt), and marking set (V), then it has a loop to determine if processing queue is empty; if not, it will keep running(lines 2–4).

In this loop, the algorithm first dequeues a marking M and explores all the enabled transitions t at each iteration. It checks whether t can be fired from M by verifying that each input place of the transition has sufficient tokens (lines 5–11). If t is enabled, a new marking M' is generated by consuming tokens from input places and producing tokens in output places according to arc weights (lines 12–15). After that, The algorithm adds a check function to check if the new marking M' is from multiple different features (line 16). If not, then retrieving the feature associated with the transition t, and using it to construct a feature-annotated edge from M to M'. This edge is added to the feature-annotated reachability graph (fRG, line 17). To ensure each marking is processed only once, the algorithm adds M' to the processing queue only if it is not already in the visited state set (lines 18–21).

Finally, the algorithm returns a feature-annotated reachability graph. The complete set of algorithms provides a robust foundation for extracting dynamic behaviour from PNPLs while preserving variability semantics, ultimately preparing the model for formal verification using fCTL.

Remark. In this algorithm, the approach of applying pruning techniques represents a significant advancement, as it proactively prevents the insertion of unnecessary arcs and states rather than expanding the entire feature-annotated Reachability Graph (fRG) and subsequently removing invalid elements. This dynamic pruning strategy markedly enhances efficiency by avoiding the generation of semantically invalid configurations, such as those resulting from mutually exclusive features such as [A] and [B] in the product line example. The benefits of this method are substantial: It substantially reduces computational cost by minimizing the memory and processing resources required during the construction of fRGs, leading to faster verification times.

Additionally, it constrains the size of the fRG in terms of the number of states, preventing the exponential growth typically associated with state space explosion in concurrent systems. This streamlined process not only improves scalability for complex Petri Net Product Lines (PNPLs) with multiple tokens or features but also ensures that the resulting Featured Transition System (FTS$^+$) remains lean and focused, facilitating more effective and precise fCTL-based analysis of dynamic properties such as reachability, liveness, and safety across valid configurations.

3.5.3. Algorithm Complexity

Algorithm 1 constructs a feature-annotated reachability graph (RG$_f$) from a Petri Net Product Line (PNPL), by exploring enabled transitions and generating new markings. This process is similar to a symbolic breadth-first search over the marking space. Below, we provide a detailed analysis of its time complexity.

Notation. Let us define:

- $|T|$: number of transitions in the PNPL
- $|A|$: number of arcs in the PNPL

- $|P|$: number of places in the PNPL
- $|M|$: number of reachable markings (states)

High-Level Structure. The algorithm follows a breadth-first search pattern over the marking space:

1. It initializes a queue P with the initial marking M_0.
2. While the queue is not empty, it:
   - Polls a marking M.
   - Iterates over all transitions $t \in T$ to check if t is enabled.
   - If t is enabled in M, generates a new marking M', modifies tokens, and adds an annotated transition to fRG.
   - If M' is new, it is added to the queue.

Per-Iteration Cost. Each marking M may be visited once:

- For each marking M, up to $|T|$ transitions are checked.
- Checking enabling conditions and computing M' involves token and arc weight comparisons: $O(1)$ for each transition (assuming input/output arc access is constant time).
- If a transition is enabled, computing the new marking requires modifying token counts on a small number of places: $O(1)$.
- Checking if the feature condition is satisfied involves evaluating a Boolean expression: assume cost $O(1)$.
- Inserting new transitions or markings is $O(1)$ amortized if we use hash sets.

Thus, per marking the cost is $O(|T|)$.

Total Time Complexity.

$$O(|M| \cdot |T|)$$

Remarks. The algorithm avoids full unfolding for each feature configuration and instead builds a symbolic RG that embeds variability via transition annotations. However, if the number of reachable markings grows exponentially with the number of tokens or places, the algorithm may still face state explosion in practice.

Capítulo 4

Convert feature-annotated RG into FTS$^+$

To enable formal verification using fCTL over models derived from PNPL, it is necessary to transform the feature-annotated Reachability Graph (RG) into a Featured Transition System (FTS$^+$). The RG contains behavioural semantics with embedded feature expressions that indicate the presence of variability. However, RGs are not sufficient for fCTL verification because they lack structure over configurations, atomic propositions, and consistent feature model representation. FTS$^+$ bridges this gap.

This section presents the final step in the modelling pipeline: converting feature-annotated RG derived from a PNPL into a formal Featured Transition System (FTS$^+$). This conversion prepares the model for fCTL-based property verification by organizing behaviour and variability into a uniform representation. This transformation is essential for enabling fCTL verification, as fCTL is defined over FTS$^+$ structures.

4.1. Conversion Strategies

The goal of this transformation is to preserve:

- Behaviour: All markings and transitions from the RG.
- Variability: Feature expressions guarding each transition.
- Verifiability: Incorporation of atomic propositions and feature constraints.

The conversion is guided by aligning the structure of the RG with the components of the FTS$^+$ tuple: FTS$^+$ = (S, Act, trans,I, AP, L, d, γ). To achieve this, we propose following strategies:

Conversion condition:

1. Given a Petri net product line PNPL = (FM,PN,Φ) where:
   - FM: a feature model

- PN: a Petri net that is a tuple PN = (P, T, A, W), where P and T are disjoint sets of places and transitions, and A is the set of arcs connecting either places to transitions or vice versa, W is the weight on the arcs.
- Φ: a mapping that maps very element in $P \cup T \cup A$ to a propositional formula over features in FM

2. Obtaining the set of feature configurations from FM, which is $P(FM) = \{\rho_i\}$

3. Deriving from that PNPL and get a feature-annotated Reachability Graph fRG = (V, E, f) where:

- V: reachable markings (states)
- (E,f): labelled transitions (M, t, f, M') with feature annotation $f \in \Phi$ where t $\in$ T of PNPL

Conversion steps:

Step 1: Define Core Elements

- Let S := {num(V)} (a set of numbers of reachable markings in fRG)
- Let Act := { t | (M, t, f, M') $\in$ fRG.(E,f) }
- Let trans := { (M, t, M') | (M, t, f, M') $\in$ fRG.(E,f) }
- Let γ((M, t, M')) := f
- Let I := { M^0 } = 1 (number of the initial marking)

Step 2: Define Atomic Propositions

- Let AP := V (same to reachable markings in fRG initially, could be less depending on requirements)
- Define labelling L(s): $S \to 2^{AP}$, which attach every state with a label of AP

Step 3: Integrate Feature Model

- d := P(FM) (the set of valid feature configurations)
- γ labels each transition with a relative feature expression, which present (M, t, f, M') $\in$ fRG

The output is a complete FTS$^+$ that:

- Preserves transition semantics from the feature-annotated RG
- Carries forward variability via feature guards γ
- Enables the projection and evaluation of fCTL formulas

4.2. Conversion Example

To illustrate the conversion, we refer to the PNPL of case 3 above shown in Figure 4.1 and the corresponding feature-annotated RG (fRG) shown in Figure 4.2.

This fRG has already been pruned to remove any markings that result from feature conflicts (e.g., mixed execution of A and B).

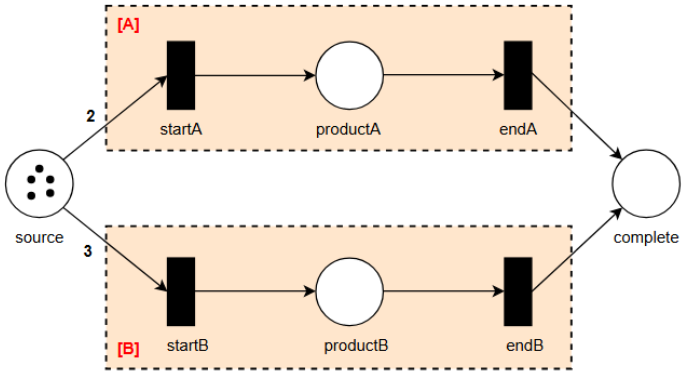


Figura 4.1: Petri Net Product Line of multiple tokens and arbitrary weighted arc

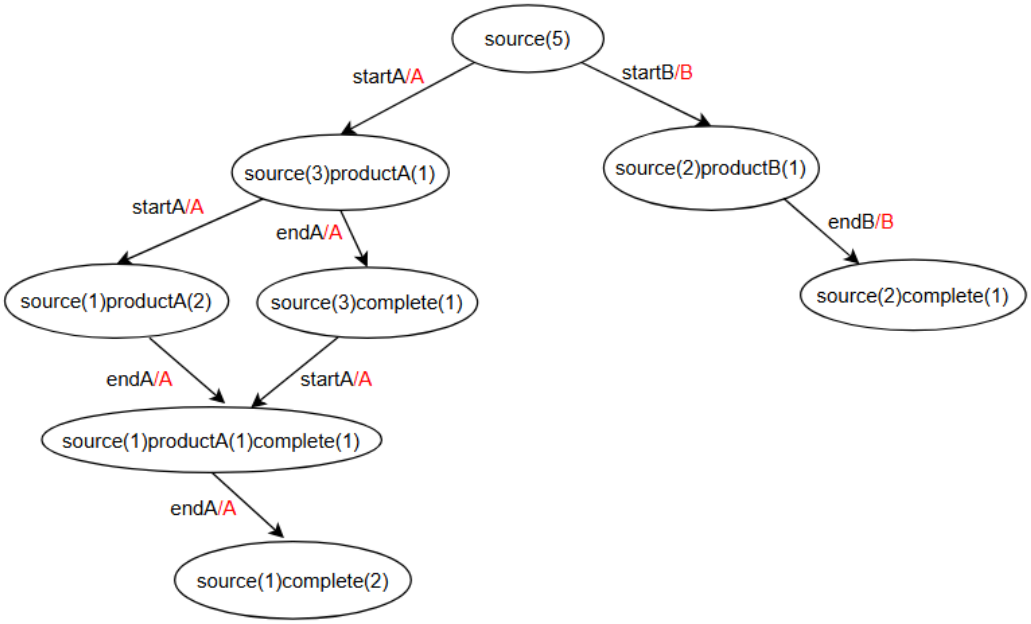


Figura 4.2: Feature-annotated RG

Now, Accoding to conversion strategies, we can construct FTS^{+}:

- S $:= \{num(V)\}$
  $:= \{1, 2, 3, 4, 5, 6, 7, 8\}$
- Act $:= \{t|(M, t, f, M') \in fRG.(E, f)\}$
  $:= \{startA, endA, startB, endB\}$

- trans $:= \{(M, t, M') | (M, t, f, M') \in fRG.(E, f)\}$
  $:= \{(1, startA, 4), (4, startA, 5), (4, endA, 6), (5, endA, 7), (6, startA, 7), (7, endA, 8), (1, startB, 2), (2, endB, 3)\}$

- I $:= \{M^0\}$
  $:= \{source(5)\} = \{1\}$

- AP $:= V$
  $:= \{source(5), source(3)productA(1), source(2)productB(1), source(1)productA(2), source(3)complete(1), source(2)complete(1), source(1)productA(1)complete(1), source(1)complete(2)\}$

- L :$L(1) = \{source(5)\}$;
  $L(4) = \{source(3)productA(1)\}$;
  $L(2) = \{source(2)productB(1)\}$;
  $L(5) = \{source(1)productA(2)\}$;
  $L(6) = \{source(3)complete(1)\}$;
  $L(3) = \{source(2)complete(1)\}$;
  $L(7) = \{source(1)productA(1)complete(1)\}$;
  $L(8) = \{source(1)complete(1)\}$

- d $:= P(FM)$
  $:= \{\{A\}, \{B\}\}$

- γ :$\gamma(1, startA, 4) = A$;
  $\gamma(1, startB, 2) = B$;
  $\gamma(4, startA, 5) = A$;
  $\gamma(4, endA, 6)) = A$;
  $\gamma(2, startA, 3) = B$;
  $\gamma(5, endA, 7) = A$;
  $\gamma(6, startA, 7) = A$;
  $\gamma(7, endA, 8)) = A$

As illustrated in Figure 5.1, the diagram of FTS$^+$ derived from PNPL in Figure 4.2 and fRG in Figure 5.1. There are eight circles with states(S) from 1 to 8 in it, initial state(1), four actions (starA, startB, endA, endB), eight transitions with feature annoted by γ, eight atomic propositions(AP) presented along with circles by label function(L).

The structure of this FTS$^+$ is similar to that of fRG, but the only difference is that FTS$^+$ have more elements such as AP, L, γ and so on.

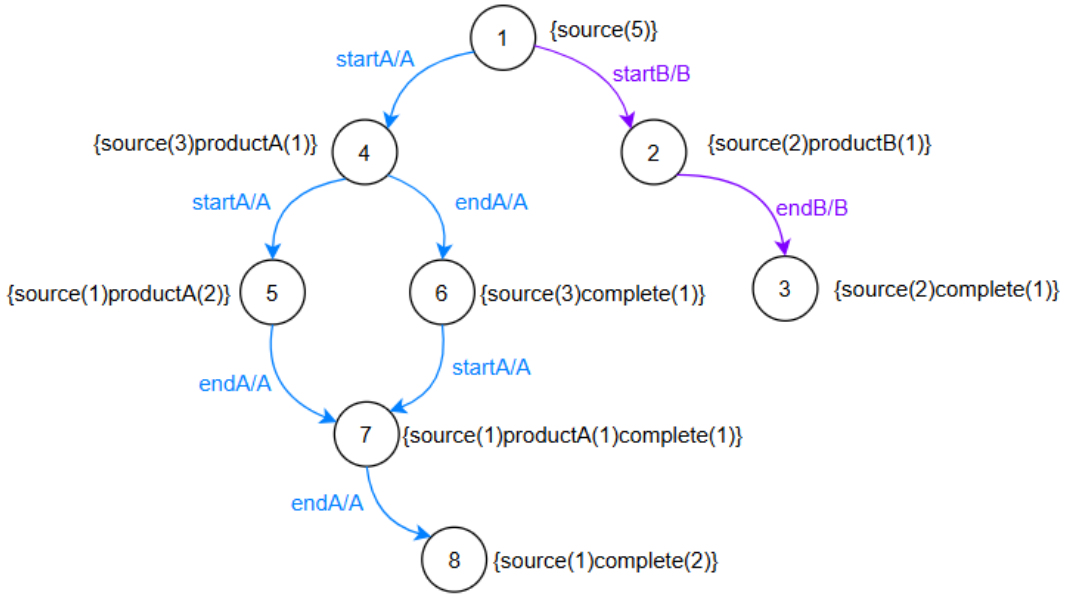


Figura 4.3: FTS+

4.3. Algorithm

4.3.1. Algorithm Description

Once we understand how individual fRG elements map into FTS$^+$ components, it becomes necessary to describe how this transformation can be automated. This section introduces an algorithmic procedure that implements the fRG-to-FTS$^+$ conversion in a scalable and consistent manner. The Algorithm 2 captures all the structural rules defined earlier, and is suitable for integration with model-checking.

Let us now go through the complete Algorithm 2 to give more details.

Algorithm 2 Algorithm for transforming feature-annotated RG into FTS$^+$

```
Inputs: A set of Transitions(T) in feature-annotated RG(fRG), A
set of markings(V) in feature-annotated RG(fRG), a set of feature
configurations(P(FM)), initial marking(M0)
Outputs: A tuple of Featured Transition Systems(FTS+)
1: S := null, Act := null, trans := null, AP := null, d := null
   I := null, P := null, Vt := null, markingToState := null
2: P.append(M0), Vt.add(M0)
3: num = 1
4: While P != null {
5:     currentM = P.poll()
6:     markingToState[currentM] = num
7:     S.add(num)
```

```
8:     num = num + 1
9:     for each (M, t, feature, M') in fRG.T then {
10:        if M == currentM and M' not in Vt then {
11:           Vt.append(M'), P.append(M') }
12:    }
13: }
14: for each (M, t, feature, M') in fRG.T then {
15:    Act.add(t)
16:    s = markingToState[M]
17:    s' = markingToState[M']
18:    trans.add( (s, t, s') )
19:    r[(s, t, s')] := feature
20: }
21: for each m in fRG.V then {
22:    AP.add(m)
23:    s = markingToState[m]
24:    L[s] := m
25: }
26: I.add('1')
27: d := P(FM)
28: FTS+ := (S, Act, trans, I, AP, L, d, r)
29: return FTS+
```

Algorithm 2 accepts the input of a list of feature-annotated RG, feature model, a set of PNPL places, and initial marking. Then output Featured Transition System structure. The internal process is illustrated as Figure 4.4.

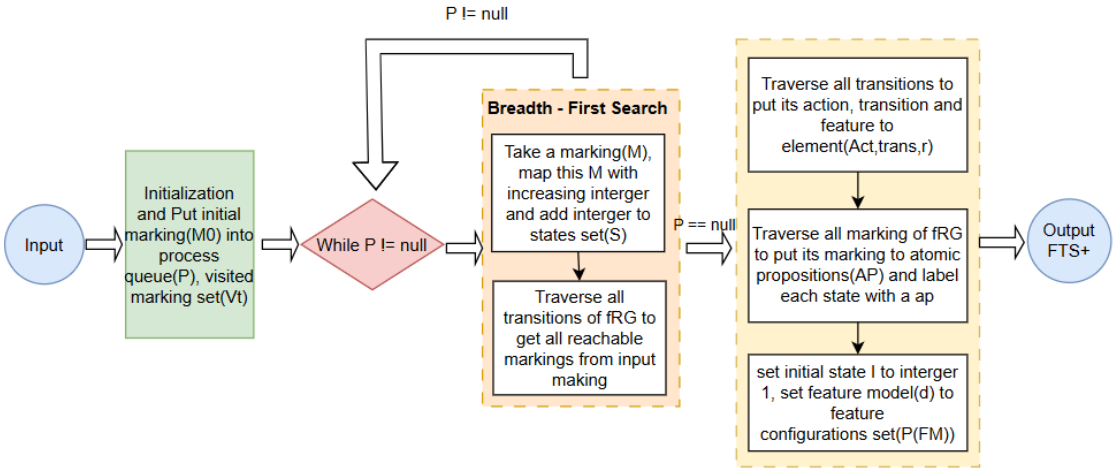


Figura 4.4: Process in Algorithm 2

Program explanation.

Firstly, the algorithm initializes empty sets for states (S), actions (Act), transitions (trans), atomic propositions (AP), feature model (d), initial state (I), process queue

(P), visited set (Vt) and markingToState set (line 1). Starting from the initial marking (M0) (Line 2), ensure that the state enumeration begins with the initial condition of the system, and add M0 to the visited set (Vt), then give the serial number (num) an initial digit 1 (line 3).

After that, it uses Breadth - First Search (BFS) to label states with order number. In the while loop, it first dequeues a marking to currentM and uses the function markingToState to map this marking with a serial number, then adds this serial number to the states set, and discovers all reachable markings by enqueuing unvisited target markings M (Lines 4-11). By this approach, it makes each state follow the transition path in ascending order.

Next, the algorithm iterates over each tuple (M, t, feature, M') in fRG(line 14) where each transition consists of the identifier of a source marking (M), a transition (t), a feature condition, and a target marking (M'). For each iteration, it adds the transition t to the action set (Act) and the triple (s, t, s') to the transition set (trans) where s and s' are the integer IDs of markings (line 15-18). The feature condition associated with each transition is stored in a mapping r (line 19), which will later define the feature labelling.

Additionally, the algorithm iterates over each marking(m) in fRG(line 21). For each iteration, it adds the markings to the atomic proposition set (AP) and assigns m as the label for the state s (via L [s]) (lines 22-25), so each state is associated with its original marking.

Finally, it sets the initial state set (I) to contain the initial marking M0 whose integer IDs is 1, links the feature model P(FM) to d, constructs the FTS^+ tuple (S, Act, trans, I, AP, L, d, r), and line 12 returns the completed FTS^+ model (lines 26-29).

This algorithm systematically transforms the feature-annotated RG into a complete FTS^+ structure, The resulting model integrates behaviour and variability, allowing direct evaluation of fCTL formulas over product families derived from PNPL models and enabling the analysis of configurable concurrent systems by integrating state properties and feature variability.

4.3.2. Algorithm Complexity

Algorithm 2 performs a series of operations over the structure of the fRG, mapping its markings and annotated transitions to states and transitions in the FTS^+. Below, we analyze its computational time complexity.

Notation. Let us define:

- $|V|$: number of markings (states) in fRG
- $|T|$: number of transitions in the fRG

Step-by-step analysis.

1. **Lines 1: Initialization.** Initializing empty data structures (sets, queues, mappings) takes constant time $O(1)$. **Total time:** $O(1)$
2. **Lines 2–13: BFS-Driven State Enumeration.** For each edge (M, t, χ, M') in RG_f, the algorithm:
   - The BFS traversal processes each reachable marking exactly once. In the worst case (all markings are reachable), this involves $|V|$ iterations of the outer loop.
   - For each marking 'currentM', the inner loop (Lines 9–12) iterates over all transitions in T to check if they originate from 'currentM'. In the worst case, this processes all $|T|$ transitions across all iterations.
   - Operations like enqueuing ('P.append'), dequeuing ('P.poll'), and checking membership in the queue or mapping are $O(1)$ with efficient data structures.

   Total time: $O(|V| + |T|)$
3. **Lines 14–20: Transition and Action Population.** This loop iterates over all $|T|$ transitions in the fRG. **Total time:** $O(|T|)$
4. **Lines 21–26: Atomic Propositions and Labelling.** This loop iterates over all $|V|$ markings in the fRG. **Total time:** $O(|V|)$
5. **Lines 27–30: Final Initialization and Assembly.** Adding the initial state to I, assigning 'd := P(FM)', and assembling the FTS+ tuple are all constant-time operations. **Total time:** $O(1)$

Total Time Complexity.

$$O(|V| + |T|)$$

This linear complexity with respect to the number of RG edges and markings makes the transformation scalable to moderately large product lines. The use of shared transitions and symbolic feature expressions also avoids the need for variant-by-variant unfolding.

Capítulo 5

Applying fCTL on FTS$^+$

The application of Feature Computation Tree Logic (fCTL) to an enhanced Featured Transition System (FTS$^+$) enables the formal verification of properties across all valid configurations of a Petri Net Product Line (PNPL). Building on the conversion of a feature-enriched Reachability Graph (fRG) into FTS$^+$ (as detailed in Chapter 4), this section explores the methodology for applying fCTL to verify temporal and variability properties. The methodology is illustrated through case studies, highlighting the scalability and correctness of the approach.

5.1. Verification Framework

Based on the research results of [4] , which studies symbolic algorithms, introduces fCTL logic and implements a toolkit based on NuSMV. The verification of PNPL dynamic properties using fCTL on FTS$^+$ follows a structured framework, leveraging the alignment between fCTL's semantics and FTS$^+$'s structure:

1. **Property Specification**: Translate system requirements (e.g., "no deadlock in configurations with feature *A*") into fCTL formulas, using feature guards to restrict properties to relevant configurations.
2. **Model Preparation**: Ensure the FTS$^+$ is well-formed, with states, transitions, and feature annotations preserved from the feature-annotated RG.
3. **Model Checking**: Apply symbolic model checking techniques [4] to evaluate fCTL formulas over the FTS^+, leveraging tools like NuSMV [4] adapted for feature-aware verification.
4. **Result Analysis**: Interpret model checking outputs to identify valid/invalid configurations, providing insights into property violations and their root causes (e.g., conflicting features).

Note that in this work, we only focus on the theoretical part in applying fCTL on FTS$^+$, and it does not involve part of applying symbolic model checking techniques and using tools like NuSMV, as it has already been demonstrated in study [4].

5.2. Mapping fCTL to FTS$^+$ Components

fCTL extends CTL with product quantifiers (guards) to reason about sets of products in SPL. The syntax and semantics, as defined in the reference (Definition 10 and 11), allow formulas to specify properties that hold for specific feature configurations.

To apply fCTL, we explicitly map its syntax to FTS^+ elements, ensuring each logical construct aligns with the system's behavioural and variability semantics:

- **Atomic Propositions (AP)**: Correspond to FTS$^+$ state labels $L(s)$, which describe marking properties (e.g., $source(5)$ indicating 5 tokens in the source place).
- **Feature Guards** $[\chi_{px}]$: Link to FTS^+'s feature model d and transition annotations γ, restricting property evaluation to configurations $px \subseteq P(FM)$ (e.g., $[A]$ limits verification to configurations including A).
- **Path Quantifiers (A/E)**: Traverse FTS^+ transitions $trans$, with " all paths (A) " requiring properties to hold for every transition sequence, and " exists a path (E) " requiring at least one valid sequence.
- **Temporal Operators (X/U/G/F)**:
  - $EX\phi$: Checks if there exists a next state (via $trans$) satisfying ϕ under the current feature configuration.
  - $E[\phi_1 U \phi_2]$: Verifies a path where ϕ_1 holds until ϕ_2 is satisfied.
  - $AG\phi$: Ensures ϕ holds globally (for all reachable states) in all paths.

5.3. Case Study: Verifying a Product Line

We demonstrate fCTL verification using the product line PNPL (introduced in Chapter 3), whose FTS$^+$ is derived in Chapter 4.

From this FTS$^+$ in Figure 5.1, it includes features A, B with valid configurations(products) $d = P(FM) = \mathcal{P}(N) = \{\{A\}, \{B\}\}$.

The meaning for each atomic proposition(AP):

$source(5)$: 5 resources pending

$source(3)productA(1)$: 3 resources pending and 1 productA on process

$source(2)productB(1)$: 2 resources pending and 1 productB on process

$source(1)productA(2)$: 1 resources pending and 2 productA on process

$source(3)complete(1)$: 3 resources pending and 1 product completed

$source(2)complete(1)$: 2 resources pending and 1 product completed

$source(1)productA(1)complete(1)$: 1 resources pending and 1 productA on process and 1 product completed

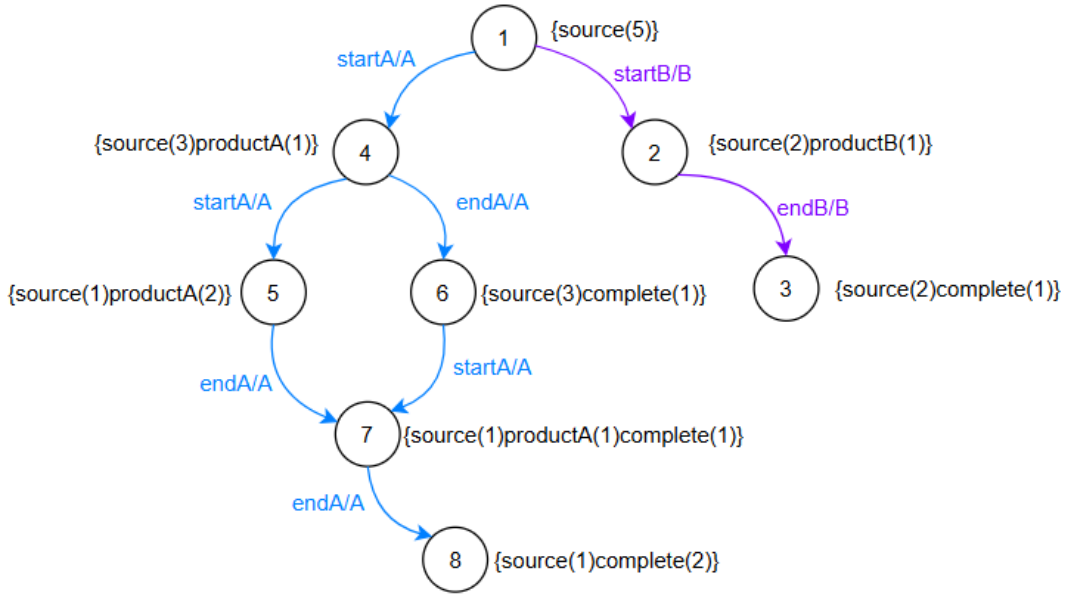


Figura 5.1: FTS+

$source(1)complete(2)$: 1 resources pending and 2 products completed

Now we use fCTL to express properties that vary by product. The syntax includes guards $[\chi_{px}]$ to restrict formulas to specific product sets $px \subseteq \mathcal{P}(N)$.

5.3.1. Property 1: Product Reachability

Requirement: For all products with A, there exists a path where a state of " 2 products completed " is eventually reached.

fCTL Formula: $[A]EFsource(1)complete(2)$

- $[A]$: Restricts to configurations with A.
- $EFsource(1)complete(2)$: there exists a path eventually reach a state where $source(1)complete(2)$ is satisfied.

Verification Steps:

1. Extract the FTS^+ for the product line, focusing on transitions annotated with A (via γ).
2. Evaluate $EFsource(1)complete(2)$ over states reachable from the initial state $s_1 = \{source(5)\}$.
3. Check if there is a path from s_1 in A configurations terminate in a state labeled $source(1)complete(2)$.

Result: The property holds. All A configurations include transitions $7 \rightarrow endA \rightarrow 8$, ensuring $source(1)complete(2)$ is reachable.

5.3.2. Property 2: Product Safety

Requirement: For all products with A, it is always globally true that $productA$ and $productB$ are not simultaneously produced.

fCTL Formula: [A]$AG(\neg(source(3)productA(1) \wedge source(2)productB(1)))$

- $[A]$: Restricts to configurations with A.
- $AG(\neg(source(3)productA(1) \wedge source(2)productB(1)))$: For all paths, $(source(3)productA(1)$ and $source(2)productB(1)$ are not satisfied at the same.

Verification Steps:

1. Extract the FTS^+ for the product line, focusing on transitions annotated with A (via γ).
2. For each state in these configurations, check if transitions are enabled.
3. Ensure no state satisfies $(source(3)productA(1) \wedge source(2)productB(1))$ in any path.

Result: The property holds. Since $source(3)productA(1)$ and $source(2)productB(1)$ are disjoint, and no transition enables both.

5.3.3. Property 3: Product protection

Requirement: Ïn configurations with B but no A, it is not possible to reach a state with $source(1)complete(2)$.

fCTL Formula: $[B \wedge \neg A]AG(\neg(source(1)complete(2))$

- $[B \wedge \neg A]$: Restricts to configurations with B no A.
- $AG(\neg(source(1)complete(2))$: For all paths, $\neg(source(1)complete(2)$ holds.

Verification Steps:

1. Extract the FTS^+ for the product line, focusing on transitions annotated with B but no A (via γ).
2. Check for all paths on configurations like $\{B\}$ (no $\{A\}$).
3. Ensure no state satisfies $(source(1)complete(2)$ in any path.

Result: The property holds. Since $source(1)complete(2)$ is reachable with guards A, there is no path to reach that state under guards B.

The case studies demonstrate that fCTL on FTS$^+$ effectively verifies dynamic properties across PNPL configurations.

5.4. Summary

This chapter applies fCTL to FTS$^+$ derived from PNPLs, providing a formal pathway to verify dynamic properties. The methodology is validated through case

studies, showing efficiency gains over traditional approaches. By integrating variability into temporal logic verification, this work enables rigorous analysis of configurable concurrent systems.

Capítulo 6

Related Work

The research presented in this work builds upon a rich foundation of work in Petri net theory, software product line (SPL) engineering, and formal verification, with a specific focus on extending these paradigms to handle variability and dynamic properties. This section reviews the key contributions that inform our approach to applying Feature Computation Tree Logic (fCTL) on Featured Transition Systems (FTS^{+}) derived from Petri Net Product Lines (PNPLs).

The verification of Petri nets and their extensions has been a subject of extensive research. [7] laid the groundwork for Petri net theory, while [27] provided detailed methods for reachability analysis. Software product line (SPL) are a software engineering methodology for developing a family of related systems by leveraging a common set of features and assets. There are several mechanisms to model variability for SPL and most of them can be classified into annotation-based and composition-based techniques [28]. Here are some works have added variability to Petri nets using SPL techniques.

Feature Petri Nets (FPNs) [29] proposed a framework combining Petri nets with feature models, known as Feature Nets. By adding propositional formulas to Petri net elements, it enables dynamic enabling and disabling of features, laying the foundation for feature annotation in SPL. As an extension of Feature Petri Nets, FPNs focus on embedding propositional formulas into Petri net elements (places and transitions) to dynamically enable or disable features. The primary goal is to model systems where feature activation/deactivation can be controlled at runtime, providing a flexible framework for SPLs with dynamic variability.

Dynamic Feature Petri Nets (DFPNs) [30] is an extension of FPNs, DFPNs introduce runtime reconfiguration mechanisms (e.g., CONNECT/DISCONNECT transitions) to adjust feature states during execution. This makes them suitable for analysing dynamic properties in systems where feature states evolve, such as adaptive or context-aware applications.

Petri Net Product Lines(PNPLs) are designed to model a family of Petri nets representing an SPL, where variability is captured through a Feature Model (FM) and a labelling function λ that annotates transitions with feature conditions.

The focus is on statically defining all possible product configurations and their behaviours.

The concept of PNPLs was pioneered by [2], who proposed a framework to model variability in concurrent systems, building on earlier work in software product lines. This approach has been extended to support slicing and configuration analysis, but verification techniques remain underdeveloped. [3] explored feature modelling in depth, providing tools for managing configuration spaces, yet their focus was on static analysis rather than dynamic behaviour. Regarding dynamic properties analysis of PNPLs, research on that is relatively limited.

Therefore, we decided to refer to one of the methods for analysing dynamic properties of PN and then choose an appropriate method for feature extension. After comparing Reachability Graph Analysis and Linear Algebraic Techniques [31], Simulation-Based Analysis [32], Model Checking [33] and Transition Invariant Analysis [34], we chose Reachability Graph Analysis as research object. The main reason is that the RG of a Petri net systematically generates all possible states (markings) by firing enabled transitions from the initial marking M_0, offering a complete representation of the state space. This is essential for PNPLs, which require the analysis of dynamic behaviour across all valid feature configurations, ensuring that properties such as reachability, deadlock, and liveness can be verified across the entire product line. Additionally, the RG analysis methods for Petri nets, developed by pioneers like [27] and [7], include mature techniques such as state equations and coverability graphs. Although originally designed for classical Petri nets and do not extend to feature-annotated models, these methods are adapted for PNPLs through feature-extended RGs, leveraging their reliability to address variability-aware systems effectively.

Computation Tree Logic (CTL) is a branching-time temporal logic used for specifying and verifying properties of systems modelled as state-transition systems, such as finite state machines, Petri nets, and other concurrent systems. Introduced by Emerson and Clarke in [35], CTL is particularly well-suited for reasoning about the possible behaviours of a system over time, capturing both safety and liveness properties in a concise and formal manner, providing a branching-time framework for verifying dynamic properties of concurrent systems. It extends classical propositional logic with temporal operators that quantify over computation paths (sequences of states) emanating from a given state, making it a powerful tool for model checking.

Feature Computation Tree Logic (fCTL) is an extension of Computation Tree Logic (CTL) designed to specify and verify properties of software product lines (SPLs) by incorporating variability into the branching-time temporal logic framework. It was developed to address the challenges of verifying properties across multiple system configurations, where traditional CTL is insufficient due to its lack of support for feature-based variability. Introduced by [4], fCTL enhances CTL's ability to reason about concurrent system behaviours by adding product quantifiers that account for the diverse configurations defined by a Feature Model (FM). This makes it particularly suitable for analysing systems modelled as Featured Transition Systems (FTS) or their enhanced versions (FTS$^+$).

In the realm of transition systems, [4] also introduced FTS as a versatile model for SPLs, demonstrating its utility in verifying feature-dependent properties. Likewise, FTS^+ is a compact model for representing the behaviours of all the products of an software product line (SPL). This work was complemented by [4], who developed fCTL to formalize temporal logic over FTS^+, enabling rigorous analysis of configurable systems. Further studies refined these ideas, applying fCTL to industrial case studies, but their approach assumes a transition system foundation, which differs structurally from PNPL.

Our idea for verifying dynamic properties of PNPL is done utilising fCTL to accomplish. Despite these advances, the integration of PNPLs with fCTL remains a gap. Some researches noted the need for dynamic property verification in PNPLs, but no systematic transformation to FTS^+ was proposed. Our work addresses this by proposing a transformation from PNPLs to FTS^+, leveraging fCTL for comprehensive verification, and building on the scalability insights from [4] to handle variability-aware state spaces. This fills a critical void identified in [4], which called for unified frameworks to analyse configurable concurrent systems.

This related work provides the necessary context for our proposed methodology, which aims to integrate PNPL modelling with fCTL-based verification, enhancing the analysis of concurrent systems with configurable behaviour.

Capítulo 7

Conclusion and Future Work

This work has successfully addressed the challenge of verifying dynamic properties in Petri Net Product Lines (PNPLs), a formalism that unifies the modelling of concurrent systems with configurable behaviour. By proposing a structured transformation process from PNPLs to feature-enriched Reachability Graphs (fRG) and subsequently to Enhanced Featured Transition Systems (FTS^+), we have bridged the gap between Petri net-based modelling and Feature Computation Tree Logic (fCTL)-based verification. The three-step methodology (constructing feature-enriched RG, converting it to FTS^+, and applying fCTL) has demonstrated its effectiveness in capturing both structural and behavioural variability, as validated through illustrative case studies such as the product line PNPL. Our contributions focused on three major transformation steps:

- From PNPL to feature-enriched RG: We developed algorithms to construct reachability graphs from PNPL models while preserving feature annotations. Three scenarios were considered—single token, multiple tokens with unit weights, and multiple tokens with non-unit weights. We also introduced a pruning strategy to eliminate states with ambiguous feature provenance.
- From RG with Features to FTS^+: We formalized the mapping from reachability graphs to FTS^+ by defining each component of the FTS^+ tuple (states, actions, transitions, labelling, and feature guards). This transformation retains both the behavioural semantics and variability from the PNPL model, ensuring that the resulting FTS^+ model is suitable for fCTL verification.
- fCTL Property Verification: We demonstrated that the resulting FTS^+ models can be used to verify properties such as deadlock freedom and reachability under different feature configurations. This was validated through a case study that illustrated the complete transformation flow and property checking process.

The experimental results highlight the scalability of our approach, leveraging symbolic model checking and pruning techniques to manage the combinatorial state space efficiently. And this enables the verification of critical properties—reachability, deadlock-freedom, and liveness—across all valid feature con-

figurations without enumerating individual variants. By integrating fCTL with FTS^+, we have filled a critical void identified in prior research, offering a unified framework for analysing configurable concurrent systems. This contribution not only enhances the practical adoption of PNPLs in domains like automotive and telecommunications but also enriches the theoretical foundations of software product line engineering.

Although this work establishes a strong foundation for verifying PNPL using fCTL, several avenues remain open for future research:

- Tool Integration and Automation: Develop a complete toolchain that automates the entire transformation pipeline, from PNPL to FTS^+, and integrates with existing model checkers such as NuSMV for fCTL evaluation.
- Scalability Improvements: Explore optimization strategies for dealing with the state explosion problem, such as symbolic representation of markings, partial order reduction, or abstraction techniques.
- Support for More Expressive Features: Extend the feature modelling language to support attributes, constraints, or dynamic features (e.g., optional behaviours based on runtime conditions).
- Empirical Evaluation: Conduct large-scale experiments on industrial PNPL case studies to evaluate the performance, scalability, and expressiveness of the proposed approach in real world settings.
- Extending Property Languages: Investigate the incorporation of other property languages or probabilistic extensions to express a broader class of system properties.
- Runtime Verification: Explore how run-time traces generated from PNPL models can be used in conformance checking or run-time monitoring using the FTS^+ and fCTL framework.

This thesis contributes a novel pathway for formal analysis of PNPL systems and opens up new directions for bridging variability modelling and formal verification in concurrent systems.

Bibliografía

[1] T. Murata, «Petri nets: Properties, analysis and applications», *Proceedings of the IEEE*, vol. 77, n.º 4, págs. 541-580, 1989.

[2] N. P. Lines, «Extensible Structural Analysis of Petri», *Transactions on Petri Nets and Other Models of Concurrency XV*, vol. 12530, pág. 27, 2021.

[3] E. Gómez-Martínez, E. Guerra, J. de Lara y A. Garmendia, «Lifted structural invariant analysis of Petri net product lines», *Journal of Logical and Algebraic Methods in Programming*, vol. 130, pág. 100 824, 2023.

[4] A. Classen, P. Heymans, P.-Y. Schobbens y A. Legay, «Symbolic model checking of software product lines», en *Proceedings of the 33rd International Conference on Software Engineering*, 2011, págs. 321-330.

[5] C. Baier y J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[6] J. L. Peterson, «Petri nets», *ACM Computing Surveys (CSUR)*, vol. 9, n.º 3, págs. 223-252, 1977.

[7] T. Murata, «Petri nets: Properties, analysis and applications», *Proceedings of the IEEE*, vol. 77, n.º 4, págs. 541-580, 2002.

[8] K. Jensen, «Coloured Petri nets: A high level language for system design and analysis», en *International conference on application and theory of Petri nets*, Springer, 1989, págs. 342-416.

[9] C. Ramchandani, «Analysis of asynchronous concurrent systems by timed Petri nets», 1974.

[10] Molloy, «Performance analysis using stochastic Petri nets», *IEEE Transactions on computers*, vol. 100, n.º 9, págs. 913-917, 1982.

[11] M. Ajmone Marsan, G. Conte y G. Balbo, «A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems», *ACM Transactions on Computer Systems (TOCS)*, vol. 2, n.º 2, págs. 93-122, 1984.

[12] R. Fehling, «A concept of hierarchical Petri nets with building blocks», en *Advances in Petri Nets 1993 12*, Springer, 1993, págs. 148-168.

[13] H. J. Genrich, «Predicate/transition nets», en *Petri Nets: Central Models and Their Properties: Advances in Petri Nets 1986, Part I Proceedings of an Advanced Course Bad Honnef, 8.–19. September 1986*, Springer, 1987, págs. 207-247.

[14] Y. K. Lee y S. J. Park, «OPNets: An object-oriented high-level Petri net model for real-time system modeling», *Journal of Systems and Software*, vol. 20, n.º 1, págs. 69-86, 1993.

[15] K. Pohl, G. Böckle y F. Van Der Linden, *Software product line engineering: foundations, principles, and techniques*. Springer, 2005, vol. 1.

[16] ETSIINF. «Recomendaciones sobre el contenido de la Memoria Final». (2017), dirección: `http://www.fi.upm.es/?pagina=1475` (visitado 01-06-2023).

[17] K. Kang, S. Cohen, J. Hess, W. E. Novak, A. S. Peterson et al., «Feature-oriented domain analysis (FODA) feasibility study», 1990.

[18] M. Silva, E. Terue y J. M. Colom, «Linear algebraic and linear programming techniques for the analysis of place/transition net systems», *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets 3*, págs. 309-373, 1998.

[19] R. Zurawski y M. Zhou, «Petri nets and industrial applications: A tutorial», *IEEE Transactions on industrial electronics*, vol. 41, n.º 6, págs. 567-583, 1994.

[20] C. Girault y R. Valk, *Petri nets for systems engineering: a guide to modeling, verification, and applications*. Springer Science & Business Media, 2013.

[21] A. Kostin, *Using transition invariants for reachability analysis of Petri nets*. IntechOpen Rijeka, Croatia, 2008.

[22] P. Clements y L. Northrop, *Software product lines*. Addison-Wesley Boston, 2002, vol. 1.

[23] A. Cimatti, E. Clarke, F. Giunchiglia y M. Roveri, «NuSMV: a new symbolic model checker», *International journal on software tools for technology transfer*, vol. 2, págs. 410-425, 2000.

[24] Akers, «Binary decision diagrams», *IEEE Transactions on computers*, vol. 100, n.º 6, págs. 509-516, 1978.

[25] M. C. Browne, E. M. Clarke y O. Grümberg, «Characterizing finite Kripke structures in propositional temporal logic», *Theoretical computer science*, vol. 59, n.º 1-2, págs. 115-131, 1988.

[26] P. Buchholz y P. Kemper, «Hierarchical reachability graph generation for Petri nets», *Formal Methods in System Design*, vol. 21, págs. 281-315, 2002.

[27] C. A. Petri, «Kommunikation mit automaten», 1962.

[28] S. Apel, D. Batory, C. Kästner y G. Saake, «Feature-oriented software product lines», 2013.

[29] R. Muschevici, D. Clarke y J. Proenca, «Feature petri nets», en *Proceedings of the 14th international software product line conference (SPLC 2010)*, Lancaster University; Lancaster, United Kingdom, vol. 2, 2010, págs. 99-106.

[30] R. Muschevici, J. Proença y D. Clarke, «Feature Nets: behavioural modelling of software product lines», *Software & Systems Modeling*, vol. 15, n.º 4, págs. 1181-1206, 2016.

[31] J. Desel, «Basic linear algebraic techniques for place/transition nets», en *Advanced Course on Petri Nets*, Springer, 1996, págs. 257-308.

[32] Z. Lu, J. Liu, L. Dong y X. Liang, «Maintenance process simulation based maintainability evaluation by using stochastic colored Petri net», *Applied Sciences*, vol. 9, n.º 16, pág. 3262, 2019.

[33] V. Khomenko, «Model checking based on prefixes of Petri net unfoldings», Tesis doct., Newcastle University, 2003.

[34] K. Jensen, «Coloured Petri nets and the invariant-method», *Theoretical computer science*, vol. 14, n.º 3, págs. 317-336, 1981.

[35] E. A. Emerson y E. M. Clarke, «Characterizing correctness properties of parallel programs using fixpoints», en *Automata, Languages and Programming: Seventh Colloquium Noordwijkerhout, the Netherlands July 14–18, 1980 7*, Springer, 1980, págs. 169-181.