GSDLAB TECHNICAL REPORT

The Semantics of Feature Models via Formal Languages (Extended Version)

Aliakbar Safilian, Tom Maibaum, Zinovy Diskin

GSDLAB–TR 2015-01-02 January 2015

Generative Software Development Lab

University of Waterloo

Generative Software Development Laboratory
University of Waterloo
200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1

WWW page: http://gsd.uwaterloo.ca/

The Semantics of Feature Models via Formal Languages

Aliakbar Safilian[1], Tom Maibaum[1], Zinovy Diskin[1,2]

[1] Department of Computing and Software,
McMaster University, Canada
safiliaa | maibaum | zdiskin@mcmaster.ca
[2] Generative Software Development Lab.,
Department of Electrical and Computer Engineering,
University of Waterloo, Canada
zdiskin@gsd.uwaterloo.ca

Abstract. Feature modeling is a common framework for software design. A feature model is a graphical structure presenting a hierarchical decomposition of features, called a feature diagram, with some possible crosscutting constraints between them. Feature modeling languages are grouped into *basic* and *cardinality-based* feature models. Cardinality-based feature models subsume basic ones. In this paper, we provide a reduction process, which allows us to go from a cardinality-based feature diagram to an appropriate regular expression such that the expression faithfully captures the semantics of the feature diagram. As for CCs, we propose a formal language interpretation of them. In this way, we provide a formal language-based semantics for cardinality-based feature models. Accordingly, we describe a computational hierarchy of feature models, which guides us in how feature models can be constructively analyzed. We also characterize some existing analysis operations over feature models in terms of on languages and discuss the corresponding decidability problems.

1 Introduction

Product line engineering [30] is a well-known industrial approach to software design. A *product* is a set of *features*, where "a feature is a system property that is relevant to some stakeholders and is used to capture commonalities or discriminate among systems in a family" [8]. A *product line* (PL) is a set of products that share some common features. The main advantage of this approach to software production is a reduction in cost and development time, since, instead of producing a single product, a set of similar products are produced [19, 30].

Feature modeling is the most common approach for modeling the commonalities and variabilities in a PL. A feature model (FM) is a graphical structure presenting a hierarchical decomposition of features, called a *feature diagram* (FD), with some possible *crosscutting constraints* (CCs) between them. There are many feature modeling languages, which are grouped into *basic* and *cardinality-based*

FMs. We describe the feature modeling languages using a small part of the student awards system at McMaster University.

Fig. 1(a) is a basic FD of the system. It is a tree of features, where the edges exhibit the relationships between features. An edge with a black bullet shows a *mandatory* feature: every application must include a ref (reference), and the hollow-ended one shows an *optional* feature: an application can optionally be equipped with citizen (confirming that the applicant is a citizen). These two types of edges (mandatory and optional) are called *solitary*, while other edges are *grouped* into two groups: OR (the black angle) and XOR (the hollow angle). The XOR group {NSERC, GB, IE} shows that the student can apply for at most one and only one of the awards NSERC (Natural Sciences and Engineering Research Council), GB (Graham Bell Scholarship) and IE (International Excellence Award). The OR group {markA, publication} indicates that to apply for the IE award, the student must have either a markA, or a publication, or both in his record.

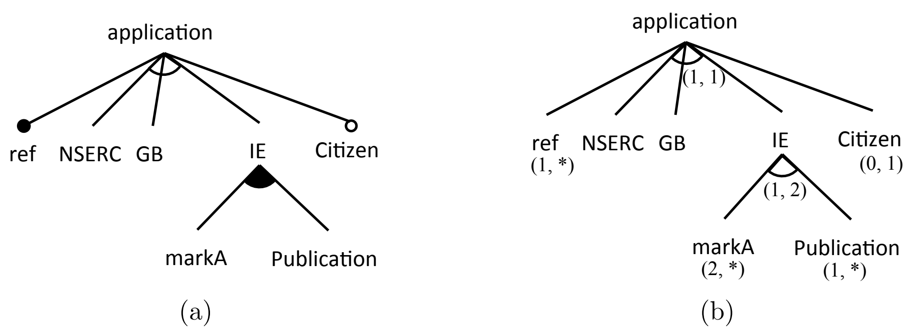


Fig. 1: (a) a basic FD (b) a cardinality-based FD

The set of valid products of a basic FD can be translated into a propositional logic formula generated over the set of features [26]. For our example, the corresponding formula would be the conjunction of the root feature (application) and "ref $\leftrightarrow$ application", "citizen $\rightarrow$ application", "NSERC $\vee$ GB $\vee$ IE $\leftrightarrow$ application", "NSERC $\wedge$ GB $\rightarrow \perp$", "NSERC $\wedge$ IE $\rightarrow \perp$", "IE $\wedge$ GB $\rightarrow \perp$", and "markA $\vee$ publication $\leftrightarrow$ IE". In this sense, any logical formula can be seen as a CC [10]. Let us have cc_1: "citizen $\longrightarrow \neg$IE" and cc_2: "NSERC $\vee$ GB $\longrightarrow$ citizen" as the CCs stating that a "citizen student cannot apply for the IE award" and "one of the requirements for the NSERC and GB awards is to be a citizen", respectively. cc_1 and cc_2 are called an *exclusive* and an *inclusive* CC, respectively. This FM represents the six valid products {application, NSERC, citizen, ref}, {application, GB, citizen, ref}, {application, IE, markA, ref}, {application, markA, publication, ref}, and {application, IE, publication, ref}. The set of all products of a given FM is called the *product line* of the FM and is denoted by $\mathcal{PL}(\mathbb{M})$.

Suppose that we need to specify some requirements regarding the number of feature instances. For example, consider the following requirements: (i) There is no upper bound on the number of instances of the features ref, markA, and publication. (ii) If the student applies for the IE award by providing A-marks, the number of markA in his/her record must be more than two. Clearly, basic FMs like in Fig. 1(a) cannot model such requirements, since they do not manage the number of instances. To address such system requirements, Czarnecki et al. proposed *cardinality-based* FMs (CFMs) [7–9], where UML-like multiplicities, called *cardinalities*, are used in place of traditional edge types. The FD of a CFM (cardinality-based FD, abbreviated to CFD) is a labeled tree of features. There are two types of cardinalities: *feature* and *group* cardinalities. Fig. 1(b) provides a CFD for the awards system including the requirements (i) and (ii). The group cardinalities $(1,1)$ and $(1,2)$ model XOR and OR groups in terms of cardinalities. The feature cardinality $(0,1)$ on citizen models its optional presence in an application. The feature cardinalities $(1,*)$ on ref and publication, $(2,*)$ on markA together satisfy the requirements (i) and (ii). If no cardinality was specified on a node then the cardinality $(1,1)$ is assumed: the cardinalities on features NSERC, GB and IE are $(1,1)$.

CCs in a CFM can refer to feature instances. Take, for example, the constraint: cc_3: "The number of instances of ref must be even". A product of a CFM is a multi-set of features satisfying the constraints. For an example, the multi-set $\{\mathsf{application}, \mathsf{IE}, \mathsf{markA}^3, \mathsf{ref}^4\}$ is a product of this model. Note that the PL of this model is an infinite set. Obviously, CFMs subsume basic FMs [8].

The common understanding of the semantics of an FM in the literature is its PL [34]. This semantics does not capture all essential and practically important information of FMs. This is mainly because an FM provides a hierarchical structure for features, which is forgotten in its PL [15, 35]. For a very simple example, consider two FMs $\mathbb{M}_1$ (a is the root and b is the only mandatory child of a) and $\mathbb{M}_2$ (b is the root and a is the only mandatory child of b). $\mathbb{M}_1$ and $\mathbb{M}_2$ represent the same PL consisting of the only product $\{a, b\}$, but their hierarchical structures are different. Capturing hierarchical structure of FMs is important for several analysis operations over FMs [4], e.g., for finding the least common ancestor (LCA) of a given set of features [27].

In [15], in order to adequately represent the hierarchical structure of basic FMs semantically, we introduced a Kripke semantics for basic FMs, and showed that basic feature modeling is a branch of behavioral modeling, which needs a modal rather than Boolean logic. In the present paper, we invoke formal language (FL) theory to approach building a semantics for cardinality-based feature modeling, which is a more challenging area of feature modeling. This method allows us to approach FM problems by translating them into FL-theory problems that could be managed by well-elaborated FL-theory methods and tools. Indeed, we provide an FL interpretation $\mathcal{L}_\mathbb{M}$ for a given FM $\mathbb{M}$. To consider $\mathcal{L}_\mathbb{M}$ as a faithful semantics for the FM, $\mathcal{L}_\mathbb{M}$ must satisfy the following two fundamental properties:

P-1 "*The multi-set interpretation of* $\mathcal{L}_\mathbb{M}$ *is equal to* $\mathcal{PL}(\mathbb{M})$".

P-2 "*$\mathcal{L}_{\mathbb{M}}$ preserves the hierarchical structure of* $\mathbb{M}$".

The meaning of **P-1** is clear. **P-2** says that the hierarchical structure of $\mathbb{M}$ can be extracted from $\mathcal{L}_{\mathbb{M}}$. This property is formalized in Definition 21. Later we will show that our FL semantics does satisfy these two requirements, see Theorem 2 and Theorem 1.

Industrial FMs may have thousands of features, and their PLs can be complex [26]. Hence, analysis operations on FMs need automated support. Several approaches, such as propositional logic- and constraint programming (CP)-based approaches, have been proposed for automated analysis of basic FMs. In these methods, a given FM is translated into logical formulas or CP and then off-the-shelf tools such as SAT solvers are used for reasoning about the FM. However, these approaches have the following deficinecies: (i) They take into account only the PL of FMs and do not capture their hierarchical structures. Due to this deficiency, some operations, say LCA, cannot be implemented in these methods; (ii) These approaches cannot support CFMs, since such FMs cannot be encoded into propositional logic or CP. Our proposed FL-based framework covers such deficiencies. In this paper, we also show that not all of the proposed analysis operations are decidable when applied to all kinds of FMs.

The plan for this paper is as follows. Sect. 2 provides a background on FL theory and some preliminary definitions. In Sect. 3, we provide a formal syntax for CFDs and a formal definition of their valid products. In Sect. 4, we describe an important generalization of CFDs, called cardinality-based regular expression diagrams (CRDs) in which labelling of nodes can be any regular expressions built over an alphabet. Then we show how to translate CRDs to regular expressions. Also, we prove that the regular expression generated in this way for a given CFD satisfies **P-1** and **P-2**. In Sect. 5, we show how to interperet CCs in FL and then give a definition of CFMs (CFDs + CCs). In Sect. 6, we introduce some analysis operations and investigate their decidability problems. Related work is discussed in Sect. 8. Sect. 9 discusses the conclusions. Finally, we discuss some important open problems/future work in Sect. 10.

Below we present the notations used throughout the paper. Some further notations are introduced where they are used.

Notation.

- $|A|$ denotes the cardinality of a set A.
- The notation $f|_A$ means the restriction of the function f to a subdomain A.
- $\mathbb{N}$ denotes the set of natural numbers.
- For a given set $X = \{x_1, \ldots, x_n\} \subset \mathbb{N}$: $\sum X = x_1 + \ldots + x_n$.

2 Background on Formal Languages

In this section, we provide a concise background on some materials in the formal language theory, which are used in the current paper. Some further concepts/results on the FL theory are introduced where they are used. For more comprehensive background, we refer the interested reader to some standard textbooks such as Linz [25], Davis [11], Kozen [23], Hopcroft [21], and Cooper [6].

Let us, first, fix the alphabet (set of symbols) and denote it by Σ. Σ^* denotes the set of all finite words (sequences of instances of symbols) built over Σ. Any subset of Σ^* is called a *language*. According to their computational properties, the languages are grouped into several kinds. The most well-known are *regular*, *context-free*, *context-sensitive*, *recursive*, and *recursively enumerable* languages. Note that, according to Turing thesis, we consider algorithms and Turing machines equivalent.

Recursively Enumerable Languages. A language $\mathcal{L}$ is called a recursively enumerable language (a.k.a. semi-computable, semi-decidable, computably enumerable) if there exists an algorithm (Turing machine) accepting the language. In other words, there is an algorithm such that it halts (terminates) for any given element (word) in $\mathcal{L}$ and outputs a symbol indicating that the input is in $\mathcal{L}$. Note that there is no guarantee that the algorithm halts for any given words (those that are not in the language).

Recursive Languages. A language $\mathcal{L}$ is called recursive (a.k.a. computable, decidable) if there exists an algorithm such that for any given word the algorithm halts and decides whether it is in the language or not.

Context-Sensitive Languages. A language is called context-sensitive if there exists an algorithm written in a monotonic (equivalently context-sensitive) grammar. A grammar is monotonic if all of whose productions are in the form of $\Gamma \rightarrow \Theta$, where Γ and Θ are strings generated over terminals and non-terminals, such that Θ is not shorter than Γ.

Context-free Languages. A language is called context-free if it can be generated by some context-free grammars. A grammar is context-free if all of its productions are in the form of $V \rightarrow \Theta$, where V is a non-terminal symbol and Θ is a string of terminals and non-terminals. In an equivalent way, we could define context-free languages by using *push-down automata* [1] [25].

Regular Languages. A language is regular if and only if it can be expressed by some *regular expressions*, *regular grammars*, or *finite automata*.

Regular expressions are defined according to the following BNF:

$$Reg ::= \varnothing \mid \varepsilon \mid \sigma \text{ (for any } \sigma \in \Sigma) \mid Reg + Reg \mid Reg.Reg \mid Reg^* \mid (Reg).$$

The expressions $\varnothing$, ε, σ (for any $\sigma \in \Sigma$) are often called primitive regular expressions.

[1] Since we do not use push-down automata in this paper, we do not go into the detail definition of such automata.

Languages associated to regular expressions (Semantics): The language associated to a regular expression Reg is denoted by $\mathcal{L}(Reg)$ and defined in an inductive way as follows:

$\mathcal{L}(\varnothing) = \varnothing$,
$\mathcal{L}(\varepsilon) = \{\varepsilon\}$,
$\mathcal{L}(\sigma) = \{\varepsilon\}$, for any $\sigma \in \Sigma$,
$\mathcal{L}(Reg_1 + Reg_2) = \mathcal{L}(Reg_1) \cup \mathcal{L}(Reg_2)$,
$\mathcal{L}(Reg^*) = (\mathcal{L}(Reg))^*$,
$\mathcal{L}((Reg)) = \mathcal{L}(Reg)$,
$\mathcal{L}(Reg_1.Reg_2) = \mathcal{L}(Reg_1).\mathcal{L}(Reg_2)$.

Finite Automata. A finite automaton is a tuple (S, T, F, I) where S is a finite set of states, $T : S \times (\Sigma \cup \varepsilon) \to 2^S$ is a transition function, $F \subseteq S$ is a set of final states, and $I \subseteq S$ is a set of initial states. The transition relation can be extended to $T^* : S \times \Sigma^* \to 2^S$ to deal with strings rather than a single symbol. $T^*(s, w) = S'$ means that starting the state s and visiting the word w, S' is the set of all possible states that the automaton may be in.

Languages associated to finite automata: Let $Aut = (S, T, F, I)$ be a finite automaton. The language associated to Aut is denoted by $\mathcal{L}(Aut)$ and is equal to $\{w \in \Sigma^* : \exists i \in I, T^*(i, w) \cap F \neq \varnothing\}$.

Transition graphs are used to represent finite automata. Fig. 2 represents an automaton for the language $\{w \in \{a, b\}^* :$ the number of instances of a is even$\}$. The initial state is identified by an incoming unlabelled arrow not originating at any state. The final states are drawn with double circles.

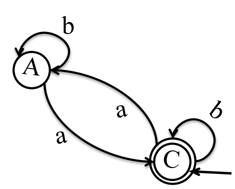


Fig. 2: A transition graph for $\{w \in \{a, b\}^* : \#_w(a)$ is even$\}$

Regular Grammars. A regular grammar is either a *right* or *left regular grammar*. The productions of a right (left, respectively) regular grammar must be in one of the following forms: $V \to \varepsilon$ (the same, respect.), $V \to \sigma$ (the same respect.), $V \to \sigma V'$ ($V \to V'\sigma$, respect.), where σ is a terminal and V, V' are non-terminals.

We also need to know the concept of *bounded regular language*:

Bounded Regular Languages. We say a regular language $\mathcal{L}$ is a bounded regular language, if there are n words $w_1, \ldots, w_n \in \Sigma^*$ such that $\mathcal{L}_1 \subseteq w_1^* \ \ldots \ w_n^*$.

Fig. 3 presents a containment hierarchy of formal languages: Regular $\subset$ Context-free $\subset$ Context-sensitive $\subset$ Recursive $\subset$ Recursively enumerable (r.e.)

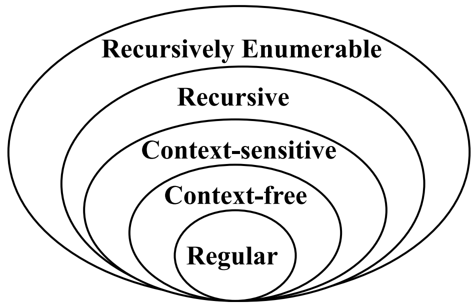


Fig. 3: A containment hierarchy of formal languages

Some Computational Properties.
The following properties of formal languages are used throughout the paper:

Closure: Regular Languages. The class of regular languages is closed under the set operations *union*, *intersection*, *complement*, *relative complement*. It is also closed under the language operations *Kleene star*, *concatenation*, and *reversal*: Let $\mathcal{L}$ be a formal language. Its reverse is denoted by $\mathcal{L}^R$ and defined as follows. $\mathcal{L}^R = \{w^R : w \in \mathcal{L}\}$ (w^R is the reverse sequence of the sequence w).

Context-free Languages. The class of context-free languages is closed under the set operations *union*, but is not closed under intersection, complement and relative complement operations. It is also closed under *Kleene star*, *reversal*, and *concatenation*. This class is also closed under intersection with any regular languages.

Context-sensitive Languages. The class of context-sensitive languages is closed under *intersection*, *union*, *complement*, *relative complement*, and *Kleene star*. However, it is not closed under other operations. This class is also closed under intersection with any regular languages.

Decidability: Note that all recursive languages (including regular, context-free, and context-sensitive languages) are decidable. Below, we state some other decidability results that are used in the paper.

Emptiness Problem. The problem is that for a given language $\mathcal{L}$, "is $\mathcal{L} = \varnothing$ decidable?"
The emptiness problem is decidable in both classes of regular and context-free languages. However, it is not decidable in the class of context-sensitive languages.

Equality Problem. Given two languages $\mathcal{L}, \mathcal{L}'$, the problem is to decide whether the question "$\mathcal{L}= \mathcal{L}$?"' is decidable or not.
The equality problem is decidable in the class of regular languages, but it is not decidable in other classes of formal languages. However, if one of the given languages is a bounded regular and the other is context-free, then the equality problem would be still decidable.

Inclusion Problem. Given two languages $\mathcal{L}, \mathcal{L}'$, the question is to decide whether "$\mathcal{L} \subset \mathcal{L}'$ " is decidable or not?
The inclusion problem is decidable in the class of regular languages, but it is not decidable in the class of other classes of formal languages. However, if $\mathcal{L}$ is context-free and $\mathcal{L}'$ is regular, then the above problem would be still decidable.

In the following, we introduce some notations that are used in the subsequent section.

Notations.

- For any REs *Reg* (languages L, respectively), $\Sigma(Reg)$ ($\Sigma(L)$, respectively) denotes the alphabet which *Reg* (L, respectively) is built on.
- Let $\mathbf{RE}(\Sigma)$ denote the class of all regular expressions built over Σ.
- Let *Gra*, *Reg* and *Aut* be a formal grammar, regular expression and automaton over an alphabet Σ, respectively.
  - Then $\mathcal{L}(Gra), \mathcal{L}(Reg), \mathcal{L}(Aut)$ denote their corresponding languages, respectively.
  - Let Σ' be another alphabet with a bijection $f : \Sigma \to \Sigma'$. Then $Reg[f]$ ($Gra[f]$ and $Aut[f]$, respectively) is a regular expression (grammar, automaton, respectively) built over Σ' using *Reg* (*Gra* and *Aut*, respectively) by substituting any element $\sigma \in \Sigma$ with $f(\sigma)$.
- The multi-set interpretation (Parikh's image) of a word w (a formal language $\mathcal{L}$, respectively) is denoted by w^{bag} ($\mathcal{L}^{\text{bag}}$, respectively).
- U_w denotes the set of elements included in a word (or multi-set) w.
- $\#_w(\sigma)$ denotes the number of instances (occurrences) of σ in a word (or multi-set) w.
- For a given word w, we consider a partial order $\sqsubseteq_w \subseteq U_w \times U_w$ defined as follows: $\forall \sigma, \sigma' \in U_w$, $\sigma \sqsubseteq_w \sigma'$ iff any instance of σ' is preceded by some instances of σ in w.
- For two words w and w', the notation $w \leq_{\text{seq}} w'$ is used to denote that w is a subsequence of w'.
- To make the regular expressions more readable, we use iterations rather than recursion, to express repetition, e.g., to show n repetition of a letter f, we use the notation f^n.

We will also need the following definitions:

Definition 1 (Substitution of a letter with a language). *Let $\mathcal{L}$ and $\mathcal{L}'$ be two languages and $\sigma \in \Sigma(\mathcal{L})$. The* Substitution of σ with $\mathcal{L}'$ *is a language denoted by $\mathcal{L}[\sigma \mapsto_{\text{L}} \mathcal{L}']$ and equal to: $\{w \in \mathcal{L} : \sigma \notin w\} \bigcup \{ww'w'' : (w\sigma w'' \in \mathcal{L}) \wedge (w' \in \mathcal{L}')\}$.* □

Notation. Let $\Sigma' = \{\sigma_1, \ldots, \sigma_k\}$ be a subset of Σ and *sub* be a function, which maps each letter σ_i of Σ' to an FL $\mathcal{L}_i$. We write $\mathcal{L}[\sigma \mapsto_{\text{L}} sub(\sigma) : \forall \sigma \in \Sigma']$ to mean $\mathcal{L}[\sigma_1 \mapsto_{\text{L}} \mathcal{L}_1] \ldots [\sigma_k \mapsto_{\text{L}} \mathcal{L}_k]$.

Definition 2 (Substitution of a letter with an expression). *Let $\mathcal{E}$ and $\mathcal{E}'$ be two regular expressions and $\sigma \in \Sigma(\mathcal{E})$. The* Substitution of σ with $\mathcal{E}'$ *is a regular expression denoted by $\mathcal{E}[\sigma \mapsto_{\mathrm{E}} \mathcal{E}']$ and specified as follows: any instance of σ in $\mathcal{E}$ is replaced by $\mathcal{E}'$.* □

Notation. Let $\Sigma' = \{\sigma_1, \ldots, \sigma_k\}$ be a subset of $\Sigma(\mathcal{E})$ and *sub* be a function, which maps each letter σ_i of Σ' to an RE $\mathcal{E}_i$. We write $\mathcal{E}[\sigma \mapsto_{\mathrm{E}} sub(\sigma) : \forall \sigma \in \Sigma']$ to mean $\mathcal{E}[\sigma_1 \mapsto_{\mathrm{E}} \mathcal{E}_1] \ldots [\sigma_k \mapsto_{\mathrm{E}} \mathcal{E}_k]$.

Definition 3 (Substitution of a symbol with an language). *Let $\mathcal{L}$ and $\mathcal{L}'$ be two languages and $\sigma \in \Sigma(\mathcal{L})$. The* Substitution of σ with $\mathcal{L}'$ *is a language denoted by $\mathcal{L}[\sigma \mapsto_{\mathrm{L}} \mathcal{L}']$ and specified as follows: any instance of σ in $\mathcal{L}$ is replaced by $\mathcal{L}'$.* □

Notation. Let $\Sigma' = \{\sigma_1, \ldots, \sigma_k\}$ be a subset of $\Sigma(\mathcal{E})$ and *sub* be a function, which maps each letter σ_i of Σ' to an RE $\mathcal{E}_i$. We write $\mathcal{E}[\sigma \mapsto_{\mathrm{E}} sub(\sigma) : \forall \sigma \in \Sigma']$ to mean $\mathcal{E}[\sigma_1 \mapsto_{\mathrm{E}} \mathcal{E}_1] \ldots [\sigma_k \mapsto_{\mathrm{E}} \mathcal{E}_k]$.

3 CFDs: Formal Definitions

We use the CFD in Fig. 4 as an example to illustrate the definitions. The feature label of each node is represented in parenthesis next to the node and G denotes the grouped nodes {e, f, g }.

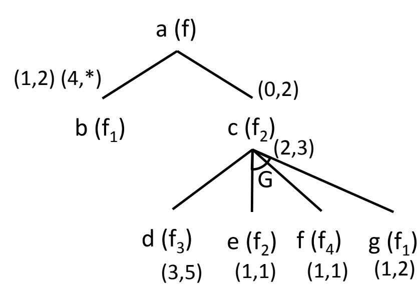


Fig. 4: A CFD

To formalize the syntax of CFDs, we will first need the following notion.

Definition 4 (Cardinalities).

(i) The cardinality-set *is the set $\mathfrak{C} = \{(k, m) \in \mathbb{N} \times (\mathbb{N} \uplus \{*\}) : (k \leq_* m) \wedge (m \neq 0)\}$, where $\leq_*$: $(\mathbb{N} \uplus \{*\}) \times (\mathbb{N} \uplus \{*\})$ is a reflexive transitive relation defined as follows: $\forall k, m \in \mathbb{N}$, $k \leq_* m$ iff $k \leq m$ and $\forall k \in \mathbb{N}$, $k \leq_* *$.*

(ii) An element $c = (k, m) \in \mathfrak{C}$ is called a cardinality*. We call k and m the* lower-bound*, denoted by $low(c)$, and* upper-bound*, denoted by $up(c)$, of c, respectively.*

(iii) A subset $C \subseteq \mathfrak{C}$ is called a cardinality interval *if there exists $I = \{1, \ldots, n\} \subset \mathbb{N}$ such that $C = \{(k_i, m_i) : i \in I\}$ in which $m_i \leq_* k_{i+1}$, for all $i, i+1 \in I$. We call k_1 and m_n the* lower-bound*, denoted by $low(C)$, and* upper-bound*, denoted by $up(C)$, of C, respectively.* □

Consider the CFD in Fig. 4 and ignore the labels on nodes. We call such a tree a *cardinality-based diagram* (CD). Indeed, a CFD is a labeled CD. A CD itself is an unlabelled tree where some subsets of non-root nodes are grouped ($\mathsf{G} = \{\mathsf{e}, \mathsf{f}, \mathsf{g}\}$ in Fig. 4) and other nodes are called solitary (the nodes b, c, and d in Fig. 4). In addition, non-root nodes and groups are equipped with some cardinality intervals (e.g., $\{(1,2),(4,*)\}$ on the node b and $\{(1,2)\}$ on G).

Definition 5 (Cardinality-based Diagrams). *A* cardinality-based diagram *(CD) is a 3-tuple $\mathbf{D} = (T, \mathcal{G}, \mathcal{C})$ consisting of the following components.*

(i) $T = (N, r, \_^{\uparrow})$ is a tree *with set N of nodes, $r \in N$ is the root, and function $\_^{\uparrow}$ maps each non-root node $n \in N_{-r} \stackrel{\text{def}}{=} N \setminus r$ to its parent $n^{\uparrow}$. The inverse function that assigns to each node n the set of its children is denoted by $n_{\downarrow}$. The set of all descendants of n is denoted by $n_{\downarrow\downarrow}$.*

(ii) $\mathcal{G} \subseteq 2^{N_{-r}}$ is a set of grouped *nodes. For all $G \in \mathcal{G}$, $|G| > 1$, and all nodes in G have the same parent, denoted by $G^{\uparrow}$. All groups in $\mathcal{G}$ are disjoint, i.e., $\forall G, G' \in \mathcal{G}.\ (G \neq G') \Rightarrow (G \cap G' = \varnothing)$. The nodes that are not in a group are called* solitary *nodes. Let $\mathcal{S}$ denote the solitary nodes, i.e., $\mathcal{S} = N_{-r} - \bigcup_{G \in \mathcal{G}} G$.*

(iii) $\mathcal{C} \subseteq (N_{-r} \uplus \mathcal{G}) \times \mathfrak{C}$ is a left-total relation called the cardinality relation*. For any element $e \in N_{-r} \uplus \mathcal{G}$, $\mathcal{C}(e)$ is a cardinality interval as defined in Definition 4(iii). In addition, for all $G \in \mathcal{G}$, $up(\mathcal{C}(G)) \leq |G|$.* □

Definition 6 (Cardinality-based Feature Diagrams). *A* cardinality-based feature diagram *(CFD) is a 3-tuple $\mathbf{FD} = (\mathbf{D}, F, l)$ where $\mathbf{D} = (T, \mathcal{G}, \mathcal{C})$ is an CD, as defined in Definition 5, F is a set of features, and function $l : N \to F$ labels each node with a feature.* □

Remark 1. The original definition of CFDs in [8] has two restrictions: (i) the cardinality of a grouped node is always $(1,1)$ and (ii) only one cardinality interval is assigned to a group. However, we generalized CFDs in the above definition without essentially complicating the framework and enabling useful generalizations in feature modeling.

Notation. Let $\mathbf{D} = (T, \mathcal{G}, \mathcal{C})$ be a CD with $T = (N, r, \_^{\uparrow})$ and $n \in N$:

– $depth(\mathbf{D})$ denotes the T's depth and $depth(n)$ denotes the n's depth in T.

– $N^{k-} = \{n \in N : depth(n) = k\}$, i.e., the nodes with depth k.

Let $\mathbf{FD}$ be a CFD with $\mathbf{D}$ as its underlying CD:

– By $depth(\mathbf{FD})$, we mean $depth(\mathbf{D})$.

Now we want to formally define a valid product of a given CFD. First, we give a definition of a valid product of its underlying CD. Note that a CD can be seen as a CFD in which the labelling is an inclusion from nodes to nodes. We call a valid product of the CD a *bare product* of the CFD. To obtain the valid products of the CFD, we just need to apply the labelling function on the bare products. A bare product is a multi-set of nodes satisfying the following membership and arity requirements.

(*membership requirements*): The root is included. If a non-root node is included then its parent must also be included, e.g., the presence of the node d in Fig. 4 implies the presence of the node c. If the parent of a mandatory node (a solitary node with lower bound cardinality greater than 0) is included then it must be included too, e.g., the presence of the node c implies the presence of the node d. If a parent of a grouped set of nodes is included then the presence of the grouped nodes must satisfy the associated group cardinalities, e.g., the presence of the node c implies the presence of two or three of the nodes e, f, and g.

(*arity requirements*): The arity of the root node is always 1. The number of instances of a non-root node is verified by the cardinality interval associated with it and the number of instances of its parent node, e.g., if the number of instances of the node c in Fig. 4 is two then the number of instances of the node d must be at least six and at most ten. In general, for non-root nodes n included in the bare product, there must be a cardinality c associated with n such that its arity is less (greater, respectively) than the multiplication of its parent's ($n^{\uparrow}$) arity and c's upper bound (lower bound, respectively).

Definition 7 (Product). *Let* $\mathbf{FD} = (\mathbf{D}, F, l)$ *be a CFD with* $\mathbf{D} = (T, \mathcal{G}, \mathcal{C})$ *and* $T = (N, r, \_^{\uparrow})$.
Bare Product: *A multi-set* BP *over the set of nodes* N *is called a* bare product *if:*
(membership):
(i) $r \in BP$,
(ii) $\forall n \in N_{-r} : n \in BP \Rightarrow n^{\uparrow} \in BP$,
(iii) $\forall n \in BP, \forall n' \in \mathcal{S} : [(n'^{\uparrow} = n) \wedge (low(\mathcal{C}(n')) > 0)] \Rightarrow (n' \in BP)$,
(iv) $\forall n \in BP, \forall G \in \mathcal{G} : (G^{\uparrow} = n) \Rightarrow [\exists c \in \mathcal{C}(G) : low(c) \leq |BP \cap G| \leq up(c)]$,
(arities):
(v) $\#_{BP}(r) = 1$,
(vi) $\forall n \in N_{-r}, \exists c \in \mathcal{C}(n) : \big(\#_{BP}(n^{\uparrow}) \times low(c)\big) \leq \#_{BP}(n) \leq \big(\#_{BP}(n^{\uparrow}) \times up(c)\big)$

Product: *A multi-set* P *over* F *is called a* product *if there exists a bare product* BP *of* $\mathbf{FD}$ *such that* P *is the result of applying the labelling function* l *on the elements of* BP*, i.e., for all features* $f \in F$,
(i) $(f \in P) \Leftrightarrow (l^{-1}(f) \cap BP \neq \varnothing)$,
(ii) $\#_P(f) = +_{n \in l^{-1}(f)} \#_{BP}(n)$
The product family of $\mathbf{FD}$ *is denoted by* $\mathcal{PL}(\mathbf{FD})$. □

The definitions stated in the rest of this section come in handy in the subsequent sections mainly in Sect. 4.2. In the following definitions, we work on just CDs. They can be easily transformed on CFDs using by labelling functions.

Definition 8 (Substitution of a leaf node with a CD). *Consider two CDs* $\mathbf{D} = (N, r, \_^{\uparrow}, \mathcal{G}, \mathcal{C})$ *and* $\mathbf{D}' = (N', n, \_^{\uparrow'}, \mathcal{G}', \mathcal{C}')$ *such that* $N \cap N' = \{n\}$ *and*

$n_{\downarrow} = \varnothing$ *(n is a leaf node in* $\mathbf{D}$*). The* substitution of n with $\mathbf{D}'$ *is a CD, denoted by* $\mathbf{D}[n \mapsto_{\mathrm{D}} \mathbf{D}']$*, equal to* $(N \cup N',\ r,\ \_^{\uparrow} \cup \_^{\uparrow'},\ \mathcal{G} \cup \mathcal{G}',\ \mathcal{C} \cup \mathcal{C}')$. □

As an example, consider the CD in Fig. 5(a) ($\mathbf{D}$). The substitution of the leaf node b in Fig. 4 with $\mathbf{D}$ is shown in Fig. 4(b).

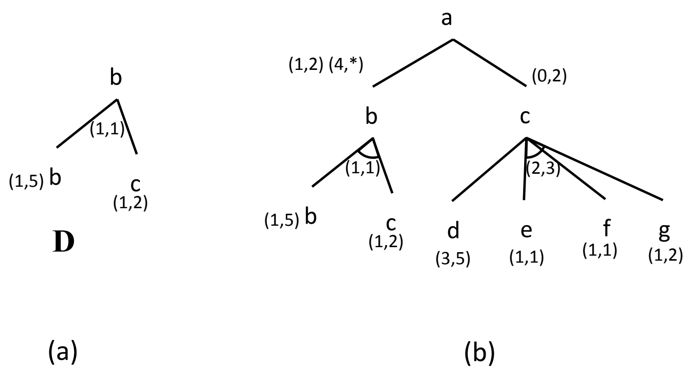


Fig. 5: (a) An CD $\mathbf{D}$, (b) Substitution of (b) in Fig. 4 with (a)

Notation. Let $\mathbf{D}$ be a CD and $N' = \{n_1, \ldots, n_k\}$ be a subset of its set of leaf nodes. Also, let *sub* be a function, which maps each element n_i of N' to a CD $\mathbf{D}_i$ such that for two distinct indices i, j, the set of nodes of $\mathbf{D}_i$ and $\mathbf{D}_j$ are disjoint. For succinctness, we usually write $\mathbf{D}[n \mapsto_{\mathrm{D}} sub(n) : \forall n \in N']$ to mean $\mathbf{D}[n_1 \mapsto_{\mathrm{D}} \mathbf{D}_1] \ldots [n_k \mapsto_{\mathrm{D}} \mathbf{D}_k]$.

The above definition motivates us to define substitution of a feature in a PL with another PL.

Definition 9 (Substitution of a feature with an PL). *Let* $\mathcal{PL}$ *and* $\mathcal{PL}'$ *be two PLs over the sets of features* F *and* F'*, respectively. For a given* $f \in F$*, the* substitution of f with $\mathcal{PL}'$ *is an PL, denoted by* $\mathcal{PL}[f \mapsto_{\mathrm{P}} \mathcal{PL}']$*, specified as follows: each instance of* f *in a product of* $\mathcal{PL}$ *is substituted by a product of* $\mathcal{PL}'$*.* □

As an example, let $F = \{f_1, f_2, f_3\}$, $F' = \{f_1', f_2'\}$, $\mathcal{PL} = \{\{{f_1}^2, f_2\}, \{f_1, {f_2}^3\}, \}$, and $\mathcal{PL} = \{\{{f_1'}^3\}\}$. Then, $\mathcal{PL}[f_1 \mapsto_{\mathrm{P}} \mathcal{PL}'] = \{\{{f_1'}^6, f_2\}, \{{f_1'}^3, {f_2}^3\}, \}$.

Notation. Let $F'' = \{f_1, \ldots, f_k\} \subseteq F$ and *sub* be a function, which maps each feature f_i of F'' to a PL $\mathcal{PL}_i$. We usually write $\mathcal{PL}[f \mapsto_{\mathrm{P}} sub(f) : \forall f \in F'']$ to mean $\mathcal{PL}[f_1 \mapsto_{\mathrm{P}} \mathcal{PL}_1] \ldots [f_k \mapsto_{\mathrm{P}} \mathcal{PL}_k]$.

The reverse operation of the substitution of a leaf in a CD (Definition 8) is defined as follows.

Definition 10 (Cutting of an CD by a node). *Let* $\mathbf{D} = (N, r, \_^{\uparrow}, \mathcal{G}, \mathcal{C})$ *be a CD and* $n \in N$*. The cutting CD of* $\mathbf{D}$ *by the node* n *is the CD* $\mathbf{D}^{-\downarrow n} = (N', r, \_^{\uparrow}|_{N'}, \mathcal{G}', \mathcal{C}|_{\mathcal{G}' \uplus N'})$*, where* $N' = N - n_{\downarrow\downarrow}$ *and* $\mathcal{G}' = \mathcal{G} \cap 2^{N'}$*, i.e., its tree* *is the tree of* $\mathbf{D}$ *except for the tree under* n*; all other components are inherited from* $\mathbf{D}$*.* □

As an example, the Fig. 6 depicts the cutting of the CFD in Fig. 4 by the node c.

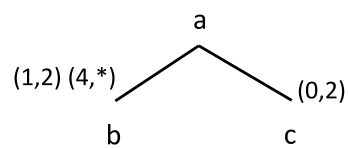


Fig. 6: Cutting of Fig. 4 by c

Definition 11 (Induced Diagram by Node). *Let* $\mathbf{D} = (N, r, \_^{\uparrow}, \mathcal{G}, \mathcal{C})$ *be a CD and* $n \in N$*. The* induced CD *by* n *is a CD* $\mathbf{D}_{\downarrow n} = (N', n, \_^{\uparrow}|_{N'}, \mathcal{G}', \mathcal{C}')$*, where* $N' = \{n' \in N : (n' = n) \vee (n' \in n_{\downarrow\downarrow})\}$*,* $\mathcal{G}' = \mathcal{G} \cap 2^{N'}$*, and* $\mathcal{C}' = \mathcal{C}|_{N' \uplus \mathcal{G}'}$*, i.e., its tree is the tree under* n *in* $\mathbf{D}$*'s tree and all other components are inherited from* $\mathbf{D}$*.* □

The CD in Fig. 7 represents the induced diagram of the node c of the CD in Fig. 4.

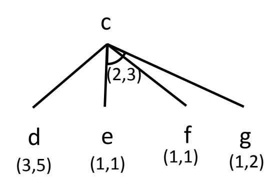


Fig. 7: Induced diagram by c of Fig. 4

Definition 12 (Upper Induced Diagram by depth). *Let* $\mathbf{D} = (N, r, \_^{\uparrow}, \mathcal{G}, \mathcal{C})$ *be a CD and* $0 \leq k \leq depth(\mathbf{D})$*. The* upper induced CD *by* k *is a CD* $\mathbf{D}^{\uparrow k} = (N', r, \_^{\uparrow}|_{N'}, \mathcal{G}', \mathcal{C}')$*, where* $N' = \{n \in N : depth(n) \geq k\}$*,* $\mathcal{G}' = \mathcal{G} \cap 2^{N'}$*, and* $\mathcal{C}' = \mathcal{C}|_{N' \uplus \mathcal{G}'}$*, i.e., its tree is a subtree of* $\mathbf{D}$*'s tree where the nodes are in depth less than or equal to* k*; all other components are inherited from* $\mathbf{D}$*.* □

The CD in Fig. 8 represents the upper induced diagram by the depth 2 in the CD in Fig. 5(b).

4 CFDs to Regular Expressions

In this section, we first define a generalization of CFDs called *Cardinality-based Regular-expression Diagrams* (CRDs). Subsequently, we give a procedure to translate a given CRD to a regular expression (RE). This provides a semantics for CRDs by using regular languages as the semantic domain. We also prove that the REs generated for a given CFD and its underlying CD satisfy the properties **P-1** and **P-2**, respectively.

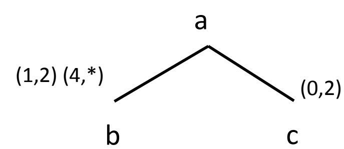


Fig. 8: Induced diagram by depth 2 of Fig. 5

4.1 Cardinality-based Regular-expression Diagrams

Definition 13 (Cardinality-based Regular-expression Diagrams). *A* cardinality-based regular-expression diagram *(CRD) over an alphabet Σ is a 3-tuple* $\mathbf{RD} = (LT_{re}, \mathcal{G}, \mathcal{C})$ *of the following components:*

(i) $LT_{re} = (N, r, \_^{\uparrow}, \Sigma, l_{re})$ *is a* labeled tree *where N, r, $\_^{\uparrow}$, are as defined in Definition 5(i), Σ is a finite set (the alphabet), and $l_{re} : N \to \boldsymbol{RE}(\Sigma)$ is a function that labels each node with a regular expression built over Σ.*

(ii) $\mathcal{G} \subseteq 2^{N_{-r}}$ *is a set of* grouped *nodes, as defined in Definition 5(ii).*

(iii) $\mathcal{C} \subseteq (N_{-r} \uplus \mathcal{G}) \times \mathfrak{C}$ *is called the* cardinality relation*, as defined in Definition 5(iii).*

The class of all CRDs over the same alphabet Σ will be denoted by $\mathcal{RD}(\Sigma)$. □

Remark 2. CRDs subsume CFDs and CDs: A CFD is a CRD in which Σ is the set of features and labels are primitive non-empty REs. A CD is also a CRD in which Σ is equal to the set of nodes and labelling is an inclusion function.

Notation. Given a CRD $\mathbf{RD}$, we will need the following notations in the sequel:

- $depth(\mathbf{RD})$ denotes the depth its underlying CD.
- $lev(\mathbf{RD})$ denotes the set of leaf nodes, i.e., $lev(\mathbf{RD}) = \{n \in N : n_{\downarrow} = \varnothing\}$.
- $glev(\mathbf{RD})$ denotes the set of the grouped leaves, i.e., $glev(\mathbf{RD}) = \{G \in \mathcal{G} : \forall n \in G.\ n_{\downarrow} = \varnothing\}$
- $plev(\mathbf{RD})$ denotes the set of non-leaf nodes all of whose children are leaves, i.e., $plev(\mathbf{RD}) = \{n \in N : n_{\downarrow} \subseteq lev(\mathbf{RD})\}$.
- $cplev(\mathbf{RD})$ denotes the nodes all of whose parents belong to $plev(\mathbf{RD})$, i.e., $cplev(\mathbf{RD}) = \{c \in n_{\downarrow} : n \in plev(\mathbf{RD})\}$.

4.2 CRDs to REs

The translation of a CRD to an RE is a bottom-up procedure and includes a finite number of steps (equal to the depth of the CRD's tree) called *shrinking steps*. Each shrinking step takes a CRD and returns another CRD such that the depth of the output's tree is less than that of the input. The output of the last step is a CRD with the singleton tree (a tree consisting of a single isolated node) whose root is labeled with an RE.

A shrinking step includes three stages: (1) *Eliminating cardinalities from leaves*, (2) *Eliminating grouped leaves*, and (3) *Depth reduction*. We will use the CFD in Fig. 4 as a running example to illustrate the translation procedure.

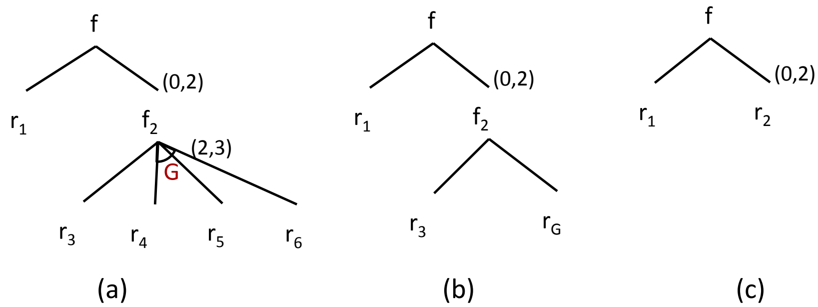


Fig. 9: RCD to RE: Shrinking Procedure on Fig. 4.

Stage 1: Eliminating cardinalities from leaves. At this stage, the REs corresponding to leaf nodes are computed and their cardinalities changed to $(1,1)$. For an example, the RE corresponding to the node b (Fig. 4) would be $f_1 + f_1^2 + f_1^4 f_1^*$. This RE represents the cardinality constraint on this node properly, as it says that the number of instances of the feature f_1 on this node must be one or two or more than three. Then, the label of the leaves are replaced by their REs, computed in the above way, and their associated cardinalities change to $(1,1)$. Fig. 9(a) represents the result of this stage applied to the CFD in Fig. 4 where $r_1 = f_1 + f_1^2 + f_1^4 f_1^*$, $r_3 = f_3^3 + f_3^4 + f_3^5$, $r_4 = f_2$, $r_5 = f_4$, and $r_6 = f_1 + f_1^2$.

Definition 14. *Given a CRD* $\mathbf{RD} = (LT_{re}, \mathcal{G}, \mathcal{C})$ *with* $LT_{re} = (N, r, \_^{\uparrow}, \Sigma, l_{re})$, $lex_{\mathbf{RD}} : lev(\mathbf{RD}) \to \boldsymbol{RE}(\Sigma)$ *is a total function which maps a leaf node in* $\mathbf{RD}$ *to an RE built over* Σ*. For a given node* $n \in lev(\mathbf{RD})$ *with* $\mathcal{C}(n) = \{(k_i, m_i)\}_{1 \le i \le j}$ *(for some* $j \in \mathbb{N}$*),* $lex_{\mathbf{RD}}(n) = r_1 + \ldots + r_j$*, where*

$$r_i = \begin{cases} l_{re}(n)^{k_i} + \ldots + l_{re}(n)^{m_i} & \text{if } m_i \neq * \\ l_{re}(n)^{k_i} \big(l_{re}(n)\big)^* & o.w. \end{cases}$$

□

The stage "eliminating cardinalities from leaves" is formalized by a function $cel : \mathcal{RD}(\Sigma) \to \mathcal{RD}(\Sigma)$, as defined in the following.

Definition 15 (Eliminating cardinalities from leaves Stage). *The function* $cel : \mathcal{RD}(\Sigma) \to \mathcal{RD}(\Sigma)$ *is called the* cardinality eliminator function *and for a given CRD* $\mathbf{RD} = (LT_{re}, \mathcal{G}, \mathcal{C})$ *with* $LT_{re} = (N, r, \_^{\uparrow}, \Sigma, l_{re})$*,* $cel(\mathbf{RD}) = (LT'_{re}, \mathcal{G}, \mathcal{C}')$ *where* $LT'_{re} = (N, r, \_^{\uparrow}, \Sigma, l'_{re})$ *and*

$$\mathcal{C}'(e) = \begin{cases} \{(1,1)\} & \text{if } e \in lev(\mathbf{RD}) \\ \mathcal{C}(e) & o.w. \end{cases}$$

$$l'_{re}(n) = \begin{cases} lex_{\mathbf{RD}}(n) & \text{if } n \in lev(\mathbf{RD}) \\ l_{re}(n) & o.w. \end{cases}$$

□

Stage 2: Eliminating the grouped leaves. At this stage, grouped leaf nodes are replaced by new nodes with proper REs. The input of this stage is the output of the first stage. For an example, consider the grouped leaves G in Fig. 9(a). The group cardinality $(2, 3)$ says that at least two and at most three of the nodes involved in the group (i.e., the nodes e, f, and g) must be included in a valid product for each instance of their parent (i.e., the node c) in the product. The following REs r'_G and r''_G represent the lower and upper bounds of the cardinality, respectively: $r'_G = r_4r_5 + r_5r_4 + r_5r_6 + r_6r_5 + r_4r_6 + r_6r_4$, $r''_G = r_4r_5r_6 + r_4r_6r_5 + r_5r_4r_6 + r_5r_6r_4 + r_6r_4r_5 + r_6r_5r_4$. Thus, the RE corresponding to the group would be $r_G = r'_G + r''_G$. Then, each grouped leaf is replaced by a new node with a cardinality $(1, 1)$ and is labeled with the computed RE. Fig. 9(b) represents the result of applying this stage to Fig. 9(a).

Notation. A concatenation permutation x of a finite set X with $|X| = n$ is a sequence $x_1 \ldots x_n$ such that $\bigcup_{1 \le i \le n}\{x_i\} = X$. Let $Per^k_m(X)$ denote the set of all concatenation permutations x with length between k and m ($k \le |x| \le m$) of X. For an example, $Per^1_2(\{r_1, r_2, r_3\})$ would be the following set of expressions: $\{r_1,\ r_2,\ r_3\} \cup \{r_1r_2,\ r_2r_1,\ r_1r_3,\ r_2r_3,\ r_3r_2\}$.

Definition 16. *Given a CRD* $\mathbf{RD} = (LT_{re}, \mathcal{G}, \mathcal{C})$ *with* $LT_{re} = (N, r, \_^{\uparrow}, \Sigma, l_{re})$, $gex_{\mathbf{RD}} : glev(\mathbf{RD}) \to \boldsymbol{RE}(\Sigma)$ *is a total function. For a given group* $G \in glev(\mathbf{RD})$ *with* $\mathcal{C}(G) = \{(k_i, m_i)\}_{1 \le i \le j}$ *(for some* $j \in \mathbb{N}$*),* $gex_{\mathbf{RD}}(G) = r_1 + \ldots + r_j$ *where for all* $1 \le i \le j$*:* $r_i = \boldsymbol{+}\ X_i$*, and* $X_i = Per^{k_i}_{m_i}(E)$ *with* $E = \{l_{re}(n) : n \in G\}$. □

The stage "eliminating cardinalities from leaves" is formalized by a function $gle : \mathcal{RD}(\Sigma) \to \mathcal{RD}(\Sigma)$, as defined in Definition 17.

Definition 17 (Eliminating grouped leaves Stage). *The function* $gle : \mathcal{RD}(\ \Sigma) \to \mathcal{RD}(\Sigma)$ *is called the* grouped leaves eliminator*. For a given CRD* $\mathbf{RD} = (LT_{re}, \mathcal{G}, \mathcal{C})$ *with* $LT_{re} = (N, r, \_^{\uparrow}, \Sigma, l_{re})$*,* $gle(\mathbf{RD})$ *is defined as follows:*

For each group node $G \in glev(\mathbf{RD})$*, a node identifier* n_G *is assigned. Let* N_G *denote the set of these node identifiers. In other words, we have a bijection* $gid : N_G \to glev(\mathbf{RD})$ *which assigns each grouped node in* $glev(\mathbf{RD})$ *to a unique node identifier in* N_G*. Then,* $gle(\mathbf{RD}) = (LT'_{re}, \mathcal{G}', \mathcal{C}')$ *with* $LT'_{re} = (N', r, \_^{\uparrow'}, \Sigma, l'_{re})$*, where* $N' = (N - glev(\mathbf{RD})) \uplus N_G$, $\mathcal{G}' = \mathcal{G} - glev(\mathbf{RD})$*, and*

$$\mathcal{C}'(e) = \begin{cases} \{(1,1)\} & \textit{if } e \in N_G \\ \mathcal{C}(e) & \textit{o.w.} \end{cases}$$

$$n^{\uparrow'} = \begin{cases} gid(n)^{\uparrow} & \textit{if } n \in N_G \\ n^{\uparrow} & \textit{o.w.} \end{cases}$$

$$l'_{re}(n) = \begin{cases} gex_{\mathbf{RD}}(gid(n)), & \textit{if } n \in N_G \\ l_{re}(n) & \textit{o.w.} \end{cases}$$

□

Stage 3: Depth Reduction. This stage takes the output of the second stage and returns a CRD whose depth is less than that of the input. To this end, the REs corresponding to the nodes all of whose child nodes are leaves are computed. Then, the label of such nodes are replaced by the corresponding computed RE and their child nodes are eliminated from the given CRD. Let us see what the result of this stage applied to the CRD in Fig. 9(b) would be. There is only one node, labeled by f_2, all of whose child nodes are leaf nodes. Fig. 9(c) shows the result, where $r_2 = f_2(r_3 r_G + r_G r_3)$.

Definition 18. *Given a CRD* $\mathbf{RD} = (LT_{re}, \mathcal{G}, \mathcal{C})$ *with* $LT_{re} = (N, r, \_^{\uparrow}, \Sigma, l_{re})$, $pex_{\mathbf{RD}} : plev(\mathbf{RD}) \to \boldsymbol{RE}(\Sigma)$ *is a total function. For a given node* $n \in plev(\mathbf{RD})$, $pex_{\mathbf{RD}}(n) = l_{re}(n)(+\ X)$, *where* $X = Per^j_j(E)$ *and* $j = |n_{\downarrow}|$, *and* $E = \{l_{re}(n') : n' \in n_{\downarrow}\}$. □

The stage "eliminating cardinalities from leaves" is formalized by a function $dre : \mathcal{RD}(\Sigma) \to \mathcal{RD}(\Sigma)$, as defined in the following:

Definition 19 (Depth Reduction Stage). *The function* $dre : \mathcal{RD}(\Sigma) \to \mathcal{RD}(\Sigma)$ *is called the* depth reducer function. *For a given CRD* $\mathbf{RD} = (LT_{re}, \mathcal{G}, \mathcal{C})$ *with* $LT_{re} = (N, r, \_^{\uparrow}, \Sigma, l_{re})$, $dre(\mathbf{RD})$ *is a CRD* $\mathbf{RD}' = (LT'_{re}, \mathcal{G}, \mathcal{C}')$ *with* $LT'_{re} = (N', r, \_^{\uparrow'}, \Sigma, l'_{re})$ *where* $N' = N - cplev(\mathbf{RD})$, $\_^{\uparrow'} = \_^{\uparrow}|_{N'}$, $\mathcal{C}' = \mathcal{C}|_{N' \cup \mathcal{G}}$, *and*

$$l'_{re}(n) = \begin{cases} pex_{\mathbf{RD}}(n) & \text{if } n \in plev(\mathbf{RD}) \\ l_{re}(n) & o.w. \end{cases}$$

□

Hence, a shrinking step is the composition of the functions defined for the above stages.

Definition 20 (Shrinking Step). *The function* $shr : \mathcal{RD}(\Sigma) \to \mathcal{RD}(\Sigma)$ *is called the* shrinking function *and is defined as* $shr = dre \circ gel \circ cel$. *(∘ denotes composition.)* □

We keep doing the shrinking steps until we get a CRD which is a singleton tree. In the running example, we need to do the shrinking step once more. The final result would be the expression $r = f(r_1 r'_2 + r'_2 r_1)$ where $r'_2 = \varepsilon + r_2 + r_2^2$. The notation $\mathcal{E}_{\mathbf{RD}}$ is used to denote the regular expression generated for a given CRD $\mathbf{RD}$. The following proposition follows obviously.

Proposition 1. *Let* $\mathbf{FD} = (\mathbf{D}, F, l)$ *be a CFD with* $\mathbf{D} = (T, \mathcal{G}, \mathcal{C})$ *and* $T = (N, r, \_^{\uparrow})$. *Then,* $\mathcal{E}_{\mathbf{FD}} = \mathcal{E}_{\mathbf{D}}[l]$. □

4.3 The Main Properties of Generated Expressions

In this section, we show that the regular expression interpretation of a given CFD **FD** with **D** as its underlying CD satisfies the properties **P-1** and **P-2**. Note that two different nodes in **FD** can be labeled with the same feature. Thus, to prove the property **P-2** (formalized in Definition 21) of the generative language, we need to work on **D**, i.e., we prove that $\mathcal{L}(\mathcal{E}_{\mathbf{D}})$ satisfies **P-2**. The satisfaction of the properties of **P-1** and **P-2** are shown in Theorems 2 and 1, respectively.

Definition 21 (Formalizing P-2). *Consider a CD* $\mathbf{D} = (T, \mathcal{G}, \mathcal{C})$ *with* $T = (N, r, \_^{\uparrow})$ *and let* $\mathcal{L}$ *be a language built over* N. *We say* $\mathcal{L}$ *preserves the* hierarchical structure *of* $\mathbf{D}$ *(or simply satisfies* **P-2** *for* $\mathbf{D}$*) if* $\forall n, n' \in N : (n' \in n_{\downarrow\downarrow}) \iff \big(\forall w \in \mathcal{L}(\mathcal{E}_{\mathbf{D}}) : (n' \in w) \Rightarrow (n \sqsubseteq_w n')\big)$. □

Theorem 1 (Satisfying P-2). *For a given CD* $\mathbf{D}$, $\mathcal{L}(\mathcal{E}_{\mathbf{D}})$ *satisfies* **P-2** *for* $\mathbf{D}$. □

Proof. We need to prove the following statements:

(1) $(n' \in n_{\downarrow}) \Rightarrow \big(\forall w \in \mathcal{L}(\mathcal{E}_{\mathbf{D}}) : (n' \in w) \Rightarrow (n \sqsubseteq_w n')\big)$

(2) $\big(\forall w \in \mathcal{L}(\mathcal{E}_{\mathbf{D}}) : (n' \in w) \Rightarrow (n \sqsubseteq_w n')\big) \Rightarrow (n' \in n_{\downarrow\downarrow})$

Note that (1) implies $(n' \in n_{\downarrow\downarrow}) \Rightarrow \big(\forall w \in \mathcal{L}(\mathcal{E}_{\mathbf{D}}) : (n' \in w) \Rightarrow (n \sqsubseteq_w n')\big)$.

Proof of (1):

Since $\mathbf{D}$ is an unlabelled tree, for any $i \leq \mathit{depth}(\mathbf{D})$, $\mathit{shr}^i(\mathbf{D})$ is a CRD where the labels of two different nodes would be two different REs built over two disjoint alphabets. Let us call such CRDs *disjoint labeled CRDs* (DL-CRD). It is obvious that for any DL-CRD $\mathbf{RD}$ and $i \leq \mathit{depth}(\mathbf{RD})$, $\mathit{shr}^i(\mathbf{RD})$ is also a DL-CRD. To prove (1), we need to prove a more general statement stated as follows:

General Version of (1):
"Consider a DL-CRD $\mathbf{RD} = (LT_{re}, \mathcal{G}, \mathcal{C})$ with $LT_{re} = (N, r, \_^{\uparrow}, \Sigma, l_{re})$. Let $n', n'' \in N$ with $l_{re}(n') = R'$ and $l_{re}(n'') = R''$ such that $n'' \in n'^{\uparrow}$. Then,
$\forall w \in \mathcal{L}(\mathcal{E}_{\mathbf{RD}}), \forall w'' \in \mathcal{L}(R'') : (w'' \leq_{\text{seq}} w) \Rightarrow [\exists w' \in \mathcal{L}(R') : w'.w'' \leq_{\text{seq}} w]$."

Let $w \in \mathcal{L}(\mathcal{E}_{\mathbf{RD}})$ and $w'' \in \mathcal{L}(R'')$ such that $w'' \leq_{\text{seq}} w$. We need to show that $\exists w' \in \mathcal{L}(R') : w'.w'' \leq_{\text{seq}} w$.

Since $\mathbf{RD}$ and $\mathit{shr}^i(\mathbf{RD})$, for any $i \leq \mathit{depth}(\mathbf{RD})$, are DL-CRDs, $\mathcal{E}_{\mathbf{RD}} = R.(R'.(R''.R^{(3)}+R^{(4)})+R^{(5)})$ for some REs $R, R^{(3)}, R^{(4)}, R^{(5)}$ (note the function *dre* in Definition 19 and Definition 18) such that the REs $R, R', R', R'', R^{(3)}, R^{(4)}, R^{(5)}$ are built over disjoint alphabets. Since $w'' \leq_{\text{seq}} w$, $w \in \mathcal{L}(R.R'.R''.R^{(3)})$. The statement is proven, since R' precedes R'' in $R.R'.R''.R^{(3)}$, i.e., $\exists w' \in \mathcal{L}(R') : w'.w'' \leq_{\text{seq}} w$.

Proof of (2):

We show that the statement $\neg(n' \in n_{\downarrow\downarrow}) \Rightarrow \neg\big(\forall w \in \mathcal{L}(\mathcal{E}_{\mathbf{D}}) : (n' \in w) \Rightarrow (n \sqsubseteq_w n')\big)$ holds, which is equivalent to (2).

Suppose $n' \notin n_{\downarrow\downarrow}$. Let k be the minimum of the depths of the nodes n and n'. Let $\mathit{shr}^{d-k}(\mathbf{D}) = \mathbf{RD}'$. There are two leaves ℓ and ℓ' in $\mathbf{RD}'$ with labels R and R' in $\mathbf{RD}'$ such that $n \in \Sigma(R)$ and $n' \in \Sigma(R')$. Since $\mathbf{RD}'$ is an DL-CRD, $\Sigma(R') \cap \Sigma(R) = \varnothing$. Note that the nodes ℓ and ℓ' would have the same parent, i.e., they are siblings. Let $p = \ell^{\uparrow} = \ell'^{\uparrow}$. There exist the following choices for ℓ and ℓ':

(i) Both are solitary nodes.

(ii) One of them, say ℓ, is in a group and another one, ℓ', is a solitary node.

(iii) Both are in a same group G.

(iv) One of them, say ℓ, is in a group G and another, ℓ', is in another group G'.
Applying the function $gel \circ cel$ (the first and second stages of $shr(\mathbf{RD}') = shr^{d-k+1}(\mathbf{D})$, respectively), we will get two leaves ℓ_1 and ℓ'_1 with labels R_1 and R'_1 in $\mathbf{RD}'_1 = shr(\mathbf{RD}')$ such that $n \in \Sigma(R_1)$ and $n' \in \Sigma(R'_1)$. Note that $\Sigma(R_1) \cap \Sigma(R'_1) = \varnothing$ and all leaves of $\mathbf{RD}'_1$ are solitary with cardinalities $(1, 1)$.

Now let us apply the function *dre* on $\mathbf{RD}'_1$ to get $shr^{d-k+1}(\mathbf{D})$. Since the function *dre* considers any permutation of the p's child nodes, there is a leaf node ℓ'' in $shr^{d-k+1}(\mathbf{D})$ labeled with an RE in the form $R''_\ell = R^{(2)} + R_1.R'_1.R^{(3)} + R'_1.R_1.R^{(4)}$. Since $\Sigma(R_1) \cap \Sigma(R'_1) = \varnothing$, there are two words $w_1, w_2 \in \mathcal{L}(R''_\ell)$ such that $n \sqsubseteq_{w_1} n'$ and $n' \sqsubseteq_{w_2} n$. Thus, keeping doing the shrinking steps till getting $shr(\mathbf{D})$, there would be a word $w \in \mathcal{L}(\mathcal{E}_\mathbf{D})$ such that $n' \in w$ but $\neg(n \sqsubseteq_w n')$. The statement (2) is proven. □

To prove that the expression generated for a given CFD satisfies the property **P-1**, we will fist need the following propositions and lemmas.

Proposition 2. *Let $\mathbf{D} = (N, r, \_^\uparrow, \mathcal{G}, \mathcal{C})$ be a CD and $n \in N$. Then, the following statements hold:*
(i) $\mathbf{D} = \mathbf{D}^{-\downarrow n}[n \mapsto_\mathrm{D} \mathbf{D}_{\downarrow n}]$.
(ii) $\mathcal{PL}(\mathbf{D}) = \mathcal{PL}(\mathbf{D}^{-\downarrow n})[n \mapsto_\mathrm{P} \mathcal{PL}(\mathbf{D}_{\downarrow n})]$.
(iii) $\mathcal{E}_\mathbf{D} = \mathcal{E}_{\mathbf{D}^{-\downarrow n}}[n \mapsto_\mathrm{E} \mathcal{E}_{\mathbf{D}_{\downarrow n}}]$. □

Proof. Follows obviously! □

Lemma 1. *Let $\mathbf{D} = (N, r, \_^\uparrow, \mathcal{G}, \mathcal{C})$ be a CD and k be a number such that $0 \leq k \leq depth(\mathbf{D})$. Then, $\mathbf{D} = \mathbf{D}^{\uparrow k}[n \mapsto_\mathrm{D} \mathbf{D}_{\downarrow n} : \forall n \in N^{k-}]$.* □

Proof. Let $N^{k-} = \{n_1, \ldots, n_i\}$. We define a set of CDs $\{\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_i\}$ recursively as follows: $\mathbf{D}_0 = \mathbf{D}$ and for any $1 \leq j \leq i$: $\mathbf{D}_j = {\mathbf{D}_{j-1}}^{-\downarrow n_j}$. Note that $\mathbf{D}^{\uparrow k} = \mathbf{D}_i$. Due to Proposition 2(i), $\mathbf{D}_j = \mathbf{D}_{j-1}[n_j \mapsto_\mathrm{D} \mathbf{D}_{\downarrow n_j}]$, for any $1 \leq j \leq i$. Therefore, $\mathbf{D} = \mathbf{D}_0 = \mathbf{D}^{\uparrow k}[n_1 \mapsto_\mathrm{D} \mathbf{D}_1] \ldots [n_i \mapsto_\mathrm{D} \mathbf{D}_i]$, which is equal to $\mathbf{D}^{\uparrow k}[n \mapsto_\mathrm{D} \mathbf{D}_{\downarrow n} : \forall n \in N^{k-}]$. □

Lemma 2. *Let $\mathbf{D} = (N, r, \_^\uparrow, \mathcal{G}, \mathcal{C})$ be a CD and $0 \leq k \leq depth(\mathbf{D})$. Then, $\mathcal{PL}(\mathbf{D}) = \mathcal{PL}(\mathbf{D}^{\uparrow k})[n \mapsto_\mathrm{P} \mathcal{PL}(\mathbf{D}_{\downarrow n}) : \forall n \in N^{k-}]$.*

Proof. Let $N^{k-} = \{n_1, \ldots, n_i\}$. We define a set of CDs $\{\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_i\}$ recursively as follows: $\mathbf{D}_0 = \mathbf{D}$ and for any $1 \leq j \leq i$: $\mathbf{D}_j = {\mathbf{D}_{j-1}}^{-\downarrow n_j}$. Note that $\mathbf{D}^{\uparrow k} = \mathbf{D}_i$. Due to Proposition 2(ii), $\mathcal{PL}(\mathbf{D}_{j-1}) = \mathcal{PL}(\mathbf{D}_j)[n_j \mapsto_\mathrm{P} \mathcal{PL}(\mathbf{D}_{\downarrow n_j})]$. Therefore, $\mathcal{PL}(\mathbf{D}) = \mathcal{PL}(\mathbf{D}_0) = \mathcal{PL}(\mathbf{D}^{\uparrow k})[n_1 \mapsto_\mathrm{P} \mathcal{PL}(\mathbf{D}_1)] \ldots [n_i \mapsto_\mathrm{P} \mathcal{PL}(\mathbf{D}_i)]$, which is equal to $\mathcal{PL}(\mathbf{D}^{\uparrow k})[n \mapsto_\mathrm{P} \mathcal{PL}(\mathbf{D}_{\downarrow n}) : \forall n \in N^{k-}]$. □

Lemma 3. *Let $\mathbf{D} = (N, r, \_^\uparrow, \mathcal{G}, \mathcal{C})$ be a CD and $0 \leq k \leq depth(\mathbf{D})$. Then, $\mathcal{E}_\mathbf{D} = \mathcal{E}_{\mathbf{D}^{\uparrow k}}[n \mapsto_\mathrm{E} \mathcal{E}_{\mathbf{D}_{\downarrow n}} : \forall n \in N^{k-}]$.*

Proof. Let $N^{k-} = \{n_1, \ldots, n_i\}$. We define a set of CDs $\{\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_i\}$ recursively as follows: $\mathbf{D}_0 = \mathbf{D}$ and for any $1 \leq j \leq i$: $\mathbf{D}_j = {\mathbf{D}_{j-1}}^{-\downarrow n_j}$. Note that

$\mathbf{D}^{\uparrow k} = \mathbf{D}_i$. Due to Proposition 2(iii), $\mathcal{E}_{\mathbf{D}_{j-1}} = \mathcal{E}_{\mathbf{D}_j}[n_j \mapsto_{\mathrm{E}} \mathcal{E}_{\mathbf{D}_{\downarrow n_j}}]$. Therefore, $\mathcal{E}_{\mathbf{D}} = \mathcal{E}_{\mathbf{D}_0} = \mathcal{E}\mathbf{D}^{\uparrow k}[n_1 \mapsto_{\mathrm{E}} \mathcal{E}_{\mathbf{D}_1}]\dots[n_i \mapsto_{\mathrm{E}} \mathcal{E}_{\mathbf{D}_i}]$, which is equal to $\mathcal{E}_{\mathbf{D}^{\uparrow k}}[n \mapsto_{\mathrm{E}} \mathcal{E}_{\mathbf{D}_{\downarrow n}} : \forall n \in N^{k-}]$. □

Now we are at the point where we can prove that the generated expression for a given CFD satisfies the property **P-1**.

Theorem 2 (Satisfying P-1). *For a given CFD* $\mathbf{FD}$, $\mathcal{L}(\mathcal{E}_{\mathbf{FD}})^{\mathrm{bag}} = \mathcal{PL}(\mathbf{FD})$. □

Proof. Let $\mathbf{FD} = (\mathbf{D}, F, l)$ be a CFD. We first prove the theorem on the underlying CD $\mathbf{D}$. Then, by applying the labelling function l on $\mathcal{E}_{\mathbf{D}}$, we prove that the multi-set interpretation of $\mathcal{E}_{\mathbf{FD}}$ satisfies **P-1**.

Let $\mathbf{D}$ be a CD. We use an inductive reasoning to prove the statement $\mathcal{L}(\mathcal{E}_{\mathbf{D}})^{\mathrm{bag}} = \mathcal{PL}(\mathbf{D})$.

(*basic step*): If $\mathbf{D}$ is a singleton tree, i.e., $depth(\mathbf{D}) = 0$, the statement follows obviously.

(*hypothesis*): Assume that the statement holds for any CD $\mathbf{D}$ with $depth(\mathbf{D}) \leq k$ for some $k \in \mathbb{N}$.

(*inductive step*): We want to prove that for any CD $\mathbf{D}$ with $depth(\mathbf{D}) = k+1$ the statement holds, i.e., $\mathcal{L}(\mathcal{E}_{\mathbf{D}})^{\mathrm{bag}} = \mathcal{PL}(\mathbf{D})$.

Let $\mathbf{D} = (N, r, \_^{\uparrow}, \mathcal{G}, \mathcal{C})$ be a CD with $depth(\mathbf{D}) = k+1$.

Due to Lemma 1, $\mathbf{D} = \mathbf{D}^{\uparrow k}[n \mapsto_{\mathrm{D}} \mathbf{D}_{\downarrow n} : \forall n \in N^{k-}]$.

Due to Lemma 3, $\mathcal{E}_{\mathbf{D}} = \mathcal{E}_{\mathbf{D}^{\uparrow k}}[n \mapsto_{\mathrm{E}} \mathcal{E}_{\mathbf{D}_{\downarrow n}} : \forall n \in N^{k-}]$.

Therefore, $\mathcal{L}(\mathcal{E}_{\mathbf{D}}) = \mathcal{L}(\mathcal{E}_{\mathbf{D}^{\uparrow k}})[n \mapsto_{\mathrm{L}} \mathcal{L}(\mathcal{E}_{\mathbf{D}_{\downarrow n}}) : \forall n \in N^{k-}]$.

(Note that the bag interpretation of any language $\mathcal{L}$ can be seen as a PL: using FLs in place of PLs in Definition 9 is allowed, e.g., $\mathcal{L}^{\mathrm{bag}}[\sigma \mapsto_{\mathrm{P}} \mathcal{L}'^{\mathrm{bag}}]$ makes sense for any $\sigma \in \Sigma(\mathcal{L})$.)

Obviously, $\mathcal{L}(\mathcal{E}_{\mathbf{D}})^{\mathrm{bag}} = \mathcal{L}(\mathcal{E}_{\mathbf{D}^{\uparrow k}})^{\mathrm{bag}}[n \mapsto_{\mathrm{P}} \mathcal{L}(\mathcal{E}_{\mathbf{D}_{\downarrow n}})^{\mathrm{bag}} : \forall n \in N^{k-}]$.

According to the hypothesis, since $depth(\mathbf{D}^{\uparrow k}) = k$, $\mathcal{L}(\mathcal{E}_{\mathbf{D}^{\uparrow k}})^{\mathrm{bag}} = \mathcal{PL}(\mathbf{D}^{\uparrow k})$.

Acceding to the hypothesis, since for any $n \in N^{k-}$: $depth(\mathbf{D}_{\downarrow n}) < k$, $\mathcal{L}(\mathcal{E}_{\mathbf{D}_{\downarrow n}})^{\mathrm{bag}} = \mathcal{PL}(\mathbf{D}_{\downarrow n})$.

Therefore, $\mathcal{L}(\mathcal{E}_{\mathbf{D}})^{\mathrm{bag}} = \mathcal{PL}(\mathbf{D}^{\uparrow k})[n \mapsto_{\mathrm{P}} \mathcal{PL}(\mathbf{D}_{\downarrow n}) : \forall n \in N^{k-}]$.

Due to Lemma 2, since $\mathcal{PL}(\mathbf{D}) = \mathcal{PL}(\mathbf{D}^{\uparrow k})[n \mapsto_{\mathrm{P}} \mathcal{PL}(\mathbf{D}_{\downarrow n}) : \forall n \in N^{k-}]$, $\mathcal{L}(\mathcal{E}_{\mathbf{D}})^{\mathrm{bag}} = \mathcal{PL}(\mathbf{D})$. The theorem is proven for CDs.

$\mathcal{PL}(\mathbf{FD}) = \mathcal{PL}(\mathbf{D})[l] = \mathcal{L}(\mathcal{E}_{\mathbf{D}})^{\mathrm{bag}}[l]$. According to Proposition 1, $\mathcal{E}_{\mathbf{D}}[l] = \mathcal{E}_{\mathbf{FD}}$. Therefore, $\mathcal{L}(\mathcal{E}_{\mathbf{D}})^{\mathrm{bag}}[l] = \mathcal{L}(\mathcal{E}_{\mathbf{FD}})^{\mathrm{bag}}$, which implies $\mathcal{PL}(\mathbf{FD}) = \mathcal{L}(\mathcal{E}_{\mathbf{FD}})^{\mathrm{bag}}$. □

4.4 Complexity Analysis of CRDs to REs Transformation

In this section, we analyze computational complexity of CRDs to REs transformation. We show that the transformation algorithm is a polynomial algorithm.

Algorithm 1 presents a pseudo code for the function defined in Definition 14. This algorithm is given a CRD and a leaf node and returns the $n's$ corresponding regular expression in the CRD. Its time complexity is in the class $O(|\mathcal{C}(n)|)$. Let us consider an upper bound on the number cardinality intervals assigned to a node or a group. Let denote this number by upC. Then the complexity class of this algorithm is reduced to $O(1)$.

Algorithm 1 : *lex*

{Input: A CRD $\mathbf{RD} = ((N, r, \_^{\uparrow}, \Sigma, l_{re}), \mathcal{G}, \mathcal{C})$}
{Input: $n \in lev(\mathbf{RD})$ with $\mathcal{C}(n) = \{(k_i, m_i)\}_{1 \leq i \leq j}$}
$R \leftarrow \varepsilon$
for $i = 1$ to j **do**
  if $m_i \neq *$ **then**
    $R \leftarrow R + \ l_{re}(n)^{k_i} + \ldots + l_{re}(n)^{m_i}$
  else
    $R \leftarrow R + \ l_{re}(n)^{k_i}(l_{re}(n))^*$
  end if
end for
return R {Comment: $lex_{\mathbf{RD}}(n) = R$}

Algorithm 2 presents a pseudo code for the first stage (Definition 15). It is time complexity would be in $O(|N| + |\mathcal{G}|) + O(N \times |lev(\mathbf{RD})| \times upC)$. Since the number of nodes is always greater than the number of groups (i.e., $|N| > |\mathcal{G}|$), the complexity class of the algorithm would be $O(|N|)+O(N\times|lev(\mathbf{RD})|\times upC)$. Obviously, this class can be reduced to $O(N^2)$.

Algorithm 3 presents a pseudo code for the function defined in Definition 16. This algorithm is given a CRD and a leaf group and returns the group's corresponding regular expression in the CRD. Its time complexity is in the class $O(|\mathcal{C}(G)| \times |G|^{m_i})$. This class can be reduced to $O(upC \times N^{m_i})$. Let consider a upper bound on the number of nodes involved in a group. Let upG. Then the complexity class of this algorithm would be reduced to $O(upC \times N^{upG})$, which can be reduced to $O(N^{upG})$.

Algorithm 4 corresponds to the second stage (Definition 17). Its time complexity would in the class $O(|N'_{-r} \uplus \mathcal{G}'|) + O(|N'_{-r}|) + O(|N'_{-r}| - |N_G|) + O(|N_G| \times N^{upG})$. This class can be reduced to $O(N^{upG+1})$.

Algorithm 5 presents a pseudo code for the function defined in Definition 18. Its time complexity would be in $O(|n_{\downarrow}|^{|n_{\downarrow}|})$. However, if we consider a bound on the number of children of nodes, then the complexity of this algorithm would be practically reasonable. Let deg denote the this bounded number. Then, this class would be reduced to $O(1)$.

Algorithm 6 is a pseudo code corresponding to the third stage (Definition 19). Its time complexity would be in the class of $O(|N'|)$, which is equal to $O(N - cplev(\mathbf{RD}))$. This class can be reduced to $O(N)$.

According to above complexity analyses of the stages, the shrinking step would be a in a polynomial complexity. More precisely, it would be $O(N^{upG+1})+$

Algorithm 2 : *cel* (Stage 1)

{Input: A CRD $\mathbf{RD} = ((N, r, \_^{\uparrow}, \Sigma, l_{re}), \mathcal{G}, \mathcal{C})$}
{Output: A CRD $\mathbf{RD}' = ((N, r, \_^{\uparrow}, \Sigma, l'_{re}), \mathcal{G}, \mathcal{C}')$}
Ensure: $\forall e \in lev(\mathbf{RD}')$. $\mathcal{C}'(e) = \{(1, 1)\}$

for all $e \in N_{-r} \uplus \mathcal{G}$ **do**
  $\mathcal{C}'(e) \leftarrow \varnothing$
end for

for all $n \in N$ **do**
  $l'_{re}(n)) \leftarrow \varepsilon$
end for

for all $e \in N_{-r} \uplus \mathcal{G}$ **do**

  if $e \in lev(\mathbf{RD})$ **then**
    $\mathcal{C}'(e) \leftarrow \{(1, 1)\}$
  else
    $\mathcal{C}'(e) \leftarrow \mathcal{C}(e)$
  end if
end for

for all $n \in N$ **do**

  if $n \in lev(\mathbf{RD})$ **then**
    $l'_{re}(n) \leftarrow lex(\mathbf{RD}, n)$
  else
    $l'_{re}(n)) \leftarrow l_{re}(n)$
  end if
end for
return $((N, r, \_^{\uparrow}, \Sigma, l'_{re}), \mathcal{G}, \mathcal{C}')$

Algorithm 3 : *gex*

{Input: A CRD $\mathbf{RD} = ((N, r, \_^{\uparrow}, \Sigma, l_{re}), \mathcal{G}, \mathcal{C})$}
{Input: $G \in glev(\mathbf{RD})$ with $\mathcal{C}(G) = \{(k_i, m_i)\}_{1 \leq i \leq j}$}
$R \leftarrow \varepsilon$
for $i = 1$ to j **do**
  for all permutation R' with $k_i \leq \text{length}(R') \leq m_i$ in $\{l_{re}(n) : n \in G\}$ **do**
    $R \leftarrow R + R'$
  end for
end for
return R {Comment: $gex_{\mathbf{RD}}(n) = R$}

Algorithm 4 : *gle* (Stage 2)

{Input: A CRD $\mathbf{RD} = ((N, r, \_^{\uparrow}, \Sigma, l_{re}), \mathcal{G}, \mathcal{C})$}
{Output: A CRD $\mathbf{RD'} = ((N', r, \_^{\uparrow'}, \Sigma, l'_{re}), \mathcal{G}', \mathcal{C}')$}
Require: $\forall e \in lev(\mathbf{RD}).\ \mathcal{C}(e) = \{(1,1)\}$
Ensure: $glev(\mathbf{RD'}) = \varnothing$
$N' \leftarrow (N - glev(\mathbf{RD})) \uplus N_G$
$\mathcal{G}' \leftarrow \mathcal{G} - glev(\mathbf{RD})$

for all $e \in N'_{-r} \uplus \mathcal{G}'$ **do**
  $\mathcal{C}'(e) \leftarrow \varnothing$
end for

for all $n \in N'$ **do**
  $l'_{re}(n) \leftarrow \varepsilon$
end for
$^{\uparrow'} = \varnothing$

for all $e \in N'_{-r} \uplus \mathcal{G}'$ **do**

  if $e \in N_G$ **then**
    $\mathcal{C}'(e) \leftarrow \{(1,1)\}$
  else
    $\mathcal{C}'(e) \leftarrow \mathcal{C}(e)$
  end if
end for

for all $n \in N'_{-r}$ **do**

  if $n \in N_G$ **then**
    $n^{\uparrow'} \leftarrow gid(n)^{\uparrow}$
  else
    $n^{\uparrow'} \leftarrow n^{\uparrow}$
  end if
end for

for all $n \in N'$ **do**

  if $n \in N_G$ **then**
    $l'_{re}(n) \leftarrow gex(\mathbf{RD}, gid(n))$
  else
    $l'_{re}(n)) \leftarrow l_{re}(n)$
  end if
end for
return $((N', r, \_^{\uparrow'}, \Sigma, l'_{re}), \mathcal{G}', \mathcal{C}')$

Algorithm 5 : *pex*

{Input: A CRD $\mathbf{RD} = ((N, r, \_^{\uparrow}, \Sigma, l_{re}), \mathcal{G}, \mathcal{C})$}
{Input: $n \in plev(\mathbf{RD})$}
$R \leftarrow \varepsilon$

for all permutations R' with $length(R') = |n_{\downarrow}|$ in $\{lre(n') : n' \in n_{\downarrow}\}$ **do**
  $R \leftarrow R + R'$
end for
return $lre(n).R$ {Comment: $pex_{\mathbf{RD}}(n) = lre(n).R$}

Algorithm 6 : *dre* (Stage 3)

{Input: A CRD $\mathbf{RD} = ((N, r, \_^{\uparrow}, \Sigma, l_{re}), \mathcal{G}, \mathcal{C})$}
{Output: A CRD $\mathbf{RD}' = ((N', r, \_^{\uparrow\prime}, \Sigma, l'_{re}), \mathcal{G}, \mathcal{C}')$}
Require: $glev(\mathbf{RD}) = \varnothing$
Ensure: $depth(\mathbf{RD}') = depth(\mathbf{RD}) - 1$
$N' \leftarrow (N - cplev(\mathbf{RD}))$
$\_^{\uparrow\prime} \leftarrow \_^{\uparrow}|_{N'}$
$\mathcal{C}' \leftarrow \mathcal{C}|_{N' \cup \mathcal{G}}$

for all $n \in N'$ **do**
  $l'_{re}(n) \leftarrow \varepsilon$
end for

for all $n \in N'$ **do**

  if $n \in plev(\mathbf{RD})$ **then**
    $l'_{re}(n) \leftarrow pex(\mathbf{RD}, n)$
  else
    $l'_{re}(n)) \leftarrow l_{re}(n)$
  end if
end for
return $((N', r, \_^{\uparrow\prime}, \Sigma, l'_{re}), \mathcal{G}, \mathcal{C}')$

$O(N^2)$. Thus, if the CRD has no grouped node, then the time complexity of shrinking steps would be in $O(N^2)$. Otherwise, since $upG \geq 2$ (a group is reasonable if at least two nodes are involved in it), the time complexity of the shrinking step would be in $O(N^{upG+1})$.

Finally, Algorithm 7 presents a pseudo code for the transformation of CRDs to REs. The time complexity of this algorithm would be in $O(depth(\mathbf{RD}) \times N^{upG+1})$. In the worst case, the time complexity would be in $O(N^{upG+1})$, which implies that the transformation algorithm is a polynomial algorithm.

Algorithm 7 : Transformation Algorithm

{Input: A CRD $\mathbf{RD} = ((N, r, \_^{\uparrow}, \Sigma, l_{re}), \mathcal{G}, \mathcal{C})$}
{Output: A CRD $\mathbf{RD}' = ((N', r, \_^{\uparrow'}, \Sigma, l'_{re}), \mathcal{G}', \mathcal{C}')$}
Ensure: $|N'| = 1$
  Initiate $\mathbf{RD}'$
  while $depth(\mathbf{RD}) \geq 2$ **do**
    $\mathbf{RD}' \leftarrow dre(gle(cel(\mathbf{RD}')))$
  end while
  return $\mathbf{RD}'$

5 CCs and CFMs

CCs only make sense with respect to a given CFD. In the previous section, we formalized the semantics of CFDs using formal languages (more precisely, regular languages). Hence, it makes sense to use the same framework to express CCs. This will allow us to integrate the semantics of CCs and CFDs. In the following, we show how to translate the most common CCs using formal languages. Assume a CFD with a set of features F including two features f_1, f_2, and f_3. Several interesting CCs applied to a CFM are as follows:

(cc$_1$) f_1 requires f_2
(in other words: If the number of instances of f_1 is greater than 0, then the number of instances of f_2 must be greater than 0).

(cc$_2$) f_1 excludes f_2
(in other words: If the number of instances of f_1 is greater than 0, then the number of instances of f_2 must be 0).

(cc$_3$) If the number of instances of f_1 is even, then the number of instances of f_2 must be odd.
(cc$_4$) The number of instances of f_1 and f_2 are equal.

(cc_5) The number of instances of f_1, f_2, and f_3 are equal.

The first two CCs are traditional inclusive and exclusive CCs. However, they can be expressed in terms of feature instances, as we see in the parenthetical remarks above. Our method to express CCs is to use formal languages. In this approach, features are considered as alphabets of a language. In the following, we see the formal language interpretation of the above CCs. The formal language of a given CC cc is denoted by $\mathcal{L}(\mathrm{cc})$.

$$\begin{aligned}
\mathcal{L}(\mathrm{cc}_1) &= \big\{w \in F^* : \big(\#_{f_1}(w) > 0\big) \Rightarrow \big(\#_{f_2}(w) > 0\big)\big\}. \\
\mathcal{L}(\mathrm{cc}_2) &= \big\{w \in F^* : \big(\#_{f_1}(w) > 0\big) \Rightarrow \big(\#_{f_2}(w) = 0\big)\big\}. \\
\mathcal{L}(\mathrm{cc}_3) &= \big\{w \in F^* : (\exists n \in \mathbb{N}.\#_{f_1}(w) = 2n) \Rightarrow (\exists n \in \mathbb{N}.\#_{f_1}(w) = 2n+1)\big\}. \\
\mathcal{L}(\mathrm{cc}_4) &= \big\{w \in F^* : \#_{f_1}(w) = \#_{f_2}(w)\big\}. \\
\mathcal{L}(\mathrm{cc}_5) &= \big\{w \in F^* : \#_{f_1}(w) = \#_{f_2}(w) = \#_{f_3}(w)\big\}.
\end{aligned}$$

Proposition 3. *$\mathcal{L}(cc_1)$, $\mathcal{L}(cc_2)$, and $\mathcal{L}(cc_3)$ are regular, $\mathcal{L}(cc_4)$ is context-free, and $\mathcal{L}(cc_5)$ is context-sensitive.* □

Proof. A language is regular iff it can be expressed by some regular expressions, regular grammars, or finite state automata (FSA). Let $F = \{f_1, \ldots, f_n\}$ for some $n \geq 3$.

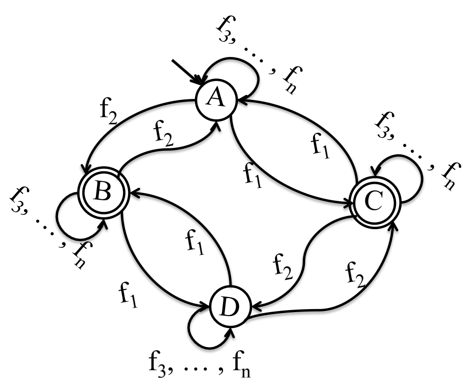
$\mathcal{L}(\mathrm{cc}_1)$ can be expressed by the following regular expression, where $r = (f_1 + \ldots + f_n)^*$:

$$f_2^* + rf_1rf_2r + rf_2rf_1r.$$

$\mathcal{L}(\mathrm{cc}_2)$ can be expressed by the following regular expression:

$$(f_2 + \ldots + f_n)^* + (f_1 + f_3 + \ldots + f_n)^* + (f_3 + \ldots + f_n)^*.$$

The following FSA accepts $\mathcal{L}(\mathrm{cc}_3)$. The initial state is identified by an incoming unlabelled arrow not originating at any state. The final states are drawn with double circles.



$\mathcal{L}(\mathrm{cc}_4)$ and $\mathcal{L}(\mathrm{cc}_5)$ are very well-known context-free and context-sensitive languages, respectively.

Theorem 3. *Given a context-free FM $\mathbb{M}$, the operations Void Feature Models, Dead Features, Valid Product, Core Features, and Least Common Ancestor are decidable.* □

Proof (**Proof of Theorem 3).**
Let F be the set of features of $\mathbb{M}$.

Since context-free languages are decidable, the Valid Product problem is decidable.

The emptiness problem of context-free languages is decidable. Thus, the Void Feature Model problem is decidable.

Let $\mathcal{L}$ be the language of the expression $F^* f F^*$. The problem of determining whether the feature f is a dead feature of $\mathbb{M}$ or not is, indeed, to determine whether $\mathcal{L} \cap \mathcal{L}_{\mathbb{M}} = \varnothing$ or not. Note that $\mathcal{L}$ is regular. Hence, $\mathcal{L} \cap \mathcal{L}_{\mathbb{M}}$ is context-free. Since the emptiness problem of context-free languages is decidable, the Dead Feature problem is decidable too.

Consider a subset $P \subseteq F$. We want to determine whether the set of features P is included in all products or not. Let $|P| = n$, $\mathcal{L}' = \{w \in F^* : w^{\text{bag}} = P\}$, and $\mathcal{L} = \{w_1 a'_1\ w_2 a'_2\ \ldots\ w_n a'_n\ w_{n+1} : a'_1 \ldots a'_n \in \mathcal{L}' \text{ and } w_i \in F^*\}$. The problem is reduced to determining whether $\mathcal{L}_{\mathbb{M}} \subseteq \mathcal{L}$ or not. In other words, the problem is reduced todetermining whether $\mathbb{M} \cap \mathcal{L}^c = \varnothing$ or not ($\mathcal{L}^c$ denotes the complement of $\mathcal{L}$). Note that $\mathcal{L}$ is a regular language and so is $\mathcal{L}^c$. Hence, the language $\mathbb{M} \cap \mathcal{L}^c$ is context-free. Since the emptiness problem in the class of context-free languages is decidable, the original problem, i.e., determining if P is included in all products, is decidable. Since the number of subsets of F is finite, the problem of finding the set of Core Features is also decidable. □

Remark 3. What we need in cc_4 is counting the number of instances of f_1 and f_2. If the order of the symbols is ignored, then, according to the Parikh's theorem [29], $\mathcal{L}(\text{cc}_4)$ as a context-free language is not indistinguishable from a regular language.

Hence, a CFM is a CFD plus a set of languages expressing the CCs. In fact, a set of CCs can be seen as the intersection of the languages expressing the CCs.

Definition 22 (Cardinality-based Feature Models). *A* cardinality-based feature model *(CFM) is a pair $\mathbb{M} = (\mathbf{FD}, \mathcal{L}_{cc})$ with $\mathbf{FD}$ a CFD and $\mathcal{L}_{cc}$ a language built over F (the set of features) expressing the CCs.* □

Thus, a CFM is basically a tuple of formal languages $\mathbb{M} = (\mathcal{L}_{\mathbf{FD}}, \mathcal{L}_{cc})$ with $\mathcal{L}_{\mathbf{FD}}$ and $\mathcal{L}_{cc}$ denoting the FLs of the CFD **FD** and CCs, respectively. The formal language associated with the whole model is denoted by $\mathcal{L}_{\mathbb{M}}$ and is equal to $\mathcal{L}_{\mathbf{FD}} \cap \mathcal{L}_{cc}$. Since any class of languages is closed under intersection with regular languages [11] and $\mathcal{L}_{\mathbf{FD}}$ is regular, the type of $\mathcal{L}_{\mathbb{M}}$ is given by the type of $\mathcal{L}_{cc}$. Hence, CFMs can be grouped based on the types of their language, say regular and context-free FMs. This grouping is important because it guides us in how FMs can be constructively analyzed.

Definition 23 (Dynamic & Static Semantics). *For a given FM* $\mathbb{M}$,

(i) $\mathcal{L}_{\mathbb{M}}$ *is called the* dynamic semantics *of* $\mathbb{M}$. *Any word* $w \in \mathcal{L}_{\mathbb{M}}$ *is called a dynamic product. We then write* $w \models_{\boldsymbol{DY}} \mathbb{M}$.

Two models $\mathbb{M}$ *and* $\mathbb{M}'$ *are called dynamic equivalent, denoted by* $\mathbb{M} \equiv_{\boldsymbol{ST}} \mathbb{M}'$, *if and only if* $\mathcal{PL}(\mathbb{M}) = \mathcal{PL}(\mathbb{M}')$.

(ii) The multi-set interpretation of $\mathcal{L}_{\mathbb{M}}$, $\mathcal{L}_{\mathbb{M}}^{\text{bag}}$, *is called the* static semantics *of* $\mathbb{M}$. *Any element* P *of* $\mathcal{L}_{\mathbb{M}}^{\text{bag}}$ *is called a* static product. *We then write* $P \models_{\boldsymbol{ST}} \mathbb{M}$.

Two models $\mathbb{M}$ *and* $\mathbb{M}'$ *are called dynamic equivalent,* $\mathbb{M} \equiv_{\boldsymbol{ST}} \mathbb{M}'$, *if and only if* $\mathcal{PL}(\mathbb{M}) = \mathcal{PL}(\mathbb{M}')$. □

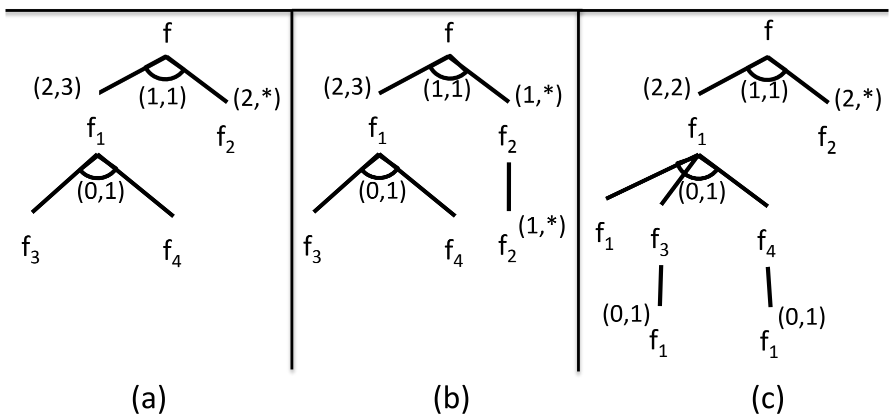


Fig. 10: (a) $\mathbb{M}$, (b) $\mathbb{M}'$ ($\equiv_{\mathbf{DY}}$ $\mathbb{M}$) , (c) $\mathbb{M}''$ ($\equiv_{\mathbf{ST}}$ $\mathbb{M}$)

As an example, consider the three models $\mathbb{M}$, $\mathbb{M}'$, and $\mathbb{M}''$ in Fig. 10(a), (b), (c), respectively. The regular expression encoding of $\mathbb{M}$ is $\mathcal{E}_{\mathbb{M}} = f.(f_2.f_2.(f_2)^* + f_1.f_1.(\varepsilon + f_1).(\varepsilon + f_3 + f_4))$. The regular expression encoding of $\mathbb{M}'$ is $\mathcal{E}_{\mathbb{M}'} = f.(f_2.(f_2)^*.f_2.(f_2)^* + f_1.f_1.(\varepsilon + f_1).(\varepsilon + f_3 + f_4))$. It is obvious that $\mathcal{L}(\mathcal{E}_{\mathbb{M}}) = \mathcal{L}(\mathcal{E}_{\mathbb{M}'})$, which means $\mathbb{M} \equiv_{\mathbf{DY}} \mathbb{M}'$. On the other hand, $\mathbb{M}''$ is not dynamic equivalent to $\mathbb{M}$. $\mathbb{M}''$ and $\mathbb{M}$ are static equivalent, i.e., $\mathbb{M}'' \equiv_{\mathbf{ST}} \mathbb{M}$.

Remark 4. The above example shows obviously that static semantics (PL) is a poor abstract view for CFMs, while the dynamic semantics (FL) extract much more semantics of CFMs.

6 Analysis Operations

In this section, we investigate the decidability problem for some well-known analysis operations. Some operations take only one FM (along with another potential input that is not an FM) as input and perform some analysis on the FM. Below is a sample list of such operations:

Valid Product: takes an FM and a multi-set of features as inputs and decides whether it is a valid product of the FM or not.

Core Features: takes an FM and returns the set of features that are included in all the products.

Void Feature Model: takes an FM as input and decides whether its PL is empty or not.

Dead Feature: takes an FM and a feature and decides whether the feature is *dead* in the FM or not. A feature f in an FM $\mathbb{M}$ is called dead if $\nexists P \in \mathcal{PL}(\mathbb{M})$ such that $f \in P$.

Least Common Ancestor: takes an FD and a set of features and returns their lowest common ancestor feature.

Theorem 3. *Given a context-free FM $\mathbb{M}$, the operations Void Feature Models, Dead Features, Valid Product, Core Features, and Least Common Ancestor are decidable.* □

Proof. Let F be the set of features of $\mathbb{M}$.

Since context-free languages are decidable, the Valid Product problem is decidable.

The emptiness problem of context-free languages is decidable. Thus, the Void Feature Model problem is decidable.

Let $\mathcal{L} = F^*\{f\}F^*$. The problem of determining whether the feature f is a dead feature of $\mathbb{M}$ or not is, indeed, to determine whether $\mathcal{L} \cap \mathcal{L}_\mathbb{M} = \varnothing$ or not. Note that $\mathcal{L}$ is regular. Hence, $\mathcal{L} \cap \mathcal{L}_\mathbb{M}$ is context-free. Since the emptiness problem of context-free languages is decidable, the Dead Feature problem is decidable too.

Let $\mathcal{L}$ denote the set of all prefixes of the words of $\mathcal{L}_\mathbb{M}$. $\mathcal{L}$ is a context-free language. To prove this, we take the grammar of $\mathcal{L}_\mathbb{M}$ in Chomsky Normal Form and for every production $A \to BC$, add productions $A_\varepsilon \to BC_\varepsilon$ and $A_\varepsilon \to B_\varepsilon$. Also, for every production $A \to f$ (for some terminals f), we consider the production $A_\varepsilon \to f$. Finally, we change the starting variable S to S_ε and add the production $S_\varepsilon \to \varepsilon$. The context-free grammar generated in this way represents the language $\mathcal{L}$. Thus, $\mathcal{L}$ is decidable. The set of dynamic partial products would be equal to the bag interpretation of $\mathcal{L}$. Thus, Dynamic Partial Product problem is decidable.

Let P be an input of the Static Partial Product operation. P is finite and the arity of any feature $f \in P$ is in $\mathbb{N}$. Let the number of feature instances in P be n and $\mathcal{L}' = \{w \in F^* : w^{\text{bag}} = P\}$. Now consider the regular language $\mathcal{L} = \{w_1 a'_1 \ w_2 a'_2 \ \ldots \ w_n a'_n \ w_{n+1} : a'_1 \ldots a'_n \in \mathcal{L}' \text{ and } w_i \in F^*\}$. The Static Partial Product problem is reduced to determining whether $\mathcal{L} \cap \mathcal{L}_\mathbb{M}$ is empty or not. $\mathcal{L} \cap \mathcal{L}_\mathbb{M}$ is context free, since $\mathcal{L}$ is regular and $\mathcal{L}_\mathbb{M}$ is context-free. Since the emptiness problem of context-free languages is decidable, the Static Partial Product problem would be decidable.

Consider a subset $P \subseteq F$. We want to determine whether the set of features P is included in all products or not. Let $|P| = n$, $\mathcal{L}' = \{w \in F^* : w^{\text{bag}} = P\}$, and $\mathcal{L} = \{w_1 a'_1\ w_2 a'_2\ \ldots\ w_n a'_n\ w_{n+1} : a'_1 \ldots a'_n \in \mathcal{L}' \text{ and } w_i \in F^*\}$. The problem is reduced to determining whether $\mathcal{L}_{\mathbb{M}} \subseteq \mathcal{L}$ or not. In other words, the problem is reduced to determining whether $\mathbb{M} \cap \mathcal{L}^c = \varnothing$ or not ($\mathcal{L}^c$ denotes the complement of $\mathcal{L}$). Note that $\mathcal{L}$ is a regular language and so is $\mathcal{L}^c$. Hence, the language $\mathbb{M} \cap \mathcal{L}^c$ is context-free. Since the emptiness problem in the class of context-free languages is decidable, the original problem, i.e., determining if P is included in all products, is decidable. Since the number of subsets of F is finite, the problem of finding the set of Core Features is also decidable. □

Remark 5. Since the class of regular languages is a subclass of context-free languages, the above theorem holds for regular FMs too. Note that some analysis operations are not decidable in other classes of CFMs. For example, the Void Feature Model operation is not decidable in the class of context-sensitive CFMs, since the emptiness problem is not decidable in the class of context-sensitive languages.

Some other operations deal with two FMs. Such operations answer some questions about the relationships between the FMs.

Refactoring: takes two FMs and decides whether their PL are equal or not.

Specialization: takes two FMs $\mathbb{M}_1$ and $\mathbb{M}_2$ as inputs and decides whether the PL of $\mathbb{M}_1$ is a subset of the PL of $\mathbb{M}_2$ or not.

Theorem 4. *Given two FMs $\mathbb{M}_1$ and $\mathbb{M}_2$, the following statements hold:*

(i) If both are regular, then the Refactoring problem between them is decidable.

(ii) If $\mathbb{M}_1$ and $\mathbb{M}_2$ are regular and context-free, respectively, then the Refactoring problem is decidable iff $\mathbb{M}_1$ is bounded regular. □

Proof.

(i) The equality problem between regular languages is decidable [25].

(ii) Hopcroft in [20] showed that for two given context-free languages $\mathcal{L}_1$ and $\mathcal{L}_2$, if one of them, say $\mathcal{L}_1$, is a bounded regular language, then the equality problem between these two languages is decidable. □

Remark 6. In general, the equality problem in the class of context-free languages is undecidable. Therefore, the Refactoring problem is not decidable in the class of context-free FMs.

Theorem 5. *Given two FMs $\mathbb{M}_1$ and $\mathbb{M}_2$, the following statements hold:*

(i) If both are regular, then the Specialization problem between them is decidable.

(ii) If $\mathbb{M}_1$ and $\mathbb{M}_2$ are regular and context-free, respectively, then the Specialization problem $\mathcal{PL}(\mathbb{M}_2) \subseteq \mathcal{PL}(\mathbb{M}_1)$ is decidable. □

Proof.

(i) The inclusion problem in the class of regular languages is decidable [28]. Thus, the Specialization problem is decidable in the class of regular languages.

(ii) The problem "$\mathbb{M}_2$ is a specialization of $\mathbb{M}_1$" is reducible to the problem $\mathcal{L}_{\mathbb{M}_2} \subseteq \mathcal{L}_{\mathbb{M}_1}$. In other words, it is equivalent to the problem of determining whether $\mathcal{L}_{\mathbb{M}_2} \cap \mathcal{L}^c_{\mathbb{M}_1} = \varnothing$ or not. Since the class of regular languages is closed under complement, $\mathcal{L}^c_{\mathbb{M}_1}$ is regular. Thus, $\mathcal{L}_{\mathbb{M}_2} \cap \mathcal{L}^c_{\mathbb{M}_1}$ is context-free. Since the emptiness problem in the class of context-free languages is decidable, the Specialization problem in this case would be decidable. □

7 Tool Support

As already seen, we characterize well-known analysis operations over CFMs in terms of on formal languages. Now, what we need is automated support for them. Since CFMs are much more complex than basic FMs, automated analysis of such FMs is a challenging and open issue [4, 31]. As discussed in Sect. 6, all the analysis operations are decidable over the class of regular CFMs. Therefore, in this section, we only take into account regular CFMs. Recall that a CFM is regular if its CCs are regular.

There are several off-the-shelf tools, which deal with finite state automata (FSA) including HKC [5], LIBVATA [24], RABIT [1], ALASKA [13], GOAL [36], FSA6.2xx [38], FAT [17], and JFLAP [32].

Since most of the above tools are given FSAs as inputs, we first need to translate a given regular expression to a finite automata. Note that we transform a given CFD to a regular expression. Fortunately, there exist some tools, which support such a transformation. For an example, we can use FSA6.2xx for this purpose.

Since CFDs and their CCs are translated to two different langaugues, we would need also to execute their intersection. FAT and FSA6.2xx are appropriate for implementing the intersection problem between two FSAs.

To reduce the computational complexity (specially the space complexity) in executing the analysis operations, we would prefer to work on an minimal automaton semantically equal to the given automaton. This problem is so called minimization problem, which means transformation of a given automaton to another automaton such that the language of the generated automaton is equal to the language of the given one and also it is minimal in terms of the number of states. FSA6.2xx also does this mission very well. Now we are at the point where we can utilize the above tools to do analysis operations on regular CFMs.

The valid product problem on a CFM is reduced to the membership problem on the CFM's language interpretation. FAT, JFLAP, and FSA6.2xx are appropriate for implementing the membership problem.

The void feature model problem is reduced to the emptiness problem on languages. The emptiness problem for a given language $\mathcal{L}$ can be seen as the equality problem between $\mathcal{L}$ and the empty language. The equality problem over FSAs is supported by HKC.

Consider a CFM $\mathbb{M}$ on a set of features F and a feature $f \in F$. We want to decide whether f is a dead feature over $\mathbb{M}$ or not. This problem can be reduced to the decision problem $\mathcal{L}(\mathbb{M}) \cap \mathcal{L}(F^* f F^*) = \varnothing$. Note that the language $\mathcal{L}(F^* f F^*)$ is regular and can be represented by an FSA. The intersection of the FSAs corresponding to the languages $\mathcal{L}(\mathbb{M})$ and $\mathcal{L}(F^* f F^*)$ can be done by FSA6.2xx. Let $\mathcal{A}$ denote the output FSA. Then the equality problem between $\mathcal{A}$ and the empty FSA can be executed by HKC.

The refactoring problem between two CFMs is simply reduced to the equality problem between their languages. The equality problem between FSAs is supported by HKC.

The specialization problem for two given CFMs can be reduced to the inclusion problem for their corresponding languages. The inclusion problem is supported by several tools including HKC, LIBVATA, RABIT, ALASKA, and RABIT.

8 Related Work

8.1 Connection between FDs and Context-free Grammars

In this section, we survey the literature relevant to the connection between FMs and FLs.

The connection between basic FDs and grammars is shown by de Jong and Visser [12]. They use textual representations of FDs written in a domain-specific language called the *feature description language* [37]. The corresponding textual representation of a given FD is similar to a context-free grammar. The grammar generated for the model in Fig. 11, according to [12], is as follows (nonterminals and terminals start with capital and small letters, respectively.):

RenovationFactory $\longrightarrow$ SourceLang ImplLang
SourceLang $\longrightarrow$ cobol | sdl | sql | cobol sdl | cobol sql | sdl sql | cobol sdl sql
ImplLang $\longrightarrow$ asf | java | asf traversal | java traversal

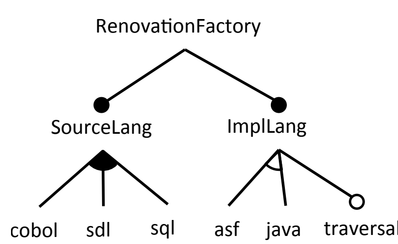


Fig. 11: A model abbreviated from [12]

Batory, in [3], shows the connection between FDs and iterative tree grammars [22]. His and [12]'s translation procedures are essentially the same. Table 1 gives some basic examples showing how Batory's encoding works. Terminals are shown by italic letters. Optional features are surrounded by brackets.

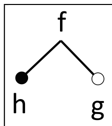
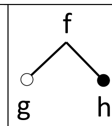
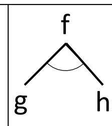
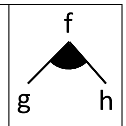
			
f$\rightarrow h[g]$	f$\rightarrow [g]h$	f$\rightarrow g \mid h$	f $\rightarrow$ t+ t $\rightarrow g \mid h$

Table 1: Translating FDs to iterative tree grammars

In [12] and [3], set of *atomic features*, those features that appear in leaf nodes, is considered as terminals and other features as nonterminals. Thus, a word accepted in the above grammar generated for Fig. 11 is a subset of {cobol, sdl, sql, asf, java, traversal} and hence the language of the corresponding grammar does not represent the product line of the model. In other words, the corresponding generative grammar for a given FD does not satisfy the property **P-1**.

Another problem of the above procedures is that they give a left-to-right ordering on siblings (the nodes with the same parent). To illustrate why this is a problem, note the left-most column in Table 1: the left-most feature, h, precedes the right-most feature, g. Such an ordering forces two syntactically equivalent FDs to have different semantics: the grammars of the two FDs in the first and the second columns in Table 1 have different associated languages. In addition, such an ordering on siblings forces the generative grammars to not satisfy the property **P-2**.

Czarnecki et al, in [8], formalize the semantics of CFDs using context-free grammars. Unlike in [3] and [12], this work considers the set of terminals to be equal to the set of all features for a given CFD and generative grammars satisfy the property **P-2**. However, it gives a left-to-right ordering on siblings. Thus, this method does not satisfy the property **P-1** and there are syntactically equivalent CFDs with non-equal generative grammars.

All the above approaches may result in ambiguous grammars, which makes them bad candidates for the semantics of models. However, there is a constructive way [25] to fix this problem, since the languages of generated grammars are not inherently ambiguous. A context-free language is inherently ambiguous if there is no unambiguous grammar for it [16]. Another problem of all of the above methods is that they do not consider CCs in their translation procedure. This is a very important deficiency, since CCs has a central role to play in feature modeling.

8.2 Partial Product Lines

In [15], we give a relational semantics for basic FMs. The structure corresponding to a given FM is called the *Partial Product Line* (PPL) of the FM. The states of this structure are called *partial products*, which are sets of features satisfying the *exclusive constraints* (a partial product must not violate the exclusive constraints), *subfeature relationship* (a feature cannot be included in a partial product if its parent feature is not), and the *instantiation-to-completion* (**I2C**) principle (processing a new branch of the feature tree should only begin after processing of the current branch has reached a full product). The initial state is a singleton set $\{r\}$ where r is the root feature. The PL of the FM is a subset of the set of partial products. Fig. 12(a) is an FM and its PPL is represented in Fig. 12 (b). In this figure, the full products are boxed. *Singletonicity* is one of

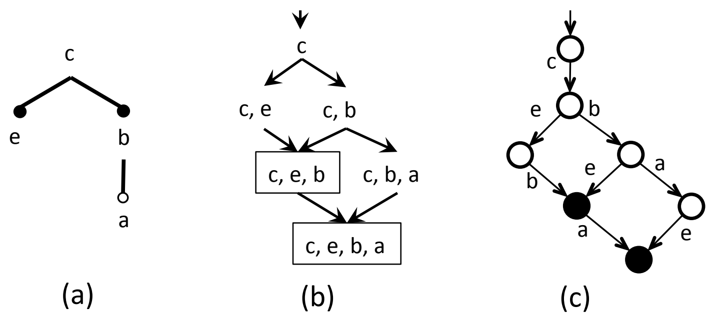


Fig. 12: (a) an FM $\mathbb{M}$, (b) PPL($\mathbb{M}$), (c) $Aut(\mathbb{M})$

the important properties of PPLs. This property says that if there is a transition $P \longrightarrow P'$ between two products P and P', then $P' = P \uplus \{f\}$ for some feature $f \notin P$. This property allows us to translate PPLs into finite state automata in a straightforward way. Fig. 12(c) shows the corresponding automaton, where the final states are specified by black circles. Let $Aut(\mathbb{M})$ denote the automaton corresponding to an FM $\mathbb{M}$.

Applying the translation procedure on $\mathbb{M}$ described in Sect. 4, $\mathcal{E}_{\mathbb{M}} = f\ (\ g(i+j)\ h + h\ (g\ (i+j)))$. It is clear that $\mathcal{L}(\mathcal{E}_{\mathbb{M}}) = \mathcal{L}(Aut(\mathbb{M}))$. What is interesting is that $Aut(\mathbb{M})$ is the minimal automaton in the sense that there is no other finite automaton with a smaller number of states which also accepts $\mathcal{L}(\mathbb{M})$. This claim will be proved formally for all basic FMs in a forthcoming paper. Note that this property of PPLs makes them effective in the sense of computational complexity. Also, this relationship between $\mathcal{L}_{\mathbb{M}}$ and $PPL(\mathbb{M})$ for a given basic FM $\mathbb{M}$ proves that PPLs preserve the hierarchical structure of $\mathbb{M}$.

8.3 Modeling PLs with Semirings

Höfner et al. developed an algebra, called *product family algebra*, for basic PLs whose basis is the structure of idempotent semirings [18]. A product family algebra over a set of features F is 5-tuple $\mathcal{A} = (A, +, \varnothing, \times, \{\varnothing\})$, where $A = 2^{2^F}$, $\varnothing$ represents the empty PL, $\{\varnothing\}$ is a dummy/pseudo PL with only one product: nothing, and $+, \times$ are defined as follows: for all $P, P' \in A : P \times P' = \{p \cup p' :$

$p \in P, p' \in P'\}$ and $P + P' = P \cup P'$. In this way, $+$ and $\times$ can be seen as choice between PLs and their mandatory presence, respectively. It is proven that $\mathcal{A}$ forms a semiring ,where $(A, +, 0)$ and $(A, \times, 1)$ are the commutative monoid and monoid parts, respectively, such that $+$ is idempotent and $\times$ is commutative. Therefore, a PL is seen as a term generated in a commutative idempotent semirings.

The PL of a given basic FM $\mathbb{M}$ is encoded as a term in the PL algebra generated over the leaf features of $\mathbb{M}$. As an example, consider the following feature diagram, which is adopted from [18]. The encoded term corresponding to this FM is as follows: $car = (manual + automatic) \times horsepower \times (1 + aircondition)$.

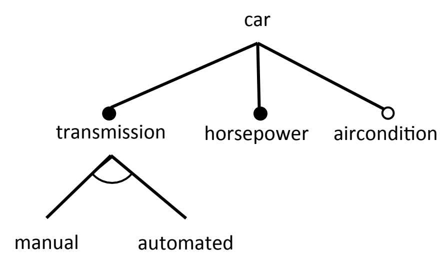


Fig. 13: an FM - adopted from [18] (page 7 , Fig. 1)

Note that the set of all formal languages (Σ^*) together with concatenation and union operations can be seen as a semiring $(\Sigma^*, \cup, \varnothing, ., \varepsilon)$. However, it is not a commutative smearing, since concatenation is not commutative. Also, this smearing is not idempotent.

As mentioned above, for Höfner et al., a product is the set of leaves in the feature tree, while non-leaf features are derived terms; in contrast, we follow a common FM-practice and consider all features in the tree to be basic. This implies that the product family algebra does not satisfy the **P-1** property.

Since there is the operation $\times$ is considered as an idempotent operation, the product family algebra for a given model does not satisfies the **P-2** property. Also, using an idempotent operation for the product operation $(\times)$ disallows us to use it for cardinality-based feature models.

9 Conclusions

In this paper, we have provided a formal definition of CFDs and also their valid products in a set theoretic way. We have proposed two level of generalization for CFDs. In the first generalization, we have relaxed some constraints on group cardinalities. We believe that this very simple generalization provides us much more succinct and expressive tool for system modeling. The second generalization (emerged in the regular expression translation procedure) is called cardinality-based regular expression diagrams (CRDs), in which the labels of nodes can be any regular expression built over the set of features. We believe that CRDs is a mean moving us to modeling much more complicated systems, in which we need to deal with structural (non-atomic) features, e.g., programming codes, etc.

We have provided a reduction process, which allows us to go from a CFD to an RE. The procedure works for CRDs. The generative expression for a given CFD has two main properties: it captures the hierarchical structure of the CFD; it also captures the product line of the CFD. These properties enable us to confidently claim that this translation faithfully captures the semantics of CFDs.

Regular languages have some nice computational properties. These properties, such as the decidability of emptiness, inclusion, and equality problems, help us to propose algorithmic solutions for analysis operations over CFDs. In addition, the complexity class of all regular languages is SPACE(O(1)), i.e., the decision problems can be solved in constant space. Due to these nice computational properties, we can also claim that regular expressions provide a nice computable framework for reasoning about CFDs.

As for CCs, we have proposed a formal language interpretation of them. In this way, we could integrate the formal semantics of CFDs and CCs. Also, it allows us to group CFMs based on their semantics, which guides us how they can be constructively analyzed.

Based on this formal language interpretation of CFMs, we have provided two kinds of semantics, called dynamic and static. The dynamic semantics of a given model is equal to the FL of the whole model. The dynamic semantics of CFMs is a new concept, but the static one is, indeed, equivalent to the semantics captured in [8].

We also have characterized some existing analysis operations over CFMs in terms of on the FL framework. This allows us to use some off-the-shelf language tools, such as JFLAP [33], to do analysis on CFMs. Note that automated support for analysis over CFMs were always a challenging issue. We also have investigated the decidability problems of the introduced analysis operations for different kinds of CFMs. We noticed that some analysis operations are not decidable in all classes of CFMs.

10 Open Problems/Future Work

Based on the closure properties of regular languages, say closure under intersection, union, complement, etc., we believe that our framework is a very good candidate for managing multiple product lines [2]. Indeed, in a forthcoming paper, we will discuss how to manage FDs using the FL-framework.

The computational complexity problem of analysis operations would be a crucial issue in implementing them for CFMs, which needs to be investigated.

In the literature, the object-constraint language (OCL) has been proposed for expressing CCs in CFMs [9]. Our next mission is to discover the OCL-definable languages. It can be also fruitful for the model driven engineering (MDE) area, since the MDE community uses mainly OCL to express constraints. This way, we can investigate the expressiveness of OCL in terms of languages. Our conjecture is that there should be some practical CCs that cannot be expressed in OCL. Bellow, we provide some hints to support our conjectures.

It is well-known *conjecture* that, theoretically, OCL is first order logic (FOL) plus transitivity and counting. FOL-definability leads to the class of star-free regular languages [14]. Considering transitivity, the class of OCL-definable languages would be equal to the class of regular languages. Considering the counting operation and equality, some context-free and sensitive languages are also covered. However, not all context-free languages can be expressed using only counting and equality. All the above conjectures need to be investigated theoretically.

References

1. Parosh Aziz Abdulla, Yu-Fang Chen, Lorenzo Clemente, Lukáš Holík, Chih-Duo Hong, Richard Mayr, and Tomáš Vojnar. Advanced ramsey-based büchi automata inclusion testing. In *CONCUR 2011–Concurrency Theory*, pages 187–202. Springer, 2011.
2. Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert France. Managing multiple software product lines using merging techniques. *France: UniversityofNiceSophiaAntipolis. TechnicalReport, ISRN I3S/RR*, 6, 2010.
3. Don Batory. *Feature models, grammars, and propositional formulas.* Springer, 2005.
4. David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.
5. Filippo Bonchi and Damien Pous. Checking nfa equivalence with bisimulations up to congruence. *ACM SIGPLAN Notices*, 48(1):457–468, 2013.
6. S Barry Cooper. *Computability theory.* CRC Press, 2003.
7. Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich Eisenecker. Generative programming for embedded software: An industrial experience report. In *Generative Programming and Component Engineering*, pages 156–172. Springer, 2002.
8. Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.
9. Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints: A progress report. In *International Workshop on Software Factories*, pages 16–20, 2005.
10. Krzysztof Czarnecki and Andrzej Wasowski. Feature diagrams and logics: There and back again. In *Software Product Line Conference, 2007. SPLC 2007. 11th International*, pages 23–34. IEEE, 2007.
11. Martin Davis. *Computability, complexity, and languages: fundamentals of theoretical computer science.* Academic Press, 1994.
12. Merijn de Jonge and Joost Visser. Grammars as feature diagrams. In *ICSR7 Workshop on Generative Programming*, pages 23–24. Citeseer, 2002.
13. Martin De Wulf, Laurent Doyen, Nicolas Maquet, and Jean-François Raskin. Alaska. In *Automated Technology for Verification and Analysis*, pages 240–245. Springer, 2008.

14. Volker Diekert and Paul Gastin. First-order definable languages. *Logic and Automata: History and Perspectives*, 2:261, 2008.
15. Zinovy Diskin, Aliakbar Safilian, Tom Maibaum, and Shoham Ben-David. Modeling product lines with kripke structures and modal logic. (GSDLab TR 2014-08-01), 08/2014 2014.
16. Seymour Ginsburg. *The Mathematical Theory of Context Free Languages.[Mit Fig.]*. McGraw-Hill Book Company, 1966.
17. Sandra Hilber. Finite automata tool, http://cl-informatik.uibk.ac.at/software/fat/. 2009.
18. Peter Höfner, Ridha Khédri, and Bernhard Möller. An algebra of product families. *Software and System Modeling*, 10(2):161–182, 2011.
19. Peter Höfner, Ridha Khédri, and Bernhard Möller. Supplementing product families with behaviour. *Int. J. Software and Informatics*, 5(1-2):245–266, 2011.
20. John E. Hopcroft. On the equivalence and containment problems for context-free languages. *Mathematical systems theory*, 3(2):119–124, 1969.
21. John E Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Addison Wesley, 2007.
22. Alexander Koller, Michaela Regneri, and Stefan Thater. Regular tree grammars as a formalism for scope underspecification. In *ACL*, pages 218–226. Citeseer, 2008.
23. Dexter Kozen. *Automata and computability*. Springer, 1997.
24. Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. Vata: A library for efficient manipulation of non-deterministic tree automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 79–94. Springer, 2012.
25. Peter Linz. *An introduction to formal languages and automata*. Jones & Bartlett Publishers, 2011.
26. Mike Mannion. Using first-order logic for product line model validation. In *Software Product Lines*, pages 176–187. Springer, 2002.
27. Marcilio Mendonca, Andrzej Wasowski, Krzysztof Czarnecki, and Donald Cowan. Efficient compilation techniques for large scale feature models. In *Proceedings of the 7th international conference on Generative PROGRAMMING and component engineering*, pages 13–22. ACM, 2008.
28. Albert R Meyer and Larry J Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Switching and Automata Theory, 1972., IEEE Conference Record of 13th Annual Symposium on*, pages 125–129. IEEE, 1972.
29. Rohit J Parikh. On context-free languages. *Journal of the ACM (JACM)*, 13(4):570–581, 1966.
30. Klaus Pohl, Günter Böckle, and Frank Van Der Linden. *Software product line engineering: foundations, principles, and techniques*. Springer, 2005.
31. Clément Quinton, Daniel Romero, and Laurence Duchien. Cardinality-based feature models with constraints: a pragmatic approach. In *Proceedings of the 17th International Software Product Line Conference*, pages 162–166. ACM, 2013.
32. Susan H Rodger and Thomas W Finley. *JFLAP: an interactive formal languages and automata package*. Jones & Bartlett Learning, 2006.
33. Susan H Rodger and Thomas W Finley. *JFLAP: an interactive formal languages and automata package*. Jones & Bartlett Learning, 2006.
34. Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007.

35. Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. Reverse engineering feature models. In *ICSE 2011*, pages 461–470. IEEE, 2011.
36. Yih-Kuen Tsay, Yu-Fang Chen, Ming-Hsien Tsai, Wen-Chin Chan, and Chi-Jian Luo. Goal extended: Towards a research tool for omega automata and temporal logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 346–350. Springer, 2008.
37. Arie Van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.
38. Gertjan van Noord. Fsa6. 2xx: Finite state automata utilities. *http://odur. let. rug. nl/vannoord/Fsa/fsa. html, accessed*, 3(10):2003, 2002.