A Formal Semantics for Multi-level Staged Configuration

Classen, Andreas; Hubaux, Arnaud; Heymans, Patrick

Published in:
Proceedings of the Third Workshop on Variability Modelling of Software-intensive Systems (VaMoS'09)

Publication date:
2009

Document Version
Early version, also known as pre-print

PReCISE – FUNDP
University of Namur
Rue Grandgagnage, 21
B-5000 Namur
Belgium

TECHNICAL REPORT	January 7, 2009
AUTHORS	A. Classen, A. Hubaux, P. Heymans
APPROVED BY	P. Heymans
EMAILS	{acs,ahu,phe}@info.fundp.ac.be
STATUS	Extended version of a paper submitted to the *Third International Workshop on Variability Modelling of Software-intensive Systems* (VaMoS'09), Sevilla, Spain.
REFERENCE	P-CS-TR SPLBT-00000002
PROJECT	MoVES
FUNDING	FNRS, Interuniversity Attraction Poles Programme of the Belgian State of Belgian Science Policy

A Formal Semantics for Multi-level Staged Configuration

A Formal Semantics for Multi-level Staged Configuration

Andreas Classen,* Arnaud Hubaux and Patrick Heymans
PReCISE Research Centre,
Faculty of Computer Science,
University of Namur
5000 Namur, Belgium

E-mail: {acs,ahu,phe}@info.fundp.ac.be

Abstract

Multi-level staged configuration (MLSC) of feature diagrams has been proposed as a means to facilitate configuration in software product line engineering. Based on the observation that configuration often is a lengthy undertaking with many participants, MLSC splits it up into different levels that can be assigned to different stakeholders. This makes configuration more scalable to realistic environments. Although its supporting language (cardinality based feature diagrams) received various formal semantics, the MLSC process never received one. Nonetheless, a formal semantics is the primary indicator for precision and unambiguity and an important prerequisite for reliable tool-support.

We present a semantics for MLSC that builds on our earlier work on formal feature model semantics to which it adds the concepts of level *and* configuration path*. With the formal semantics, we were able to make the original definition more precise and to reveal some of its subtleties and incompletenesses. We also discovered some important properties that an MLSC process should possess and a configuration tool should guarantee. Our contribution is primarily of a fundamental nature, clarifying central, yet ambiguous, concepts and properties related to MLSC. Thereby, we intend to pave the way for safer, more efficient and more comprehensive automation of configuration tasks.*

1 Introduction

Feature Diagrams (FDs) are a common means to represent, and reason about, variability during Software Product Line (SPL) Engineering (SPLE) [18]. In this context, they have proved to be useful for a variety of tasks such as project scoping, requirements engineering and product configuration, and in a number of application domains such as telecoms, automotive and home automation systems [4, 16, 24, 6, 18, 3].

The core purpose of an FD is to define concisely the set of legal *configurations* – generally called *products* – of some (usually software) artefact. An example FD is shown in Figure 1. Basically, FDs are trees[1] whose nodes denote features and whose edges represent top-down hierarchical decomposition of features. Each decomposition tells that, given the presence of the parent feature in some configuration c, some combination of its children should be present in c, too. Which combinations are allowed depends on the type of the decomposition, that is, the Boolean operator associated to the parent. In addition to their tree-shaped backbone, FDs can also contain cross-cutting constraints (usually *requires* or *excludes*) as well as side constraints in a textual language such as propositional logic [2].

Given an FD, the *configuration* or *product derivation process* is the process of gradually making the choices defined in the FD with the purpose of determining the product that is going to be built. In a realistic development, the configuration process is a small project itself, involving many people and taking up to several months [19]. In order to master the complexity of the configuration process, Czarnecki *et al.* [8] proposed the concept of *multi-level staged configuration* (MLSC), in which configuration is carried out by different stakeholders at different levels of product development or customisation. In simple staged configuration, at each stage some variability is removed from the FD until none is left. MLSC generalises this idea to the case were a set of related FDs are configured, each FD per-

*FNRS Research Fellow.

[1] Sometimes DAGs are used, too [14].

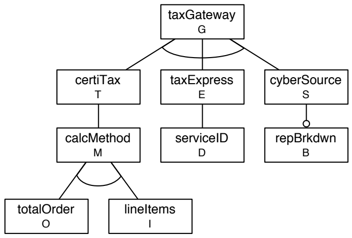
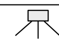
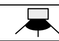
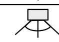
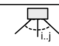


Figure 1. FD example, adapted from [8].

Table 1. FD decomposition operators

Concrete syntax	Boolen operator	Cardinality
	$and : \wedge$	$\langle n..n \rangle$
	$or : \vee$	$\langle 1..n \rangle$
	$xor : \oplus$	$\langle 1..1 \rangle$
		$\langle i..j \rangle$

taining to a so-called 'level'. This addresses problems that occur when different abstraction levels are present in the same FD and also allows for more realism since a realistic project would have several related FDs rather than a single big one [20, 19].

Even though its supporting language (cardinality based FDs) received various formal semantics [7, 21], the MLSC process never received one. Nonetheless, a formal semantics is the primary indicator for precision and unambiguity and an important prerequisite for reliable tool-support. This paper is intended to fill this gap with a semantics for MLSC that builds on our earlier work on formal semantics for FDs [21]. The earlier semantics of [21] will be herein referred to as *static*, because it concentrates on telling which configurations are allowed (and which are disallowed), regardless of the process to be followed for reaching one or the other configuration. We thus extend this semantics with the concepts of *stage*, *configuration path* and *level*.

The contribution of the paper is a precise and formal account of MLSC that makes the original definition [8] more explicit and reveals some of its subtleties and incompletenesses. The semantics also allowed us to discover some important properties that an MLSC process should possess and a configuration tool should guarantee.

The paper is structured as follows. Section 2 recalls the static FD semantics and introduces a running example. Section 3 recapitulates the main concepts of staged configuration which are then formalised in Section 4 with the introduction of the dynamic semantics. Ways to implement and otherwise use the semantics are discussed in Section 5 while extensions are proposed in Section 6. The paper will be concluded in Section 7.

2 Static FD semantics ($[\![.]\!]_{FD}$)

In [21], we gave a general formal semantics to a wide range of FD dialects. The full details of the formalisation cannot be reproduced here, but we need to recall the essentials.[2] The formalisation was performed following the guidelines of Harel and Rumpe [12], according to whom each modelling language L must possess an unambiguous mathematical definition of three distinct elements: the *syntactic domain* $\mathcal{L}_L$, the *semantic domain* $\mathcal{S}_L$ and the *semantic function* $\mathcal{M}_L : \mathcal{L}_L \to \mathcal{S}_L$, also traditionally written $[\![\cdot]\!]_L$.

Our FD language will be simply called FD, and its syntactic domain is defined as follows.

Definition 1 (Syntactic domain $\mathcal{L}_{FD}$) *$d \in \mathcal{L}_{FD}$ is a 6-tuple $(N, P, r, \lambda, DE, \Phi)$ such that:*

- *N is the (non empty) set of features (nodes).*
- *$P \subseteq N$ is the set of primitive features.*
- *$r \in N$ is the root.*
- *$DE \subseteq N \times N$ is the decomposition relation between features which forms a tree. For convenience, we will use $children(f)$ to denote $\{g \mid (f, g) \in DE\}$, the set of all direct sub-features of f, and write $n \to n'$ sometimes instead of $(n, n') \in DE$.*
- *$\lambda : N \to \mathbb{N} \times \mathbb{N}$ indicates the decomposition type of a feature, represented as a cardinality $\langle i..j \rangle$ where i indicates the minimum number of children required in a product and j the maximum. For convenience, special cardinalities are indicated by the Boolean operator they represent, as shown in Table 1.*
- *Φ is a formula that captures crosscutting constraints ($\ll requires \gg$ and $\ll includes \gg$) as well as textual constraints. Without loss of generality, we consider Φ to be a conjunction of Boolean formulae on features, i.e. $\Phi \in \mathbb{B}(N)$, a language that we know is expressively complete wrt. $\mathcal{S}_{FD}$ [22].*

Furthermore, each $d \in \mathcal{L}_{FD}$ must satisfy the following well-formedness rules:

- *r is the root: $\forall n \in N (\nexists n' \in N \bullet n' \to n) \Leftrightarrow n = r$,*
- *DE is acyclic: $\nexists n_1, .., n_k \in N \bullet n_1 \to .. \to n_k \to n_1$,*
- *Terminal nodes are $\langle 0..0 \rangle$-decomposed.*

Definition 1 is actually a formal definition of the graphical syntax of an FD such as the one shown in Figure 1;

[2] Some harmless simplifications are made wrt. the original [21].

for convenience, each feature is given a name and a one-letter acronym. The latter depicts an FD for the tax gateway component of an e-Commerce system [8]. The component performs the calculation of taxes on orders made with the system. The customer who is going to buy such a system has the choice of three tax gateways, each offering a distinct functionality. Note that the hollow circle above feature B is syntactic sugar, expressing the fact that the feature is optional. In $\mathcal{L}_{FD}$, an optional feature f is encoded with a dummy (i.e. non-primitive) feature d that is $\langle 0..1\rangle$-decomposed and having f as its only child [21]. Let us call B_d the dummy node inserted between B and its parent. The diagram itself can be represented as an element of $\mathcal{L}_{FD}$ where $N = \{G, T, E, ...\}$, $P = N \setminus \{B_d\}$, $r = G$, $E = \{(G, T), (G, E), ...\}$, $\lambda(G) = \langle 1..1\rangle$, ... and $\Phi = \emptyset$.

The semantic domain formalises the real-world concepts that the language models, and that the semantic function associates to each diagram. FDs represent SPLs, hence the following two definitions.

Definition 2 (Semantic domain $\mathcal{S}_{FD}$) *$\mathcal{S}_{FD} \triangleq \mathcal{PP}P$, indicating that each syntactically correct diagram should be interpreted as a product line, i.e. a set of* configurations *or* products *(set of sets of primitive features).*

Definition 3 (Semantic function $[\![d]\!]_{FD}$) *Given $d \in \mathcal{L}_{FD}$, $[\![d]\!]_{FD}$ returns the* valid feature combinations *$FC \in \mathcal{PP}N$ restricted to primitive features: $[\![d]\!]_{FD} = FC_{|P}$, where the valid feature combinations FC of d are those $c \in \mathcal{P}N$ that:*

- *contain the root: $r \in c$,*
- *satisfy the decomposition type: $f \in c \wedge \lambda(f) = \langle m..n\rangle \Rightarrow m \leq |children(f) \cap c| \leq n$,*
- *justify each feature: $g \in c \wedge g \in children(f) \Rightarrow f \in c$,*
- *satisfy the additional constraints: $c \models \Phi$.*

The reduction operator used in Definition 3 will be used throughout the paper; it is defined as follows.

Definition 4 (Reduction $A_{|B}$)

$$A_{|B} \triangleq \{a' | a \in A \wedge a' = a \cap B\} = \{a \cap B | a \in A\}$$

Considering the previous example, the semantic function maps the diagram of Figure 1 to all its valid feature combinations, i.e. $\big\{\{G, T, M, O\}, \{G, T, M, I\}, ...\big\}$.

As shown in [21], this language suffices to retrospectively define the semantics of most common FD languages. The language for which staged configuration was initially defined [8], however, cannot entirely be captured by the above semantics [22]. The concepts of *feature attribute*, *feature reference* and *feature cardinality*[3] are missing. Attributes can easily be added to the semantics [7], an exercise we leave for future work. Feature cardinalities, as used for the *cloning* of features, however, would require a major revision of the semantics [7].

Benefits, limitations and applications of the above semantics have been discussed extensively elsewhere [21]. We just recall here that its main advantages are the fact that it gives an unambiguous meaning to each FD, and makes FDs amenable to automated treatment. The benefit of defining a semantics before building a tool is the ability to reason about tasks the tool should do on a pure mathematical level, without having to worry about their implementation. These so-called decision problems are mathematical properties defined on the semantics that can serve as indicators, validity or satisfiability checks.

In the present case, for instance, an important property of an FD, its *satisfiability* (i.e. whether it admits at least one product), can be mathematically defined as $[\![d]\!]_{FD} \neq \emptyset$. Another property is product inclusion, i.e. given a product, checking whether it is part of the FD. Yet another is to determine whether there are features that will never be part of a product, i.e. dead features: $P \setminus \bigcup [\![d]\!]_{FD} \neq \emptyset$. As we will see later on, the lack of formal semantics for staged configuration makes it difficult to precisely define such properties.

For the remainder of the paper, unless otherwise stated, we always assume d to denote an FD, and (N, P, r, λ, DE, Φ) to denote the respective elements of its abstract syntax.

3 Multi-level staged configuration

According to the semantics introduced in the previous section, an FD basically describes which configurations are allowed in the SPL, regardless of the *configuration process* to be followed for reaching one or the other configuration. Still, such a process is an integral part of SPL application engineering. According to Rabiser *et al.* [19], for instance, the configuration process generally involves many people and may take up to several months.

Czarnecki *et al.* acknowledge the need for explicit process support, arguing that in contexts such as *"software supply chains, optimisation and policy standards"*, the configuration is carried out in *stages* [8]. According to the same authors, a stage can be defined *"in terms of different dimensions: phases of the product lifecycle, roles played by participants or target subsystems"*. In an effort to make this explicit, they propose the concept of *multi-level staged configuration* (MLSC).

[3] Czarnecki *et al.* [8] distinguish *group* and *feature cardinalities*. Group cardinalities immediately translate to our decomposition types and $\langle 0..1\rangle$ feature cardinalities to optional features. The $\langle i..k\rangle$ feature cardinalities, with $i \geq 0$ and $k > 1$, however, cannot be encoded in $\mathcal{L}_{FD}$.

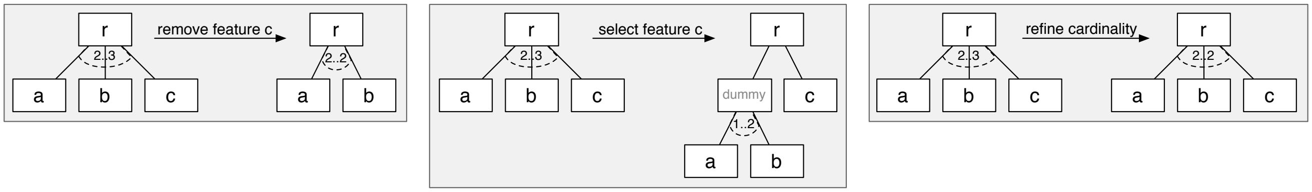


Figure 2. Specialisation steps, adapted from [8].

The principle of staged configuration is to remove part of the variability at each stage until only one configuration, the final product, remains. In [8], the refinement itself is achieved by applying a series of syntactic transformations to the FD. Some of these transformations, such as setting the value of an attribute, involve constructs that are not formalised as part of the semantics defined in Section 2. The remaining transformations are show in Figure 2. Note that they are expressed so that they conform to our semantics.

Multi-level staged configuration is the application of this idea to a series of related FDs $d_1, .., d_\ell$. Each level has its own FD, and, depending on how they are linked, the configuration of one level will induce an automatic specialisation of the next level's FD. The links between diagrams are defined explicitly through *specialisation annotations*. A specialisation annotation of a feature f in d_i, ($f \in N_i$), consists of a Boolean formulae ϕ over the features of $d_{i-1} (\phi \in \mathbb{B}(N_{i-1}))$. Once level $i-1$ is configured, ϕ can be evaluated on the obtained configuration $c \in [\![d_{i-1}]\!]_{FD}$, using the now standard Boolean encoding of [2], i.e. a feature variable n in ϕ is $true$ iff $n \in c$. Depending on its value and the specialisation type, the feature f will either be removed or selected through one of the first two syntactic transformations of Figure 2. An overview of this is shown in Table 2.

Let us illustrate this on the example of the previous section: imagine that there are two times at which the customer needs to decide about the gateways. The first time (level one) is when he purchases the system. All he decides at this point is which gateways will be available for use; the diagram that needs to be configured is the one shown on the left of Figure 3. Then, when the system is being deployed (level two), he will have to settle for one of the gateways and provide additional configuration parameters, captured by the first diagram on the right side of Figure 3. Given the inter-level links, the diagram in level two is automatically specialised based on the choices made in level one.

Note that even though both diagrams in the example are very similar, they need not be so. Also note that the original paper mentions the possibility, that several configuration levels might run in parallel. It applies, for instance, if levels represent independent decisions that need to be taken by different people. As we show later on, such situations give rise to interesting decision problems.

Finally, note that the MLSC approach, as it appears in [8], is entirely based on *syntactic* transformations. This makes it difficult to decide things such as whether two levels A and B are commutative (executing A before B leaves the same variability as executing B before A). This is the main motivation for defining a formal semantics, as follows in the next section.

4 Dynamic FD semantics ($[\![.]\!]_{CP}$)

We introduce the dynamic FD semantics in two steps. The first, Section 4.1, defines the basic staged configuration semantics; the second, Section 4.2, adds the multi-level aspect.

4.1 Staged configuration semantics

Since we first want to model the different stages of the configuration process, regardless of levels, the syntactic domain $\mathcal{L}_{FD}$ will remain as defined in Section 2. The semantic domain, however, changes since we want to capture the idea of building a product by deciding incrementally which configuration to retain and which to exclude.

Indeed, we consider the semantic domain to be the set of all possible *configuration paths* that can be taken when building a configuration. Along each such path, the initially full *configuration space* ($[\![d]\!]_{FD}$) progressively shrinks (i.e., configurations are discarded) until only one configuration is left, at which point the path stops. Note that in this work, we thus assume that we are dealing with *finite* configuration processes where, once a unique configuration is reached, it remains the same for the rest of the life of the application. Extensions of this semantics, that deal with reconfigurable systems, are discussed in Section 6. For now, we stick to Definitions 5 and 7 that formalise the intuition we just gave.

Definition 5 (Dynamic semantic domain $\mathcal{S}_{CP}$) *Given a finite set of features N, a* configuration path π *is a finite sequence $\pi = \sigma_1 ... \sigma_n$ of length $n > 0$, where each $\sigma_i \in \mathcal{PP}N$ is called a* stage*. If we call the set of such paths C, then $\mathcal{S}_{CP} = \mathcal{P}C$.*

Table 2. Possible inter-level links; original definition [8] left, translation to FD semantics right.

Specialisation type	Condition value	Specialisation operation	Equivalent Boolean constraint with $f \in N_i, \phi \in \mathbb{B}(N_{i-1}), c \in [\![d_{i-1}]\!]_{FD}$	
$positive$	$true$	select	$\phi(c) \Rightarrow f$	*Select* f, i.e. Φ_i becomes $\Phi_i \cup \{f\}$, if $\phi(c)$ is true.
$positive$	$false$	none		
$negative$	$false$	remove	$\neg\phi(c) \Rightarrow \neg f$	*Remove* f, i.e. Φ_i becomes $\Phi_i \cup \{\neg f\}$, if $\phi(c)$ is false.
$negative$	$true$	none		
$complete$	$true$	select	$\phi(c) \Leftrightarrow f$	*Select or remove* f depending on the value of $\phi(c)$.
$complete$	$false$	remove		

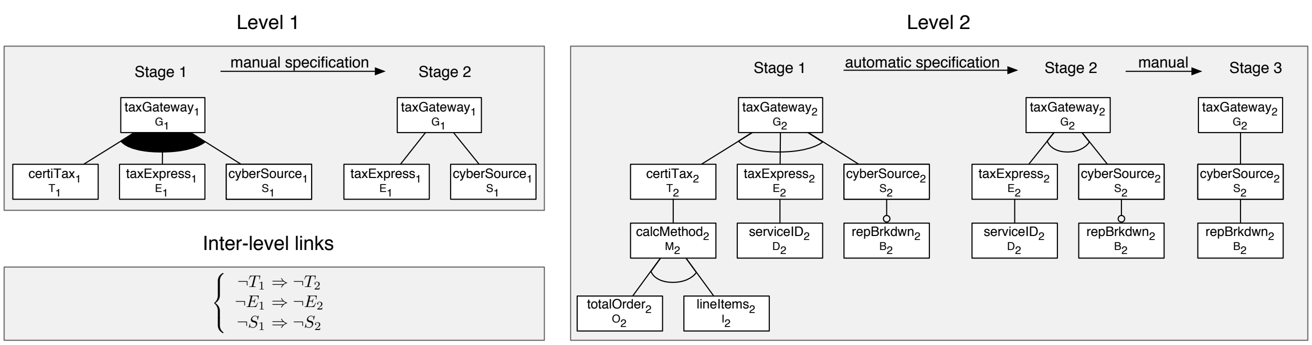


Figure 3. Example of MLSC, adapted from [8].

The following definition will be convenient when expressing properties of configuration paths.

Definition 6 (Path notation and helpers)

- *ϵ denotes the empty sequence*
- $last(\sigma_1 ... \sigma_k) = \sigma_k$

Definition 7 (Staged configuration semantics $[\![d]\!]_{CP}$) *Given an FD $d \in \mathcal{L}_{FD}$, $[\![d]\!]_{CP}$ returns all* legal *paths π (noted $\pi \in [\![d]\!]_{CP}$, or $\pi \models_{CP} d$) such that*

(7.1) $\sigma_1 = [\![d]\!]_{FD}$

(7.2) $\forall i \in \{2..n\} \bullet \sigma_i \subset \sigma_{i-1}$

(7.3) $|\sigma_n| = 1$

Note that this semantics is not meant to be used as an implementation directly, for it would be very inefficient. This is usual for denotational semantics which are essentially meant to serve as a conceptual foundation and a reference for checking the conformance of tools [23]. Along these lines, we draw the reader's attention to condition (7.2) which will force compliant configuration tools to let users make only "useful" configuration choices, that is, choices that effectively eliminate configurations. At the same time, tools must ensure that a legal product eventually remains reachable given the choices made, as requested by condition (7.3).

As an illustration, Figure 4 shows an example FD and its legal paths. A number of properties can be derived from the above definitions.

Theorem 8 (Properties of configuration paths)

(8.1) $[\![d]\!]_{FD} = \emptyset \Leftrightarrow [\![d]\!]_{CP} = \emptyset$

(8.2) $\forall c \in [\![d]\!]_{FD} \bullet \exists \pi \in [\![d]\!]_{CP} \bullet last(\pi) = \{c\}$

(8.3) $\forall \pi \in [\![d]\!]_{CP} \bullet \exists c \in [\![d]\!]_{FD} \bullet last(\pi) = \{c\}$

Contrary to what intuition might suggest, (8.2) and (8.3) do not imply that $|[\![d]\!]_{FD}| = |[\![d]\!]_{CP}|$, they merely say that every configuration allowed by the FD can be reached as part of a configuration path, and that each configuration path ends with a configuration allowed by the FD.

Czarnecki *et al.* [8] define a number of transformation rules that are to be used when specialising an FD, three of which are shown in Figure 2. With the formal semantics, we can now verify whether these rules are expressively complete, i.e. whether is it always possible to express a σ_i

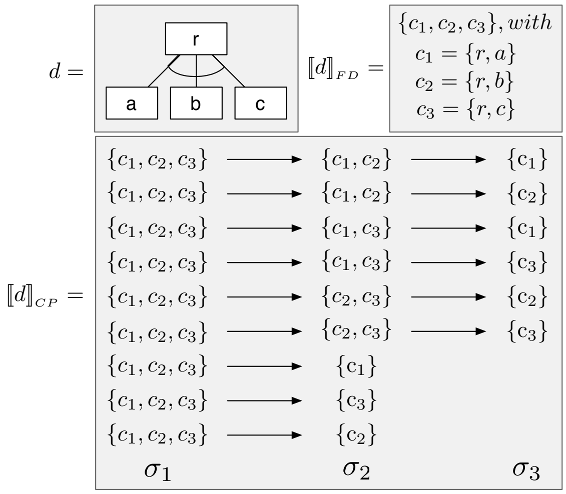


Figure 4. The staged configuration semantics illustrated.

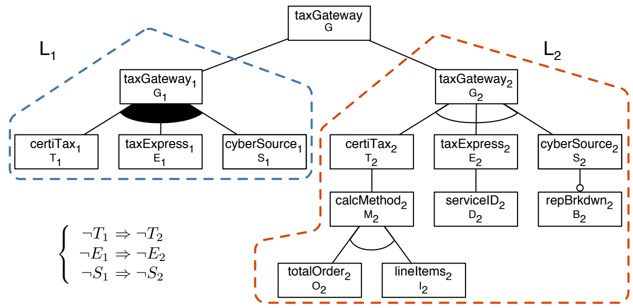


Figure 5. Example of Figure 3 in $\mathcal{L}_{DynFD}$.

($i > 1$) through the application of the three transformation rules.

Theorem 9 (Incompleteness of transformation rules) *The transformation rules shown in Figure 2 are expressively incomplete wrt. the semantics of Definition 7.*

Proof. Consider a diagram consisting of a parent feature $\langle 2..2\rangle$-decomposed with three children a, b, c. It is not possible to express the σ_i consisting of $\{a, b\}$ and $\{b, c\}$, by starting at $\sigma_1 = \big\{\{a, b\}, \{a, c\}, \{b, c\}\big\}$ and using the proposed transformation rules (since removing one feature will always result in removing at least two configurations). $\square$

Note that this is not necessarily a bad thing, since Czarnecki *et al.* probably chose to only include transformation steps that implement the most frequent usages. However, the practical consequences of this limitation need to be assessed empirically.

4.2 Adding levels

Section 4.1 only deals with dynamic aspects of staged configuration of a single diagram. If we want to generalise this to MLSC, we need to consider multiple diagrams and links between them. To do so, there are two possibilities: (1) define a new abstract syntax, that makes the set of diagrams and the links between them explicit, or (2) encode this information using the syntax we already have.

We chose the latter option, mainly because it allows to reuse most of the existing definitions and infrastructure, and because it can more easily be generalised. Indeed, a set of FDs, linked with conditions of the types defined in Table 2, can be represented as a single big FD. The root of each individual FD becomes a child of the root of the combined FD. The root is and-decomposed and the inter-level links are represented by Boolean formulae. To keep track of where the features in the combined FD came from, the level information will be made explicit as follows.

Definition 10 (Dynamic syntactic domain $\mathcal{L}_{DynFD}$) *$\mathcal{L}_{DynFD}$ consists of 7-tuples $(N, P, L, r, \lambda, DE, \Phi)$, where:*

- *$N, P, r, \lambda, DE, \Phi$ follow Definition 1,*
- *$L = L_1 ... L_\ell$ is a partition of $N \setminus \{r\}$ representing the list of levels.*

So that each $d \in \mathcal{L}_{DynFD}$ satisfies the well-formedness rules of Definition 1, has an and-decomposed root, and each level $L_i \in L$:

- *is connected through exactly one node to the global root: $\exists! n \in L_i \bullet (r, n) \in DE$, noted hereafter $root(L_i)$,*
- *does not share decomposition edges with other levels (except for the root): $\forall (n, n') \in DE \bullet (n \in L_i \Leftrightarrow n' \in L_i) \vee (n = r \wedge n' = root(L_i))$,*
- *is itself a valid FD, i.e. $(L_i,\ P \cap L_i,\ root(L_i),\ \lambda \cap (L_i \to \mathbb{N} \times \mathbb{N}),\ DE \cap (L_i \times L_i),\ \emptyset)$ satisfies Definition 1.*[4]

Figure 5 illustrates how the example of Figure 3 is represented in $\mathcal{L}_{DynFD}$. Note that, for the purpose of this paper, we chose an arbitrary concrete syntax for expressing levels, viz. the dotted lines. This is meant to be illustrative, since a tool implementation should rather present each level separately, so as to not harm scalability.

Given the new syntactic domain, we need to revise the semantic function. As for the semantic domain, it can remain the same, since we still want to reason about the possible configuration paths of an FD. The addition of multiple

[4] The set of constraints here is empty because it is not needed for validity verification.

levels, however, requires us to reconsider what a *legal* configuration path is. Indeed, we want to restrict the configuration paths to those that obey the levels specified in the FD. Formally, this is defined as follows.

Definition 11 (Dynamic FD semantics $[\![d]\!]_{DynFD}$) *Given an FD $d \in \mathcal{L}_{DynFD}$, $[\![d]\!]_{DynFD}$ returns all paths π that are legal wrt. Definition 7, i.e. $\pi \in [\![d]\!]_{CP}$, and for which exists a* legal level arrangement*, that is π, except for its initial stage, can be divided into ℓ $(= |L|)$* levels*: $\pi = \sigma_1\Sigma_1..\Sigma_\ell$, each Σ_i corresponding to an L_i such that:*

(11.1) Σ_i is fully configured: $|final(\Sigma_i)_{|L_i}| = 1$, and

(11.2) $\forall \sigma_j\sigma_{j+1} \bullet \pi = ...\sigma_j\sigma_{j+1}...$ and $\sigma_{j+1} \in \Sigma_i$, we have

$$(\sigma_j \setminus \sigma_{j+1})_{|L_i} \subseteq (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}).$$

As before, this will be noted $\pi \in [\![d]\!]_{DynFD}$, or $\pi \models_{DynFD} d$.

We made use of the following helper.

Definition 12 (Final stage of a level Σ_i) *For $i = 1..\ell$,*

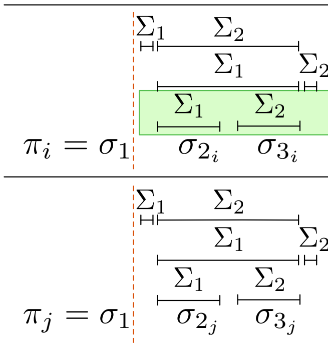
$$final(\Sigma_i) \triangleq \begin{cases} last(\Sigma_i) & \text{if } \Sigma_i \neq \epsilon \\ final(\Sigma_{i-1}) & \text{if } \Sigma_i = \epsilon \text{ and } i > 1 \\ \sigma_1 & \text{if } \Sigma_i = \epsilon \text{ and } i = 1 \end{cases}$$

The rule (11.2) expresses the fact that each configuration deleted from σ_j (i.e. $c \in \sigma_j \setminus \sigma_{j+1}$) during level L_i must be necessary to delete one of the configurations of L_i that are deleted during this stage. In other words, the set of *deleted* configurations needs to be included in the set of *deletable* configurations for that level. The deletable configurations in a stage of a level are those that indeed remove configurations pertaining to that level (hence: first reduce to the level, then subtract), whereas the deleted configurations in a stage of a level are all those that were removed (hence: first subtract, then reduce to level to make comparable). Intuitively, this corresponds to the fact that each decision has to affect only the level at which it is taken.

4.3 Illustration

Let us illustrate this with the FD of Figure 5, which we will call d, itself being based on the example of Figure 3 in Section 3. The semantic domain of $[\![d]\!]_{DynFD}$ still consists of configuration paths, i.e. it did not change from those of $[\![d]\!]_{CP}$ shown in Figure 4. Yet, given that $[\![d]\!]_{DynFD}$ takes into account the levels defined for d, not all possible configuration paths given by $[\![d]\!]_{CP}$ are legal. Namely, those that do not conform to rules (11.1) and (11.2) need to be discarded. This is depicted in Figure 6, where the upper box denotes the staged configuration semantics of d ($[\![d]\!]_{CP}$), and the lower box denotes $[\![d]\!]_{DynFD}$, i.e. the subset of $[\![d]\!]_{CP}$ that conforms to Definition 11.

Table 3. Validation of level arrangements.



Level arrangement for path	rule (11.1)	rule (11.2)
π_i, arrangement 1	FALSE	/
π_i, arrangement 2	TRUE	FALSE
π_i, arrangement 3	TRUE	TRUE
π_j, arrangement 1	FALSE	/
π_j, arrangement 2	TRUE	FALSE
π_j, arrangement 3	TRUE	FALSE

We now zoom in on two configuration paths $\pi_i, \pi_j \in [\![d]\!]_{CP}$, shown with the help of intermediate FDs in the lower part of Figure 6. As noted in Figure 6, π_j is not part of $[\![d]\!]_{DynFD}$ since it violates Definition 11, whereas π_i satisfies it and is kept. The rationale for this is provided in Table 3. Indeed, for π_j, there exists no level arrangement that would satisfy both rules (11.1) and (11.2). This is because in σ_{2_j}, it is not allowed to remove the feature B_2, since it belongs to L_2, and L_1 is not yet completed. Therefore, either there is still some variability left in the FD at the end of the level, which is thus not fully configured (the first possible arrangement of π_j in Table 3 violates rule (11.1)), or the set of deleted configurations is greater than the set of deletable configurations (the other two arrangements of π_j in Table 3, which violate rule (11.2)). For π_i, on the other hand, a valid level arrangement exists and is indicated by the highlighted line in Table 3. All details for this illustration are provided in Appendix A.

5 Towards automation and analysis

This section explores properties of the semantics we just defined, sketches paths towards automation and examines how it can be used for analysis.

5.1 Properties of the semantics

In Definition 11, we require that it has to be possible to divide a configuration path into level arrangements that satisfy certain properties. The definition being purely declarative, it does not allow an immediate conclusion as to how many valid level arrangements one might find. The following two theorems show that there is exactly one.

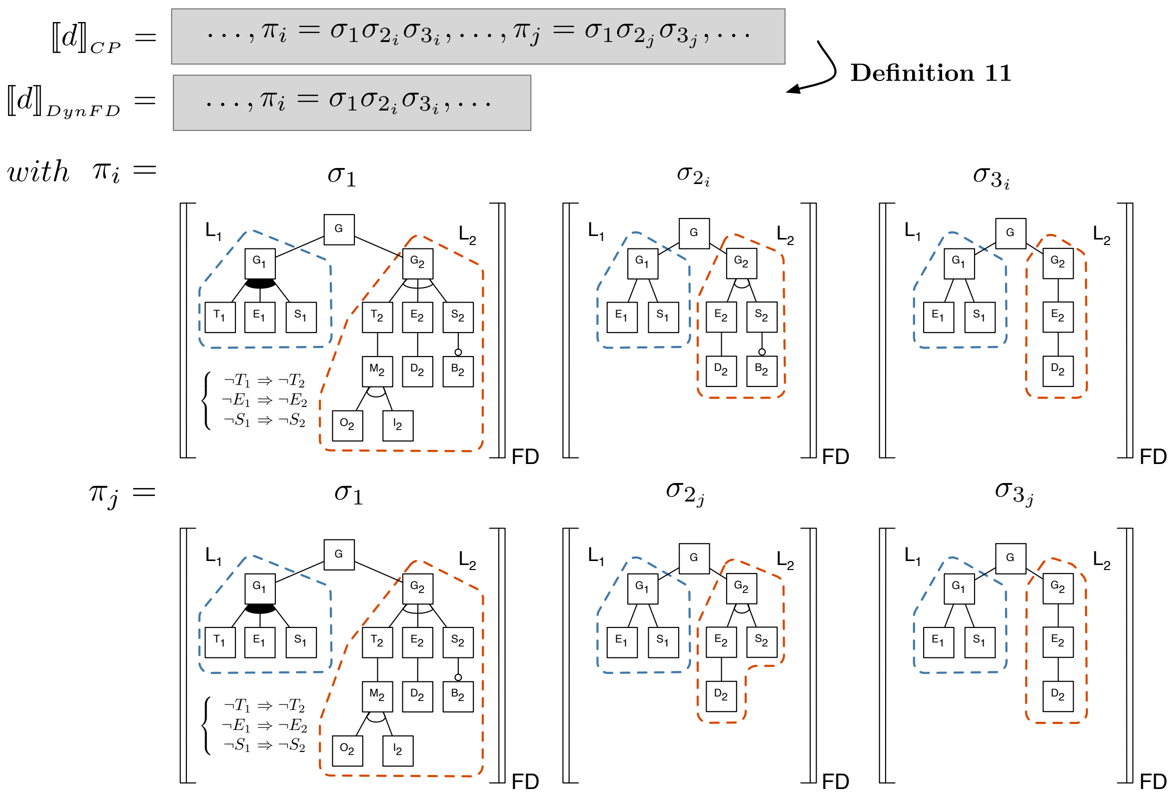


Figure 6. Example of Figure 3 in $[\![d]\!]_{CP}$ and $[\![d]\!]_{DynFD}$.

Theorem 13 (Properties of level arrangements) *Given a diagram $d \in \mathcal{L}_{DynFD}$, each configuration path $\pi \in [\![d]\!]_{DynFD}$ with $\Sigma_1..\Sigma_\ell$ as a valid level arrangement satisfies the following properties.*

(13.1) If $\sigma_j \in \Sigma_i$ then $\forall k < j \bullet |\sigma_{k\,|L_i}| > |\sigma_{j\,|L_i}|$.

(13.2) If $\sigma_j \in \Sigma_i$ and $\sigma_j \neq last(\Sigma_i)$ then $|\sigma_{j\,|L_i}| > 1$.

(13.3) If $|\sigma_{j\,|L_i}| = 1$ then $\forall k > j \bullet \sigma_k \notin \Sigma_i$.

(13.4) If $|\sigma_{j\,|L_i}| = 1$ then $\forall k > j \bullet |\sigma_{k\,|L_i}| = 1$.

Proof.

(13.1) Because of rule (7.2), at each stage at least one configuration is deleted: $\forall k < j \bullet |\sigma_k| > |\sigma_j|$. In addition, (11.2) guarantees that those deleted pertain to the level including the stage: $\forall k < j \bullet |\sigma_{k\,|L_i}| > |\sigma_{j\,|L_i}|$.

(13.2) Immediate consequence of the previous property combined with rule (11.1) saying that $|last(\Sigma_i)_{|L_i}| = 1$.

(13.3) Take the case $k = j + 1$. Suppose that $\sigma_{j+1} \in \Sigma_i$. Because of property (13.1), this would mean that $|\sigma_{j\,|L_i}| > |\sigma_{j+1\,|L_i}|$, i.e. $|\sigma_{j+1\,|L_i}| = 0$, which is impossible since $\forall c \in \sigma_j \bullet root(L_i) \in c$. The cases $k > j + 1$ are similar.

(13.4) $|\sigma_{j\,|L_i}| = 1$ means that all configurations $c \in \sigma_j$ contain $\sigma_{j\,|L_i}$. Given (7.2), only full configurations can be removed, hence the property.

□

Theorem 14 (Uniqueness of level arrangement) *For any diagram $d \in \mathcal{L}_{DynFD}$, a level arrangement for a configuration path $\pi \in [\![d]\!]_{DynFD}$ is unique.*

Proof. Let us suppose that it is possible to find a diagram d that has a valid configuration path $\pi \in [\![d]\!]_{DynFD}$ with more than one level arrangement. Note, that in that case, we need more than one level to begin with: $|L| > 1$. Wlog, each $\pi \in [\![d]\!]_{DynFD}$ with multiple arrangements falls into one of the following three categories.

(a) π consists of a single stage $\pi = \sigma_1$: these π cannot have multiple level arrangements since every level is empty.

(b) σ_2 can be assigned to two different levels

$\pi =$	σ_1	σ_2	$\ldots$
Level arrangement 1		Σ_i	$\ldots$
Level arrangement 2		Σ_{i+k}	$\ldots$

with $\Sigma_1..\Sigma_{i-1} = \epsilon$, $k \geq 1$ and $\Sigma_{i+1}..\Sigma_{i+k-1} = \epsilon$. This situation is impossible; if both assignments obey Definition 11, i.e. they both satisfy rules 11.1 and 11.2, they necessarily exclude each other:

- If arrangement 1 is legal, then $|\sigma_{1\,|L_i}| > 1$ by property (13.1) meaning that arrangement 2 would violate rule (11.1).
- If arrangement 2 is legal, then $|\sigma_{1\,|L_i}| = 1$ by rule (11.1) and arrangement 1 would violate property (13.3).

(c) σ_j, with $j > 2$, can be assigned to two different levels

$\pi =$	$\dots$	σ_{j-1}	σ_j	$\dots$
Level arrangement 1	$\dots$	$\dots$	Σ_i	$\dots$
Level arrangement 2	$\dots$	$\dots$	Σ_{i+k}	$\dots$

with $k \geq 1$ and $\Sigma_{i+1}..\Sigma_{i+k-1} = \epsilon$. This case is similar to (b), i.e. if both arrangements are legal, they also exclude each other:

- If arrangement 1 is legal, then $|\sigma_{j-1\,|L_i}| > 1$ by property (13.1). Level arrangement 2 is impossible since it would violate rule (11.1).
- If arrangement 2 is legal, then $|\sigma_{j-1\,|L_i}| = 1$ by rule (11.1). Given property (13.3), arrangement 1 is then impossible.

Since these are all the situations that might occur, it is impossible for a $\pi \in [\![d]\!]_{DynFD}$ to have multiple level arrangements. □

An immediate consequence of this result is that it is possible to determine a legal arrangement *a posteriori*, i.e. given a configuration path, it is possible to determine a unique level arrangement describing the process followed for its creation. Therefore, levels need not be part of the semantic domain. This result leads to the following definition.

Definition 15 (Subsequence of level arrangement) *Given an FD d and $L_i \in L, \pi \in [\![d]\!]_{DynFD}$, $sub(L_i, \pi)$ denotes the subsequence Σ_i of π pertaining to level L_i for the level arrangement of π that satisfies Definition 11.*

Continuing with Definition 11, remember that rule (11.2) requires that every *deleted* configuration be *deletable* in the stage of the associated level. An immediate consequence of this is that, unless we have reached the end of the configuration path, the set of *deletable* configurations must not be empty, established in Theorem 16. A second theorem, Theorem 17, shows that configurations that are deletable in a stage, are necessarily deleted in this stage.

Theorem 16 *A necessary, but not sufficient replacement for rule (11.2) is that $(\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}) \neq \emptyset$.*

Proof. Immediate via *reductio ad absurdum.* □

Theorem 17 *For rule (11.2) of Definition 11 holds*

$$(\sigma_j \setminus \sigma_{j+1})_{|L_i} \subseteq (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}) \Rightarrow (\sigma_j \setminus \sigma_{j+1})_{|L_i} = (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}).$$

Proof. In Theorem 22 included in Appendix B, we prove that always

$$(\sigma_j \setminus \sigma_{j+1})_{|L_i} \supseteq (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i}).$$

which means that if in addition $(\sigma_j \setminus \sigma_{j+1})_{|L_i} \subseteq (\sigma_{j\,|L_i} \setminus \sigma_{j+1\,|L_i})$ holds, both sets are equal. □

In Theorem 9, Section 4.1, we showed that the transformation rules of Figure 2, i.e. those proposed in [8] that relate to constructs formalised in the abstract syntax of Definition 10, are not expressively complete wrt. the basic staged configuration semantics of Definition 7. The two following theorems provide analogous results, but for the dynamic FD semantics. Basically, the property still holds for the dynamic FD semantics of Definition 11, and a similar property holds for the proposed inter-level link types of Table 2.

Theorem 18 (Incompleteness of transformation rules) *The transformation rules shown in Figure 2 are expressively incomplete wrt. the semantics of Definition 11.*

Proof. We can easily construct an example for $\mathcal{L}_{DynFD}$; it suffices to take the FD used to prove Theorem 9 and to consider it as the sole level of a diagram. From there on, the proof is the same. □

Theorem 19 (Incompleteness of inter-level link types) *The inter-level link types proposed in [8] are expressively incomplete wrt. the semantics of Definition 11.*

Proof. Basically, the proposed inter-level link types always have a sole feature on their right-hand side. It is thus impossible, for example, to express the fact that if some condition ϕ is satisfied for level L_i, all configurations of level L_{i+1} that have f will be excluded if they also have f' (i.e. $\phi \Rightarrow (f' \Rightarrow \neg f)$). □

5.2 Implementation strategies

A formal semantics is generally the first step towards an implementation, serving basically as a specification. In the case of FDs, two main types of tools can be considered: *modelling* tools, used for creating FDs, and *configuration* tools, used during the product derivation phase. Since the only difference between $\mathcal{L}_{FD}$ and $\mathcal{L}_{DynFD}$ is the addition of configuration levels, it should be rather straightforward to extend existing FD modelling tools to $\mathcal{L}_{DynFD}$. In addition, the core of the presented semantics deals with configuration. Let us therefore focus on how to implement a configuration tool for $\mathcal{L}_{DynFD}$, i.e. a tool that allows a user to configure a feature diagram $d \in \mathcal{L}_{DynFD}$, allowing only the configuration paths in $[\![d]\!]_{DynFD}$, and preferably without having to calculate the whole of $[\![d]\!]_{FD}$, $[\![d]\!]_{CP}$ or $[\![d]\!]_{DynFD}$. Also note that, since we do not consider ourselves experts in human-machine interaction, we restrict the following discussion to the implementation of the semantics independently from the user interface. It goes without saying that at least the same amount of thought needs to be devoted to this activity [5].

The foundation of a tool, except for purely graphical ones, is generally a reasoning back-end. Mannion and Batory [17, 2] have shown how an FD d can be encoded as a Boolean formula, say $\Gamma_d \in \mathbb{B}(N)$; and a reasoning tool based on this idea exists for $\mathcal{L}_{FD}$ [25]. The free variables of Γ_d are the features of d, so that, given a configuration $c \in [\![d]\!]_{FD}$, $f_i = true$ denotes $f_i \in c$ and $false$ means $f_i \notin c$. The encoding of d into Γ_d is such that evaluating the truth of an interpretation c in Γ_d is equivalent to checking whether $c \in [\![d]\!]_{FD}$. More generally, satisfiability of Γ_d is equivalent to non-emptiness of $[\![d]\!]_{FD}$. Given this encoding, the reasoning back-end will most likely be a SAT solver, or a derivative thereof, such as a logic truth maintenance system (LTMS) [10] as suggested by Batory [2].

The configuration tool mainly needs to keep track of which features were selected, which were deselected and what other decisions, such as restricting the cardinality of a decomposition, were taken. This *configuration state* basically consists in a Boolean formula $\Delta_d \in \mathbb{B}(N)$, that captures which configurations have been discarded. Feasibility of the current configuration state, i.e. whether all decisions taken were consistent, is equivalent to satisfiability of $\Gamma_d \wedge \Delta_d$. The configuration process thus consists in adding new constraints to Δ_d and checking whether $\Gamma_d \wedge \Delta_d$ is still satisfiable. Indeed, it is possible to find a Boolean constraint for every syntactic transformation rule mentioned in Figure 2. This means, that each time the user takes a decision, she implicitly specifies a formula $\delta \in \mathbb{B}(N)$. The tool first needs to check the validity of this decision wrt. $[\![d]\!]_{FD}$ and previous decisions, i.e. satisfiability of $\Gamma_d \wedge \Delta_d \wedge \delta$. If it is consistent, the tools adds δ to Δ_d (as a conjunction) and moves to the next decision.

A tool implementing the procedure sketched in the previous paragraph will inevitably respect $[\![d]\!]_{FD}$. In order to respect $[\![d]\!]_{CP}$, however, the configuration tool also needs to make sure that each time a decision δ is taken, all other decisions implied by δ be taken as well, for otherwise rule (7.2) might be violated in subsequent stages. This can easily be achieved using an LTMS which can propagate constraints as the user makes decisions. This way, once she has selected a feature f that excludes a feature f', the choice of f' will not be presented to the user anymore. The LTMS will make it easy to determine which variables, i.e. features, are still free and the tool should only present those to the user.

The extended procedure would still violate $[\![d]\!]_{DynFD}$, since it does not enforce constraints that stem from level definitions. A second extension is thus to make sure that the tool respects the order of the levels as defined in d, and only presents choices pertaining to the current level L_i until it is dealt with. This means that the formula of a decision δ may only involve features f that are part of the current level (rule (11.2)). It also means that the tool needs to be able to detect when the end of a level L_i has come (rule (11.1)), which is equivalent to checking whether, in the current state of the LTMS, all of the $f \in L_i$ are assigned a fixed value.

Given these guidelines, it should be relatively straightforward to come up with an architecture and some of the principal algorithms for a tool implementation.

5.3 FD Analysis

As argued in Section 2, the advantage of a semantics is the ability to specify properties and indicators independently from an actual implementation. In the present case, several such properties come to mind.

Commutativity of levels. During the elaboration of an FD, the analyst needs to decide the order of the levels. Since this order determines which configuration paths are considered legal, an important information at this point is whether two levels can be interchanged, i.e. whether they are *commutative*. Two levels are called commutative if the tails of the configuration paths, starting with the later one, are the same for either order of them.

Definition 20 (Commutative levels) *Given an FD d, so that $L = [..L_a..L_b..]$, L_a and L_b are said to be* commutative*, iff the following holds:*

$$\begin{aligned}&\{final(sub(L_b, \pi)) | \pi \in [\![d]\!]_{DynFD}\}\\ &\quad = \quad \{final(sub(L_a, \pi)) | \pi \in [\![d']\!]_{DynFD}\},\end{aligned}$$

where $d' = d$ except for $L' = L$ with L_a and L_b inverted.

Feature impact. Once an FD is elaborated and the different stakeholders start making decisions, it is interesting to identify the features with the biggest impact on the variability and subsequent process. Consider a case in which a stakeholder is indifferent about the order in which she decides about some feature's inclusion. If she first decides about the feature with the largest impact, i.e. the one that eliminates the most of the other choices, this can substantially speed up the process. One way to achieve this is to present the choices following the partial order induced by

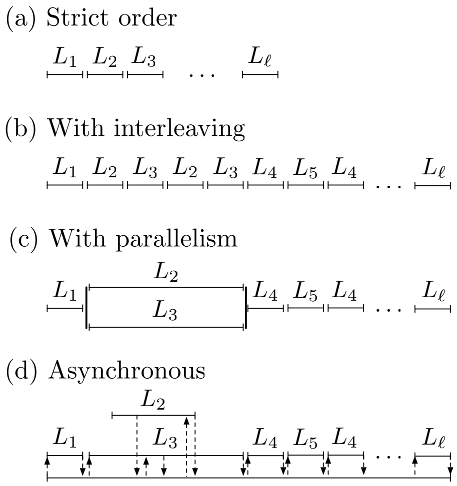


Figure 7. Possible extensions.

the decomposition relation. This way is not optimal, however, since children of the same parent are not ordered, and since it does not take the additional constraints ϕ into account. Hence we propose the following (wrt. the implementation strategy of the previous section).

Definition 21 (Feature impact) *A feature f is said to have* maximal impact *among features $F \subseteq N$ if*

$$f = \max_{n \in N} \big(freevars(\Gamma_d \wedge \Delta_d) - freevars(\Gamma_d \wedge \Delta_d \wedge n)\big)$$

and minimal impact *if*

$$f = \min_{n \in N} \big(freevars(\Gamma_d \wedge \Delta_d) - freevars(\Gamma_d \wedge \Delta_d \wedge n)\big)$$

6 Extensions and future work

A number of extensions to the dynamic FD semantics can be envisioned. We will first consider approaches that relax the restrictions put on the definition of the FD levels, and then discuss the more obvious approach of adding new FD constructs to the formalism.

6.1 Relaxing restrictions on levels

The semantics defined in Section 4 inherits from the original definition of MLSC [8] the assumption that levels are configured one after the other in a strict order until the final configuration is obtained. We will gradually lift these hypotheses, and discuss their implications on the semantics.

Interleaved levels. As stated in Definition 10, for an FD $d \in \mathcal{L}_{DynFD}$, levels are specified with a strict order that must be followed during the configuration process, as illustrated in Figure 7(a). In a realistic project, however, this restriction might prove to be too strong. It is more likely that several stakeholders may want to take decisions pertaining to their respective levels during the same period of time, for instance because they are independent. A graphical illustration is provided in Figure 7(b). Nevertheless, project management might still want to avoid chaotic or arbitrary procedures, by imposing a more loose order which allows for interleaving of levels.

Because of the interleaving, this intuition cannot be captured by any kind of order on the set of levels. A loose order with interleaving is a specification of the possible level arrangements, rather than an order on the levels themselves. Definition 11 requires for a configuration path to be legal, that there must be an assignment of stages to levels that satisfies certain properties. Intuitively, one can think of this assignment as a labelling of the transitions between stages by levels, i.e.

$$\sigma_i \xrightarrow{L_j} \sigma_{i+1} \triangleq \sigma_{i+1} \in \Sigma_j.$$

Now consider the sequence consisting only of the L_j's noted on top of the arrows. With the current definition, this sequence is required to be of the form

$$L_1..L_1..L_2..L_2....L_\ell.$$

Interleaving means that we want to also allow, for instance,

$$L_1..L_2..L_3..L_2..L_3..L_4..L_5..L_4....L_\ell,^5$$

while still being able to specify which interleavings are allowed (e.g. only L_2/L_3 and L_4/L_5), and what the minimal order is (e.g. a strict order from L_6 on). One way to do that is with a regular expression over the L_i's, such as

$$L_1^*(L_2|L_3)^*(L_4|L_5)^*L_6^*...L_\ell^*$$

that expresses the example constraints noted above, or

$$L_1^*L_2^*...L_\ell^*$$

corresponding to the constraint as it is in the current definition. So, rather than to specify an order on L, Definition 10 would require to specify a regular expression over elements of L that has to be enforced in the semantic function of Definition 11. Given that regular expressions can be expressed equivalently by automata and statecharts, those alternative formalisms could also be used to specify the allowed sequences. Namely an automaton, or statechart, where states correspond to levels and transitions denote sequence, might allow stakeholders and managers with a less formal background to specify the configuration process intuitively.

Interleaving also gives rise to a number of interesting analysis properties. During the elaboration of the FD,

[5] Where L_1 now has to be fully configured at its last occurrence in the sequence rather than when L_2 starts.

mainly when defining the configuration process, it is important to know which levels are safe to be interleaved. To this end, it is interesting to know whether two levels are independent, i.e. whether there are direct/indirect inter-level links between them, or whether they influence the same features in a subsequent level.

Parallel levels. Actually, interleaving, with well-done tool support, already allows for pseudo-parallelism; similar to how a single core processor allows multiple programs to run in parallel by interleaving their instructions. It requires, however, the model to be accessible for configuration simultaneously from different places. Google Docs, and the recent software-as-a-service trend, show how this is possible even within a browser, yet these approaches generally require a live Internet connection. Lifting this barrier to distributed offline collaboration would make the semantics even more fit to real scenarios, since parallel configuration does happen in the real world, for instance in automotive software engineering [8]. Figure 7 illustrates the differences between strict order, interleaving and parallelism graphically.

The idea of parallel levels, as shown in Figure 7(c), is thus to give a clone of the base FD to each stakeholder, so that she can configure her level independently from other stakeholders, and to merge the clones once all stakeholders have finished their configuration. When the implementation strategy of Section 5 is followed, the merge operation itself is rather trivial, since the decisions of each stakeholder are recorded in a Boolean formula. A merge conflict arises when the merge (i.e. the conjunction) of the individual decisions yields an illegal configuration. In this case, it is important to be able to pinpoint the exact cause. Merge conflicts can be minimised if special care is taken during the elaboration of the FD when the parallel levels are specified. Namely, there need to be indicators that evaluate the degree of independence of two levels and that can be used to evaluate the probability of conflicts during the merge. Note that automata or statecharts do not readily allow to express parallelism, therefore, Petri nets or activity diagrams should be considered for the specification of allowed level sequences.

Asynchronous levels. The advantage of parallel levels is that distributed groups can work independently on their local copies. The model of parallelism introduced previously, however, still assumes a certain amount of co-ordination, namely at the fork and merge points. This can lead to problems; imagine, for instance, that L_2 and L_3 run in parallel. If the configuration of L_3 takes longer than expected, the subsequent levels will have to wait for L_3 to finish, even though L_2 is already configured.

This problem could be solved by considering a completely asynchronous approach, as shown in Figure 7(d). There is a central base model, but instead of executing configurations on the base model, each level is locally configured and merges back its decision into the base model on the fly. This way, L_2 can be merged back to the central model even before L_3 is finished. If merges are assumed to work in the other direction as well, then this can also reduce the potential for conflict, since each level can merge its changes back to the central model as it progresses with the configuration (note that, if a merge is done for every stage, this is roughly equal to the interleaving mode). One conflict avoidance strategy would be to first take decisions with a great impact (see Definition 21), merge them back to the base model, and continue with decisions of lesser impact. This asynchronous level model actually corresponds to how the popular source control systems SVN and CVS work, and is thus the closest to reality.

Crosscutting levels. In Definition 11, we require that each level has to be an FD in itself, with no sharing of features or decomposition edges between levels. It could be imagined to lift this hypothesis, too, and thereby allow levels to be crosscutting. Indeed, Lee *et al.* suggested to group features into *binding units*, denoting major functionalities of the system, so that nodes shared between binding units allow for additional constraints [15]. Given that binding units can be represented by levels in our semantics, it seems that such an extension would make sense. Implications on the semantics, however, have to be considered carefully.

Infinite sequences of levels. Finally, our semantics assumes in Definition 7 that each configuration path is finite. A growing community of researchers, however, considers systems that change at runtime, so-called self-adaptive systems [11, 15]. In terms of configuration paths, this means (1) that a configuration path is not finite, since configuration can continue indefinitely at runtime and (2) that there is never a stage in which only one configuration is left, since from then on no further configuration would be possible.

Our semantics would therefore need to be extended in several ways. First, the sequence of levels can no longer be specified with a regular language or by a finite state automaton as assumed above, but rather by automata that define languages with infinite words, such as Büchi automata. In addition to that, as the configuration path will not necessarily end in a single configuration it is not possible to determine which of the many configurations of a runtime level is the one that is currently active. Therefore, besides specifying inclusion or exclusion of configurations, we need to be able to specify which of the included configurations is active and which are inactive.

6.2 Adding Constructs

Another, more obvious way, to extend this work is by adding new constructs to the FD language. In Section 2, we pointed out that the concepts of *feature attribute* and *feature cloning* (through *feature references* and $\langle i..j \rangle$ *fea-*

ture cardinalities with $j > 1$) are part of the FD language for which MLSC was originally proposed [8] but not of the present semantics. Recently, Benavides *et al.* [3] showed some interesting applications of feature attributes, mainly in conjunction with optimisation problems. Other constructs, such as *binding time* annotations [13, 9], are also candidates for inclusion in an extended version of the semantics.

Adding feature attributes to the semantics should be rather straightforward. Indeed, it suffices to (i) add an attribute construct to the abstract syntax, to (ii) add constructs that allow to express constraints on attributes in Φ, to (iii) extend the semantic domain with a function that returns the value of a feature attribute, and to (iv) adapt the semantic function so that it treats attribute modifications as specialisation steps. The only major implication of this change is that the semantic domain will become infinite if the domain of an attribute is infinite. The SAT-based implementation strategy remains valid, but in order to take advantage of the full range of applications on feature attributes (e.g. attribute maximisation and minimisation), constraint solvers need to be used as well.

If the levels of an FD are used to represent binding times, then binding time constraints such as those proposed in [9] translate to the restrictions on allowed level arrangements discussed in the previous subsection. Using regular languages to express constraints on the relative temporal placement of levels is, however, not likely to be very intuitive. Other formalisms, such as Allen's interval algebra [1] should be considered instead.

Finally, feature cloning would require some more drastic changes to the semantics, since the same feature might appear several times in a configuration. Simply defining a configuration as a multiset of features, rather than a set, would not solve the problem. Indeed, multisets do not allow to always identify the parent of a feature, since the parent might appear twice in the multiset. In order to do so, the semantic domain needs to keep information about the FD decomposition relation, as done by Czarnecki *et al.* [7].

7 Conclusion

We introduced a dynamic formal semantics for FDs that allows reasoning about its configuration paths, i.e. the configuration process, rather than only about its allowed configurations. Extending the basic dynamic semantics with levels yields a semantics for MLSC. The contribution of the paper is therefore a precise and formal account of MLSC that makes the original definition [8] more explicit and reveals some of its subtleties and incompletenesses. Based on the semantics we show some interesting properties of configuration paths and outline an implementation strategy that uses SAT solvers as the reasoning back-end.

Acknowledgements

This work is sponsored by the Interuniversity Attraction Poles Programme of the Belgian State of Belgian Science Policy under the MoVES project and the FNRS.

References

[1] J. F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[2] D. S. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th Int. Software Product Line Conference (SPLC)*, pages 7–20, 2005.

[3] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated Reasoning on Feature Models. *Proceedings of the 17th International Conference (CAiSE'05) LNCS, Advanced Information Systems Engineering.*, 3520:491–503, 2005.

[4] J. Bosch. Keynote—Software Product Families in Nokia. *Proceedings of the 9th International Conference on Software Product Lines (SPLC'05)*, pages 2–6, September 2005.

[5] C. Cawley, D. Nestor, A. Preußner, G. Botterweck, and S. Thiel. Interactive visualisation to support product configuration in software product lines. In *Proceedings of the Second International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'08), Essen, Germany*, 2008.

[6] S. Cohen, B. Tekinerdogan, and K. Czarnecki. A case study on requirements specification: Driver Monitor. In *Proc. of the Workshop on Techniques for Exploiting Commonality Through Variability Management at the Second International Conference on Software Product Lines (SPLC'02)*, 2002.

[7] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[8] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[9] E. Dolstra, G. Florijn, M. de Jonge, and E. Visser. Capturing timeline variability with transparent configuration environments. In J. Bosch and P. Knauber, editors, *IEEE Workshop on Software Variability Management (SVM'03)*, Portland, Oregon, May 2003. IEEE.

[10] K. Forbus and J. de Kleer. *Building Problem Solvers*. The MIT Press, 1993.

[11] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid.

Dynamic software product lines. *IEEE Computer*, 41(4):93–95, 2008.

[12] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff - part I: The basic stuff. Technical Report MCS00-16, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Israel, September 2000.

[13] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[14] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annales of Software Engineering*, 5:143–168, 1998.

[15] J. Lee and K. C. Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *10th International Software Product Line Conference*, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[16] K. Lee, K. C. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. In *ICSR-7: Proceedings of the 7th International Conference on Software Reuse*, pages 62–77, London, UK, 2002. Springer-Verlag.

[17] M. Mannion. Using First-Order Logic for Product Line Model Validation. In *Proceedings of the Second Software Product Line Conference (SPLC'02)*, LNCS 2379, pages 176–187, San Diego, CA, Aug. 2002. Springer.

[18] K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, July 2005.

[19] R. Rabiser, P. Grunbacher, and D. Dhungana. Supporting product derivation by adapting and augmenting variability models. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 141–150, Washington, DC, USA, 2007. IEEE Computer Society.

[20] M.-O. Reiser and M. Weber. Managing highly complex product families with multi-level feature trees. In *IEEE International Conference on Requirements Engineering (RE'06)*, pages 146–155, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[21] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, pages 139–148, Minneapolis, Minnesota, USA, September 2006.

[22] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks (2006), doi:10.1016/j.comnet.2006.08.008, special issue on feature interactions in emerging application domains*, page 38, 2006.

[23] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, 1977.

[24] S. Thiel and A. Hein. Modeling and Using Product Line Variability in Automotive Systems. *IEEE Software*, 19(4):66–72, 2002.

[25] J.-C. Trigaux and P. Heymans. Varied feature diagram (vfd) language: A reasoning tool. Technical Report EPH3310300R0462 / 215315, PRECISE, University of Namur, January 2007. PLENTY: Product Line ENgineering of food TraceabilitY software.

A Illustration: calculation details

This section presents the calculation details of the two configuration paths, π_i and π_j, used in the example of Section 4.3. In the remainder of this section, we will use d when referring to the FD of Figure 5 and we note $\sigma \xrightarrow{L_j} \sigma'$ if $\pi = ..\sigma\sigma'..$ and $\sigma' \in \Sigma_j$.

A.1 Legal path (π_i)

This configuration path, shown in Figure 3 and in the middle part of Figure 6, actually corresponds to the one used in the original illustration by Czarnecki *et al.* [8], which was summarised in Section 3. One can see in Figure 3 that it consists of two manual configuration stages and one automatic specialisation stage. Since the *inter-level links* already implement the latter stage, there is no need to represent it as a dedicated transition between two stages. The results of the two former stages result from manual configurations that cannot be automatically derived from the constraint set and thus have to be represented by two separate configuration sets, *viz.* σ_{2_i} and σ_{3_i}.

In terms of $[\![d]\!]_{CP}$, π_i is a sequence $\sigma_1\sigma_{2_i}\sigma_{3_i}$ with:

$$
\begin{array}{lcll}
\sigma_1 & = & \{ & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, T_2, M_2, O_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, T_2, M_2, I_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, E_2, D_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2, B_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, G_2, T_2, M_2, O_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, G_2, T_2, M_2, I_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, G_2, E_2, D_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, T_2, M_2, O_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, T_2, M_2, I_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, S_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, S_2, B_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, E_2, D_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2, B_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{T_1}, G_2, T_2, M_2, O_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{T_1}, G_2, T_2, M_2, I_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{E_1}, G_2, E_2, D_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{S_1}, G_2, S_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{S_1}, G_2, S_2, B_2\} \quad \} \\
\sigma_{2_i} & = & \{ & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, E_2, D_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2\}, \\
& & & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2, B_2\} \quad \} \\
\sigma_{3_i} & = & \{ & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, E_2, D_2\} \quad \}
\end{array}
$$

Note here that $|(\sigma_{3_i})| = 1$, which denotes the end of the configuration path (Definition 7), resulting in the single product $\{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, E_2, D_2\}$.

In order to show that $\pi_i \in [\![d]\!]_{DynFD}$, we have to find a level arrangement that satisfies rules (11.1) and (11.2). Let us examine each possible level arrangement in turn.

(a) For $\Sigma_{1_i} = \epsilon$ and $\Sigma_{2_i} = \sigma_{2_i}\sigma_{3_i}$, we have

$$|final(\Sigma_{1_i})_{|L_1}| = |\sigma_{1\,|L_1}| = 7 > \mathbf{1}$$

The arrangement is thus **rejected** because (11.1) evaluates to false.

(b) For $\Sigma_1 = \sigma_{2_i}\sigma_{3_i}$ and $\Sigma_{2_i} = \epsilon$, we have

$$|final(\Sigma_{1_i})_{|L_1}| = |\sigma_{3_i\,|L_1}| = 1$$
$$|final(\Sigma_{2_i})_{|L_2}| = |\sigma_{3_i\,|L_2}| = 1$$

and (11.1) evaluates to true. We have for $\sigma_1 \xrightarrow{L_1} \sigma_{2_i}$:

$$
\begin{array}{lll}
\mathbf{1}.\ \sigma_1 \setminus \sigma_{\mathbf{2_i}} = & \{ & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, T_2, M_2, O_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, T_2, M_2, I_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, E_2, D_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2, B_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, G_2, T_2, M_2, O_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, G_2, T_2, M_2, I_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, G_2, E_2, D_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, T_2, M_2, O_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, T_2, M_2, I_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, S_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}, G_2, S_2, B_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, G_2, T_2, M_2, O_2\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, G_2, T_2, M_2, I_2\}, \\
& & \{\mathbf{G_1}, \mathbf{E_1}, G_2, E_2, D_2\}, \\
& & \{\mathbf{G_1}, \mathbf{S_1}, G_2, S_2\}, \\
& & \{\mathbf{G_1}, \mathbf{S_1}, G_2, S_2, B_2\} \quad \} \\
\mathbf{2}.\ (\sigma_1 \setminus \sigma_{\mathbf{2_i}})_{|\mathbf{L_1}} = & \{ & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{E_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{S_1}\} \quad \}
\end{array}
$$

and

$$
\begin{array}{lll}
\mathbf{1}.\ \sigma_{1\,|\mathbf{L_1}} = & \{ & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{E_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{S_1}\} \quad \} \\
\mathbf{2}.\ \sigma_{\mathbf{2_i}\,|\mathbf{L_1}} = & \{ & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}\} \quad \} \\
\mathbf{3}.\ \sigma_{1\,|\mathbf{L_1}} \setminus \sigma_{\mathbf{2_i}\,|\mathbf{L_1}} = & \{ & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{T_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{E_1}\}, \\
& & \{\mathbf{G_1}, \mathbf{S_1}\} \quad \}
\end{array}
$$

such that $(\sigma_1 \setminus \sigma_{2_i})_{|L_1} \subseteq (\sigma_{1\,|L_1} \setminus \sigma_{2_i\,|L_1})$ is true and we have for $\sigma_{2_i} \xrightarrow{L_1} \sigma_{3_i}$:

$$\begin{aligned} (\sigma_{\mathbf{2_i}} \setminus \sigma_{\mathbf{3_i}})_{|\mathbf{L_1}} = \quad & \{ \quad \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2\}, \\ & \quad\;\; \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2, B_2\} \quad \} \\ \sigma_{\mathbf{2_i}\,|\mathbf{L_1}} \setminus \sigma_{\mathbf{3_i}\,|\mathbf{L_1}} = \quad & \emptyset \end{aligned}$$

The arrangement is thus **rejected** because (11.2) evaluates to false since $(\sigma_{2_i} \setminus \sigma_{3_i})_{|L_1} \subseteq (\sigma_{2_i\,|L_1} \setminus \sigma_{3_i\,|L_1})$ does not hold. This rejection is obvious since level L_1 was already fully configured at σ_{2_i}, hence the absence of deletable configurations from σ_{2_i} in L_1.

(c) For $\Sigma_{1_i} = \sigma_{2_i}$ and $\Sigma_{2_i} = \sigma_{3_i}$, we have

$$|final(\Sigma_{1_i})_{|L_1}| = |\sigma_{2_i\,|L_1}| = 1$$

$$|final(\Sigma_{2_i})_{|L_2}| = |\sigma_{3_i\,|L_2}| = 1$$

and (11.1) evaluates to true. We know from above that for $\sigma_1 \xrightarrow{L_1} \sigma_{2_i}$:

$$(\sigma_1 \setminus \sigma_{2_i})_{|L_1} \subseteq (\sigma_{1\,|L_1} \setminus \sigma_{2_i\,|L_1})$$

and we have for $\sigma_{2_i} \xrightarrow{L_2} \sigma_{3_i}$:

$$\begin{aligned} (\sigma_{\mathbf{2_i}} \setminus \sigma_{\mathbf{3_i}})_{|\mathbf{L_2}} = \quad & \{ \quad \{G_2, S_2\}, \\ & \quad\;\; \{G_2, S_2, B_2\} \quad \} \\ \sigma_{\mathbf{2_i}\,|\mathbf{L_2}} \setminus \sigma_{\mathbf{3_i}\,|\mathbf{L_2}} = \quad & \{ \quad \{G_2, S_2\}, \\ & \quad\;\; \{G_2, S_2, B_2\} \quad \} \end{aligned}$$

i.e. that $(\sigma_{2_i} \setminus \sigma_{3_i})_{|L_2} \subseteq (\sigma_{2_i\,|L_2} \setminus \sigma_{3_i\,|L_2})$. Since both (11.1) and (11.2) evaluate to true, the arrangement is **accepted**.

One level arrangement, *viz.* $\Sigma_{1_i} = \sigma_{2_i}$ and $\Sigma_{2_i} = \sigma_{3_i}$, satisfies rules (11.1) and (11.2) and thus $\pi_i \in [\![d]\!]_{DynFD}$.

A.2 Illegal path (π_j)

Let us now illustrate the illegal configuration path $\pi_j \in [\![d]\!]_{CP}$ sketched in Figure 6. Intuitively, π_j is illegal because feature B_2, belonging to L_2, is deselected even though L_1 is not yet finished. The configuration path π_j is thus the same as π_i except for σ_{2_j}, which is now:

$$\begin{aligned} \sigma_{2_j} \quad = \quad \{ \quad & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, E_2, D_2\}, \\ & \{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2\} \quad \} \end{aligned}$$

Again, in order for $\pi_j \in [\![d]\!]_{DynFD}$ there must be at least one level arrangement that satisfies rules (11.1) and (11.2). Let us examine each possible arrangement in turn.

(a) For $\Sigma_{1_j} = \epsilon$ and $\Sigma_{2_j} = \sigma_{2_j}\sigma_{3_j}$, we have

$$|final(\Sigma_{1_j})_{|L_1}| = |\sigma_{1\,|L_1}| = 7 \quad > \mathbf{1}$$

The arrangement is thus **rejected** because (11.1) evaluates to false.

(b) For $\Sigma_{1_j} = \sigma_{2_j}\sigma_{3_j}$ and $\Sigma_{2_j} = \epsilon$, we have

$$|final(\Sigma_{1_j})_{|L_1}| = |\sigma_{3_j\,|L_1}| = 1$$

$$|final(\Sigma_{2_j})_{|L_2}| = |\sigma_{3_j\,|L_2}| = 1$$

and (11.1) evaluates to true. We have for $\sigma_1 \xrightarrow{L_1} \sigma_{2_j}$:

$$\begin{aligned} (\sigma_{\mathbf{1}} \setminus \sigma_{\mathbf{2_j}})_{|\mathbf{L_1}} = \quad \{ \quad & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}\}, \\ & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}\}, \\ & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}\}, \\ & \underline{\{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}\}}, \\ & \{\mathbf{G_1}, \mathbf{T_1}\}, \\ & \{\mathbf{G_1}, \mathbf{E_1}\}, \\ & \{\mathbf{G_1}, \mathbf{S_1}\} \quad \} \\ (\sigma_{\mathbf{1}} \setminus \sigma_{\mathbf{2_j}})_{|\mathbf{L_1}} = \quad\;\; & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}, \mathbf{S_1}\}, \\ & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{E_1}\}, \\ & \{\mathbf{G_1}, \mathbf{T_1}, \mathbf{S_1}\}, \\ & \{\mathbf{G_1}, \mathbf{T_1}\}, \\ & \{\mathbf{G_1}, \mathbf{E_1}\}, \\ & \{\mathbf{G_1}, \mathbf{S_1}\} \quad \} \end{aligned}$$

The arrangement is thus **rejected** because (11.2) evaluates to false since $(\sigma_1 \setminus \sigma_{2_j})_{|L_1} \supset (\sigma_{1\,|L_1} \setminus \sigma_{2_j\,|L_1})$. Intuitively, this condition is false because the configuration $\underline{\{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}, G_2, S_2, B_2\}}$ is part of the deleted ones but was not deletable since B_2 belongs to another level and all configurations containing $\{\mathbf{G_1}, \mathbf{E_1}, \mathbf{S_1}\}$ where bound to be selected.

(c) For $\Sigma_{1_j} = \sigma_{2_j}$ and $\Sigma_{2_j} = \sigma_{3_j}$, we have

$$|final(\Sigma_{1_j})_{|L_1}| = |\sigma_{2_j\,|L_1}| = 1$$

$$|final(\Sigma_{2_j})_{|L_2}| = |\sigma_{3_j\,|L_2}| = 1$$

and (11.1) evaluates to true. Like for $\sigma_{2_j}, \sigma_{3_j} \in \Sigma_{1_j}$ and $\Sigma_{2_j} = \emptyset$, we have for $\sigma_1 \xrightarrow{L_1} \sigma_{2_j}$ that $(\sigma_1 \setminus \sigma_{2_j})_{|L_1} \supset (\sigma_{1\,|L_1} \setminus \sigma_{2_j\,|L_1})$. The arrangement is thus **rejected** because (11.2) evaluates to false.

Since none of the possible arrangements satisfies both (11.1) and (11.2), the configuration path π_j does not belong to $[\![d]\!]_{DynFD}$.

B Proof helper

The following theorem is used for the proof of Theorem 22, Section 5. Basically, it says that when two sets of sets are reduced to include only sets containing certain elements and then subtracted, the result is included in (i.e. smaller than) the set obtained by subtracting first and reducing afterwards. Intuitively, the result of a subtraction operation is smaller the more elements in both sets "match up", and if a reduction is applied prior to subtracting, it becomes "more likely" for elements to match up, meaning that the result of the subtraction can be smaller than if the reduction was applied afterwards.

Theorem 22 *For some set N, if $\sigma, \sigma' \in \mathcal{P}\mathcal{P}N$ so that $\sigma \supset \sigma'$ and $L \subseteq N$, then*

$$\big\{\{c \cap L | c \in \sigma\} \setminus \{c \cap L | c \in \sigma'\}\big\} \subseteq \{c \cap L | c \in (\sigma \setminus \sigma')\}.$$

Proof.

$$\begin{aligned}
&\big\{\{c \cap L | c \in \sigma\} \setminus \{c \cap L | c \in \sigma'\}\big\} \\
= &\big\{a | (a \in \{c \cap L | c \in \sigma\}) \wedge (a \notin \{c \cap L | c \in \sigma'\})\big\} \\
= &\big\{a | (a \in \{p | p = \{x | \exists c \in \mathcal{P}N \bullet x \in c \wedge x \in L \wedge c \in \sigma\}\} \\
&\quad \wedge a \notin \{p | p = \{x | \exists c \in \mathcal{P}N \bullet x \in c \wedge x \in L \wedge c \in \sigma'\}\})\big\} \\
= &\big\{p | p = \{x | (\exists c \in \mathcal{P}N \bullet x \in c \wedge x \in L \wedge c \in \sigma) \\
&\qquad \wedge \neg(\exists c \in \mathcal{P}N \bullet x \in c \wedge x \in L \wedge c \in \sigma')\}\big\} \\
= &\big\{p | p = \{x | (\exists c \in \mathcal{P}N \bullet x \in c \wedge x \in L \wedge c \in \sigma) \\
&\qquad \wedge \ (\forall c \in \mathcal{P}N \bullet \neg(x \in c \wedge x \in L \wedge c \in \sigma'))\}\big\} \\
\subseteq & \\
&\big\{p | p = \{x | \exists c \in \mathcal{P}N \bullet (x \in c \wedge x \in L \wedge c \in \sigma) \\
&\qquad\qquad \wedge \neg(x \in c \wedge x \in L \wedge c \in \sigma')\}\big\} \\
= &\big\{p | p = \{x | \exists c \in \mathcal{P}N \bullet x \in c \wedge x \in L \wedge c \in \sigma \wedge \neg(c \in \sigma')\}\big\} \\
= &\big\{p | p = \{x | x \in c \wedge x \in L \wedge c \in \sigma \wedge c \notin \sigma'\}\big\} \\
= &\big\{c \cap L | c \in \sigma \wedge c \notin \sigma'\big\} \\
= &\big\{c \cap L | c \in (\sigma \setminus \sigma')\big\}
\end{aligned}$$

$\square$