Product Lines of Graphical Modelling Languages

Antonio Garmendia
Antonio.Garmendia@uam.es
Universidad Autónoma de Madrid
Madrid, Spain

Esther Guerra
Esther.Guerra@uam.es
Universidad Autónoma de Madrid
Madrid, Spain

Juan de Lara
Juan.deLara@uam.es
Universidad Autónoma de Madrid
Madrid, Spain

ABSTRACT

Modelling languages are essential in many disciplines to express knowledge in a precise way. Furthermore, some domains require families of notations (rather than individual languages) that account for variations of a language. Some examples of language families include those to define automata, Petri nets, process models or software architectures. Several techniques have been proposed to engineer families of languages, but they often neglect the language's concrete syntax, especially if it is graphical.

To fill this gap, we propose a modular method to build product lines of graphical modelling languages. Language features are defined in modules, which comprise both the abstract and graphical concrete syntax of the feature. A language variant is selected by choosing a valid configuration of modules, from which the abstract and concrete syntax of the variant is synthesised. Our approach permits *composition* and *overriding* of graphical elements (e.g., symbol styles, visualisation layers), the injection of pre-defined graphical styles into language families (e.g., to obtain a high-intensity contrast variant for accessibility), and the analysis of graphical conflicts at the product line level. We report on an implementation atop Eclipse/Sirius, and demonstrate its benefits by an evaluation which shows a substantial specification size reduction of our product line method with respect to a case-by-case specification approach.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**; **Domain specific languages**.

KEYWORDS

Software Language Engineering, Model-driven Engineering, Graphical Concrete Syntax, Product Lines

ACM Reference Format:
Antonio Garmendia, Esther Guerra, and Juan de Lara. 2024. Product Lines of Graphical Modelling Languages. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3640310.3674082

MODELS '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0504-5/24/09
https://doi.org/10.1145/3640310.3674082

1 INTRODUCTION

Modelling languages are pervasive across many disciplines to represent knowledge about objects, processes or systems. They are heavily used in computer science to facilitate software development, most prominently in model-driven engineering (MDE) [1]. Furthermore, many times, modelling languages come in families of related notations, like those to define automata [8], Petri nets [34], process models [32], access control [23] or software architectures [27].

A language may also have variants to target specific aspects like the modelling phase (e.g., using simpler and more permissive variants when a software project starts, and stricter, more detailed ones as the project progresses [18]), the features of the modelled system (e.g., requiring variants with different expressive power, like Petri nets with/without inhibitor arcs [34]), the context of use of the language (e.g., using variants of class diagrams that hide class attributes or role names in small screens, and display them in larger ones), or the user expertise (e.g., providing simpler language versions to novice users than to expert users [20]), among other scenarios. Variants may pertain the language syntax and semantics. In this paper, our focus is on the syntax.

Different approaches have been proposed to create families of languages, often resorting to software product line engineering [35]. Some of them are *compositional*, in which case, the features of the languages are defined as components that can be composed to produce language variants [2–4, 6, 9]. Conversely, *annotative* approaches build a unified description of all variants, which is then deprived of the features not needed to obtain a particular variant [19, 37]. Approaches exist for defining product lines of textual languages based on grammars (e.g., Neverlang [43], MontiCore [2]), however, we are not aware of proposals considering product lines of graphical languages. Our goal is to fill this gap.

We base our work on the notion of language product line proposed in [9], where a language family is defined by a set of modules, each encapsulating a meta-model fragment. Modules may declare dependencies to other modules, forming a tree structure akin to a feature model [22]. A variant is selected by choosing a configuration (a set of modules), and then, the abstract syntax is produced by merging the meta-model fragments of the chosen modules. The approach also permits expressing the semantics of the family by graph transformation. In this paper we omit the part on semantics, since it is not needed and is directly applicable to the current work. Instead, we extend the approach to support graphical concrete syntaxes for the language family.

Specifically, modules can now attach a graphical syntax specification fragment. This fragment can contain graphical elements such as symbols for meta-classes and associations, (conditional) styles, positional constraints for associations (e.g., containment), and presentation layers. Our approach composes the needed graphical specification fragments when a language variant is selected,

and manages overriding and merging (e.g., to add compartments in a container node). It also supports the definition of *graphical language aspects*, which can be automatically woven with a language family definition to create new variants. For example, we have created one aspect that produces accessible variants of a language family by overriding the defined concrete syntax with high-contrast coloured symbols; and another aspect for internationalisation that translates the symbol labels into the language of interest (e.g., English, French). Finally, we provide a method for detecting conflicts between graphical specification fragments at the product line level.

We have realised our proposal atop Capone, an open-source Eclipse tool for modular meta-model product lines presented in [9]. We have extended the tool to support graphical syntax specifications using Sirius's *odesign* models [40]; to synthesise stand-alone, ready-to-use Sirius editors for the chosen variants; and to analyse conflicts in the graphical specification of a language family. To show the benefits of our approach in terms of specification size, we have performed an experiment with four case studies of language families, obtaining substantial specification size reduction both when creating a family from scratch and when adding a new feature.

Paper organisation. Section 2 introduces modular language product lines [9]. Section 3 extends them with graphical syntax and mechanisms for graphical syntax composition. Sections 4 and 5 present conflict analysis and graphical language aspects. Section 6 describes our tool, and Section 7 reports on the evaluation. Finally, Section 8 compares with related research and Section 9 concludes.

2 LANGUAGE PRODUCT LINES

We start in Section 2.1 by introducing a running example. Then, Section 2.2 recalls the notion of language product line from [9], which covers only the abstract syntax of the language family.

2.1 Running example

Assume we would like to build a domain-specific language (DSL) [44] to model production systems. The DSL should enable the description of factories made of machines that process parts, and conveyors that transport parts between machines. There are many variations of this DSL in the literature, developed in isolation [5, 11–13, 15, 38, 42]. This variety may be because, depending on the model purpose, the DSL may need different capabilities, like:

- A basic version that can be simulated to understand how parts flow through a factory [11, 12, 15]. Fig. 1(a) shows a small model of this DSL variant. It contains a machine `ws1` processing a part `P1` and connected to a conveyor `cb1` that is transporting two parts `P2` and `P3`.
- A more realistic version of the DSL that supports modelling conveyor capacities and machine/conveyor breakdowns [38]. Fig. 1(b) shows a model using this language variant.
- Timed versions that take into account timing information (e.g., delays in machines and conveyors) or loss probabilities in connections, as depicted in Fig. 1(c) [12, 42].

As Figs. 1(a)–(c) show, different language variants may require different concepts in their abstract syntax, with impact in their concrete syntax. For example, machines in Fig. 1(b) need to have a boolean attribute to indicate if they are broken, and if so, the concrete syntax adds a cross to the machine representation.

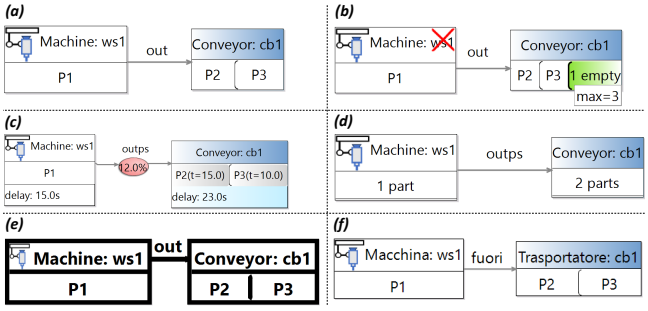

Figure 1: Some variants of the production systems family. (a) Basic machines and conveyors. (b) Possibly broken machines, conveyor capacities. (c) Time and loss probabilities. (d) Higher-level view of production system. (e) High-contrast syntax for accessibility. (f) Variant (a) in Italian.

Some variants may differ only in their concrete syntax, as they deal with the language *pragmatics* [41], i.e., how the language is used. According to the *cognitive fit* principle [33], different representations of information are suitable for different tasks or audiences. While modelling notations exhibit *visual monolinguism* [33], we would like to define concrete syntax variants to:

- Tailor the notation to different model usages or scenarios. For example, large production systems could be displayed using a more abstract view where parts are not represented individually, but only their number is displayed (cf. Fig. 1(d)).
- Adapt the language to its target users. For example, we may want to have accessible language variants – like Fig. 1(e), which shows a high-contrast variant of the language – or internationalisation – like Fig. 1(f), which shows a variant of the DSL in Italian. These variants would come in addition to the existing ones, i.e., any of the variants in Figs. 1(a)–(c) could be in English or Italian, with or without high-contrast.

Creating each language variant in isolation is very costly (generally, for n language features, there may be 2^n language variants) and can easily lead to errors and inconsistencies between variants. Instead of using a *clone-and-own* approach for creating DSL families [30], mechanisms are needed that allow the compact specification of both the abstract and concrete syntax of each member of the family. Such mechanisms should maximise the reuse of specification fragments to reduce the burden of building the DSL family, and should facilitate the extension of the family with new features and variants. Moreover, since a DSL family may have many variants, it is desirable to have analysis methods to detect potential problems related to conflicts in the definition of the concrete syntax.

Our goal is to provide such construction and analysis mechanisms. We base our proposal on the concept of modular *language product line* (LPL) presented in [9], which we summarise in the next section. Since this concept only covers the abstract syntax, Sections 3–5 will extend it to handle graphical concrete syntaxes and their analysis, which are the core contributions of this paper.

2.2 Language product lines: Abstract syntax

LPLs, as defined in [9], are made of modules, each encapsulating a meta-model fragment and declaring dependencies to a *dependency*

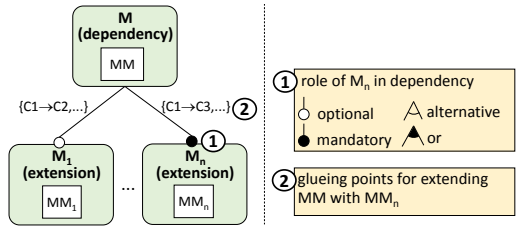

Figure 2: Structure of language modules within LPLs.

module (cf. Fig. 2). We often call the dependency module the parent, and the dependent modules its *extensions*.

Following standard practice in feature modelling [22], dependencies can be of four types: *mandatory*, *optional*, *or* and *alternative*. Dependencies state the conditions by which extension modules need to be selected in a language configuration when the parent module is selected as well. For *mandatory* dependencies, the extension module needs to be selected when the dependency module is selected. For *optional*, the extension module *can* be selected when the dependency module is selected. For *or*, one or more extension modules via this dependency type need to be selected. For *alternative*, exactly one extension module via this dependency type must be selected. In addition, each module needs to specify how to merge its meta-model fragment with the meta-model of its dependency, and graphs and morphisms are used for this purpose [14].

Def. 2.1 captures the notion of module and dependency type. Later, modules will be added an abstract syntax specification fragment (Def. 2.2) and a concrete syntax fragment (Def. 3.1).

Definition 2.1 (Module, adapted from [9]). A module is a tuple $M = \langle M_D, RO, \Psi \rangle$, where:

- M_D refers to a module, called dependency;
- $RO \in \{ALT, OR, OPT, MAN\}$ is the role of M in the dependency, one among *alternative*, *or*, *optional*, and *mandatory*;
- Ψ is a boolean formula that uses modules as variables.

M is called *top* if $M_D = M$ and $\Psi = true$. We use predicate $top(M)$ to identify top modules: $top(M) \iff M_D(M) = M \wedge \Psi(M) = true$.

In Def. 2.1, the Ψ formula specifies a cross-tree constraint stating conditions for modules to be included in a language configuration (the formulae of every module must evaluate to true in all valid configurations). Given a module M_i, we write $M_D(M_i)$ to denote the dependency of M_i, and similarly for the other components of M_i (i.e., RO, Ψ).

Definition 2.2 (Module with abstract syntax, adapted from [9]). A module with abstract syntax is a tuple $M_{AS} = \langle M_D, RO, \Psi, AS = \langle MM, IN_{AS} \rangle \rangle$, where:

- $\langle M_D, RO, \Psi \rangle$ is a module;
- AS is a tuple describing the abstract syntax, made of:
  - A meta-model MM;
  - An inclusion span IN_{AS} between MM and the meta-model of M's dependency: $IN_{AS} = MM \longleftarrow C_{AS} \longrightarrow MM(M_D)$.

We use $DEP^+(M_i)$ for the transitive closure of M_i's dependencies (i.e., its dependency, the dependency of its dependency, etc.), $DEP(M_i) = DEP^+(M_i) \setminus \{M_i\}$ for the transitive closure excluding itself (empty in top modules, and equal to DEP^+ in non-top modules), and $DEP^*(M_i) = DEP^+(M_i) \cup \{M_i\}$ for the reflexive transitive closure (i.e., including the module M_i as well). Typically, IN_{AS} is the identity inclusion for top modules.

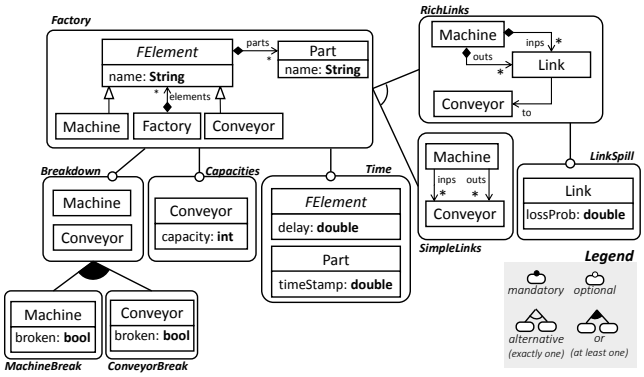

Figure 3: An abstract syntax LPL for a family of DSLs for modelling production systems.

Given the span $IN_{AS} = MM \xleftarrow{in_1} C_{AS} \xrightarrow{in_2} MM'$, we use predicate $map_{AS}(c_1, c_2)$ to denote that elements[1] $c_1 \in MM$ and $c_2 \in MM'$ are related, i.e., there is a node $n \in C_{AS}$ s.t. $in_1(n) = c_1$ and $in_2(n) = c_2$. We will use a similar notation for other spans, e.g., map_{CS} for a span IN_{CS}.

An LPL is just a set of modules where the dependencies form a tree, that is: (1) exactly one module is top, (2) the set is closed under the modules' dependencies, and (3) there are no dependency cycles. We use the term *abstract syntax LPL* if the modules only have abstract syntax. In the next section, we will extend modules so that they can also carry concrete syntax.

Definition 2.3 ((Abstract syntax) Language product line, adapted from [9]). A language product line $LPL = \{M_i\}_{i \in I}$ is a set of modules s.t.:

$$\exists_1 M_i \in LPL \cdot top(M_i) \wedge \tag{1}$$

$$\forall M_i \in LPL \cdot M_D(M_i) \in LPL \wedge \tag{2}$$

$$M_i \in DEP^+(M_i) \implies top(M_i) \tag{3}$$

If the modules in LPL have abstract syntax only, LPL is called *abstract syntax language product line*. We use $TOP(LPL)$ to denote the top module in LPL. Given a module $M_i \in LPL$, we define the sets $X(M_i) = \{M_j \in LPL \mid M_D(M_j) = M_i \wedge RO(M_j) = X)\}$, for $X \in \{ALT, OR, OPT, MAN\}$, to obtain the extension modules of M_i with role X.

Example 1. Fig. 3 shows an abstract syntax LPL for the running example. Its top module is `Factory`, which has five children modules: `Breakdown`, `Capacities`, `Time`, `SimpleLinks`, and `RichLinks`. Modules `Breakdown`, `Capacities` and `Time` are optional (each of them can be selected or not), while `SimpleLinks` and `RichLinks` are alternative (exactly one of them must be selected). That is, $OPT(\texttt{Factory}) = \{\texttt{Breakdown}, \texttt{Capacities}, \texttt{Time}\}$, and $ALT(\texttt{Factory}) = \{\texttt{SimpleLinks}, \texttt{RichLinks}\}$. If `Breakdown` is selected, then at least one of its children `MachineBreak` and `ConveyorBreak` must also be selected. Overall, the latter three modules add to the

[1] classes, references or attributes

DSL the ability to model machine and/or conveyor breakdowns. In turn, selecting `Capacities` adds capacity to conveyors, and `Time` adds a delay to machines and conveyors, and a timestamp to parts. Finally, the alternative set comprising `SimpleLinks` and `RichLinks` allows choosing between connecting machines and conveyors via references or `Link` objects, respectively. If `RichLinks` is selected, then `LinkSpill` can optionally be selected, which introduces a loss probability into link connectors.

In the figure, the modules contain meta-model fragments. The mapping between the meta-model elements in a module and its dependency are given by name. For instance, class `Machine` in module `Breakdown` is mapped to class `Machine` in module `Factory`.

Given an LPL, a configuration is a set of modules that is consistent with the dependencies of the LPL, and satisfies the formulae introduced by the modules, as Def. 2.4 states.

Definition 2.4 (Language configuration, from [9]). Given a product line LPL, a configuration $\rho \subseteq LPL$ is a set of modules s.t.:

$$TOP(LPL) \in \rho \wedge \tag{4}$$

$$M \in \rho \implies (\forall M_i \in MAN(M) \cdot M_i \in \rho \wedge \tag{5}$$

$$ALT(M) \neq \emptyset \implies \exists_1 M_i \in ALT(M) \cdot M_i \in \rho \wedge \tag{6}$$

$$OR(M) \neq \emptyset \implies \exists M_i \in OR(M) \cdot M_i \in \rho \wedge \tag{7}$$

$$M_D(M) \in \rho) \wedge \tag{8}$$

$$\bigwedge_{M_i \in LPL} \Psi(M_i)[true/\rho, false/(LPL \setminus \rho)] = true \tag{9}$$

We use $CFG(LPL)$ to denote the set of all configurations of LPL.

Example 2. The example LPL has 48 configurations, including ρ_a = {`Factory`, `SimpleLinks`}, ρ_b = {`Factory`, `Breakdown`, `Machine-Break`, `Capacities`, `SimpleLinks`}, and ρ_c = {`Factory`, `Time`, `RichLinks`, `LinkSpill`}. These three language configurations were the ones used to create the examples in Figs. 1(a)–(c).

Finally, given a language configuration ρ, we can derive a *product* meta-model MM_ρ by merging the meta-model fragments of each module $M \in \rho$. Def. 2.5 captures this intuition using the categorical notion of co-limit [26] (the result of merging a set of graphs by a set of mappings among them).

Definition 2.5 (Abstract syntax derivation, from [9]). Given an abstract syntax language product line LPL and a configuration $\rho \in CFG(LPL)$, a product meta-model MM_ρ is given by the co-limit object of all meta-models and spans in the set $\{IN_{AS}(M_i) = \langle MM(M_i) \longleftarrow C_{AS_i} \longrightarrow MM(M_D(M_i))\rangle \mid M_i \in \rho\}$.

Example 3. Fig. 4 shows the product meta-models obtained from configurations ρ_a, ρ_b and ρ_c.

Depending on the expressiveness allowed for meta-models within modules (e.g., inheritance, cardinalities, compositions), there may be product meta-models that are not well-formed (e.g., repeated attribute names, inheritance cycles). We refer to the static analysis techniques for well-formedness proposed in [10], which allow detecting such problems at the product line level.

3 GRAPHICAL CONCRETE SYNTAX FOR LPLS

Next, Section 3.1 extends modules with a graphical concrete syntax, Section 3.2 shows how to compose concrete syntax specification fragments, and Section 3.3 introduces graphical overriding and language variants that only change the concrete syntax.

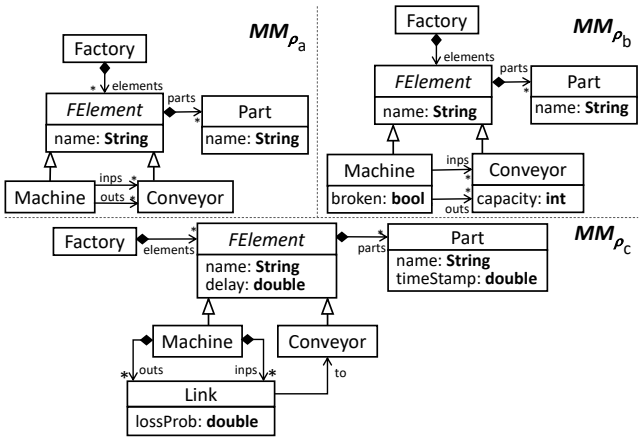

Figure 4: Product meta-models for ρ_a, ρ_b and ρ_c.

3.1 Language product lines: Concrete syntax

Graphical representations permit users to create and manipulate models visually. For this purpose, meta-models – defining the language abstract syntax – are enriched with a concrete syntax specification. We assume that such a concrete syntax specification has the form of a model, as it is customary in model-driven approaches to language definition [1]. While many technologies exist for creating graphical syntaxes (e.g., Sirius [40], GMF [17] or Eugenia [25]), our approach is agnostic with respect to the technology used.

To set our concepts, Fig. 5 shows a meta-model for graphical concrete syntaxes, inspired by Sirius. Class `Group` contains the viewpoints where different representations (i.e., views) can be defined. This way, a language can be represented in different forms, e.g., as a diagram (with nodes and edges), a table or a tree-form. This architecture is inspired by the Viewpoint and Views concepts from the ISO/IEC/IEEE-42010 standard [21].

The focus of this paper is on diagrammatic visualisations (i.e., `DiagramRepresentation`). These have a default `Layer` and may define additional ones. Objects in this meta-model can be represented as nodes (`NodeMapping`), edges (`EdgeMapping`) or containers (`ContainerMapping`). `EdgeMapping` has two subclasses that permit representing both references and objects as edges. Mappings have a style (the figure omits the styles for `EdgeMappings`), such as `Square` or `Ellipse`, and can specify a label, which can be a fixed String or the result of an OCL expression.

Four classes in the meta-model (`ContainerMapping`, `NodeMapping`, `ElementEdgeMapping`, `DiagramRepresentation`) refer to a meta-class in the language abstract syntax meta-model (`EClass` in the lower package). Hence, a concrete syntax specification is made of a model (an instance of the `ConcreteSyntax` package) plus a mapping to the abstract syntax (the `domainClass` references).

Next, we extend modules with a concrete syntax specification consisting of a concrete syntax model fragment – an instance of a concrete syntax meta-model, like the one in Fig. 5 – plus mappings

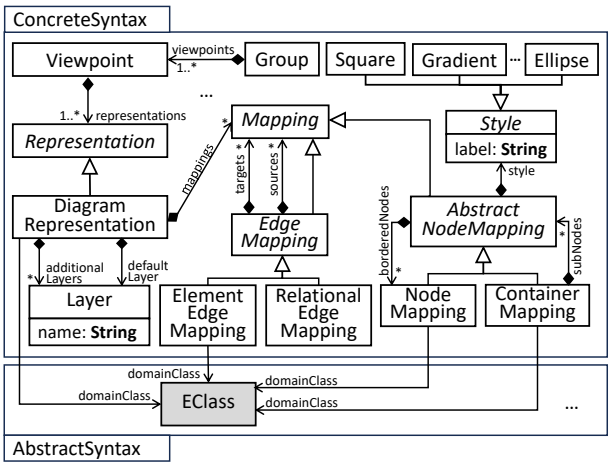

Figure 5: Simplified excerpt of meta-model for graphical concrete syntax.

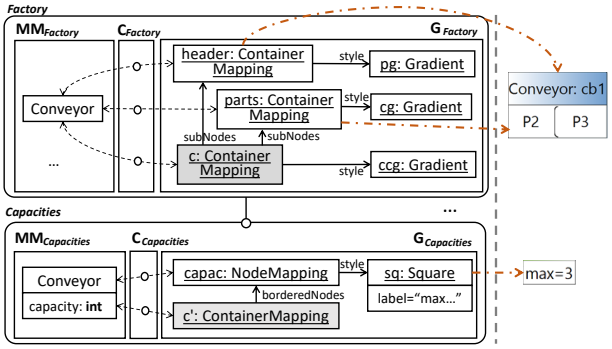

Figure 6: Two modules with concrete syntax.

of this model to the module's meta-model, and to the concrete syntax model fragment of the dependency module.

Definition 3.1 (Module with concrete syntax). A module with concrete syntax is a tuple $M_{CS} = \langle M, AS, CS = \langle G, IN_{AC}, IN_{CS}\rangle\rangle$, where:

- $M = \langle M_D, RO, \Psi\rangle$ is a module;
- $AS = \langle MM, IN_{AS}\rangle$ is a tuple describing the abstract syntax;
- CS is a tuple describing the concrete syntax, made of:
  - A model G, defining a graphical concrete syntax for MM;
  - A span IN_{AC} between G and the meta-model of M: $IN_{AC} = G \longleftarrow C_{AC} \longrightarrow MM$;
  - A span IN_{CS} between G and the concrete syntax model of M's dependency: $IN_{CS} = G \longleftarrow C_{CS} \longrightarrow G(M_D)$.

In the previous definition, span IN_{AC} maps the abstract syntax meta-model and the concrete syntax model, while span IN_{CS} models the extension of the concrete syntax defined in M's dependency.

Example 4. Fig. 6 shows two modules of the running example enriched with concrete syntax (simplified for readability). Module `Factory` specifies the concrete syntax of conveyors and machines (we omit the latter one for simplicity). Conveyors are represented by a `ContainerMapping` (object `c`, shaded) with two `ContainerMapping` subnodes: one for the blue header with the conveyor name (object `header`), and another for the parts' compartment (object `parts`). Each mapping has a `Gradient` style that specifies the colour and labels to be presented. For illustration, the right of the figure shows the concrete syntax rendering for a conveyor named `cb1`.

In the same figure, module `Capacities` extends module `Factory` by adding capacity to conveyors. Its concrete syntax model adds a new `NodeMapping` (`capac`) to show the capacity adjacent to the conveyor symbol (shaded). Note that the figure shows mapping IN_{AC} explicitly, while mapping IN_{CS} is implicitly given by equality of object names, but using primas to distinguish them. For example, `c'` in $G_{Capacities}$ is mapped to `c` in $G_{Factory}$.

A mapping between the concrete and abstract syntax is well-defined if every node in the concrete syntax is mapped to at most one class in the abstract syntax, as the next definition describes.

Definition 3.2 (Well-defined abstract-concrete mapping). A span $IN_{AC} = G \longleftarrow C_{AC} \longrightarrow MM$ between a concrete syntax model G and a meta-model MM is well-defined if $\forall c_1, c_2 \in MM, \forall e \in G$, $map_{AC}(e, c_1) \wedge map_{AC}(e, c_2) \implies c_1 = c_2$.

A module with concrete syntax requires its mappings IN_{AS}, IN_{AC} and IN_{CS} to be coherent. That is, if an element e' of the concrete syntax model extends an element e of the module's dependency, then the class e' is mapped to should extend the class e is mapped to. This is captured by Def. 3.3.

Definition 3.3 (Well-formed module with concrete syntax). A module with concrete syntax $M_{CS} = \langle M, AS, CS = \langle G, IN_{AC}, IN_{CS}\rangle\rangle$ is well-formed (wff) iff:

$$map_{CS}(e', e) \iff \exists c' \in MM, \exists c \in MM(M_D) \cdot \\ map_{AC}(e', c') \wedge map_{AC}(e, c) \wedge map_{AS}(c', c)$$

Example 5. Module `Capacities` in Fig. 6 is wff. This is so as we have map_{CS}(`c', c`), which is a valid mapping since: (1) class `Conveyor` in `Capacities` is mapped to `Conveyor` in `Factory` (i.e., map_{AS}(`Conveyor, Conveyor`)); (2) `c'` in $G_{Capacities}$ is mapped to `Conveyor` (map_{AC}(`c', Conveyor`)); and (3) `c` in $G_{Factory}$ is mapped to `Conveyor` (map_{AC}(`c, Conveyor`)). Instead, mapping `c` in $G_{Factory}$ to class `Part` would not yield a wff model.

3.2 Composing concrete syntax specifications

An LPL with concrete syntax (from now on, simply LPL) is an LPL made of modules with concrete syntax. Given a configuration ρ, a derivation yields a meta-model MM_ρ mapped to a concrete syntax model G_ρ, i.e. $IN_{AC_\rho} : G_\rho \longleftarrow C_\rho \longrightarrow MM_\rho$, as Def. 3.4 describes.

Definition 3.4 (Derivation). Given a language product line with concrete syntax LPL and a configuration $\rho \in CFG(LPL)$, we derive the span $IN_{AC_\rho} : G_\rho \overset{g_\rho}{\longleftarrow} C_\rho \overset{mm_\rho}{\longrightarrow} MM_\rho$ as follows:

(1) The meta-model MM_ρ is calculated as per Def. 2.5;
(2) The concrete syntax model G_ρ is calculated as the co-limit object of all models and spans in the set $\{IN_{CS}(M_i) = \langle G(M_i) \longleftarrow C_{CS}(M_i) \longrightarrow G(M_D(M_i))\rangle \mid M_i \in \rho\}$;

(3) The mapping model C_ρ is the minimal graph s.t. the set of functions $\{cs_i : C_{CS}(M_i) \to C_\rho \mid M_i \in \rho\}$ is jointly surjective, and $\forall c_j \in C_{CS}(M_j), c_k \in C_{CS}(M_k) \cdot g(c_j) = g(c_k) \wedge mm(c_j) = mm(c_k) \iff cs_j(c_j) = cs_k(c_k)$;
(4) The mappings g_ρ and mm_ρ are defined as $\bigcup_i g_i \circ cs_i^{-1}$ and $\bigcup_i mm_i \circ cs_i^{-1}$ respectively.

Example 6. Fig. 7 shows the derivation for a configuration containing modules `Factory` and `Capacities` (cf. Fig. 6). Model G_ρ is the glueing of models $G_{Factory}$ and $G_{Capacities}$ via mapping IN_{CS}. Mapping C_ρ is the union of $C_{Factory}$ and $C_{Capacities}$, where the elements with same mapping to MM_ρ and G_ρ are merged. On the concrete syntax, the effect of composing both modules is the addition of the box with the capacity to the conveyor representation.

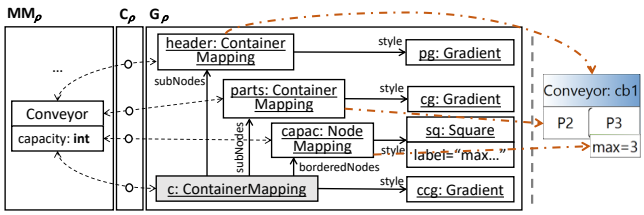

Figure 7: Derivation for the LPL excerpt of Fig. 6.

Next, we turn our attention to the correctness of the derived concrete syntax model and abstract-concrete syntax mapping. The next theorem states the correctness of the mapping.

Theorem 3.5 (Derivation mapping correctness). *Given a language product line LPL s.t. all modules are wff and all mappings IN_{AC_i} are well-defined, and a configuration $\rho \in CFG(LPL)$, then the mapping $IN_{AC_\rho} : G_\rho \xleftarrow{g_\rho} C_\rho \xrightarrow{mm_\rho} MM_\rho$ generated by Def. 3.4 is well-defined.*

Proof. Since each individual function cs_i, and the union of any two functions cs_j, cs_k may be non-injective, $\bigcup_i cs_i^{-1}$ would not be a function, but a relation. However, by Def. 3.3, $\bigcup_i mm_i \circ cs_i^{-1}$ is again a function. Let n be any node in C_ρ; cs_j and cs_k be any two functions; and $n_j \in C_{CS_j}$ and $n_k \in C_{CS_k}$ be any two nodes with $cs_j(n_j) = n = cs_k(n_k)$. Then we have that $mm_j(n_j) = mm_k(n_k)$ by condition 3 in Def. 3.4, and so $\bigcup_i mm_i \circ cs_i^{-1}$ is a function (and similarly for $\bigcup_i g_i \circ cs_i^{-1}$).

Then, we show that IN_{AC_ρ} is well-defined according to Def. 3.2. A node $n \in G_\rho$ could be mapped to two different classes in MM_ρ if:

(1) Some span IN_{AC} is not well-defined, which is not possible by the assumption of Theorem 3.5.
(2) Two nodes $n_i \in G_i$ and $n_j \in G_j$, mapped to different classes $c_i \in MM_i$ and $c_j \in MM_j$, were merged to a single node $n \in G_\rho$, and c_i and c_j were not merged to the same class in MM_ρ. Nodes n_i and n_j are merged if they are mapped to each other ($map_{CS}(n_i, n_j)$). Since all modules are wff, according to Def. 3.3, classes c_i and c_j should be mapped to each other as well, and thus merged by the co-limit construction.

Therefore, no node $n \in G_\rho$ can be mapped to two different classes, and so IN_{AC_ρ} is well-defined. □

Algorithm 1: Composing concrete syntax spec. objects

```
1  Function compose(po: CSObject, co: CSObject): CSObject
2      go ← po.clone()
3      forall f ∈ Class(co).fields do
4          if f.isMonoValued() then
5              if map_CS(po.f, co.f) then
6                  go.f ← compose(po.f, co.f)
7              else go.f ← co.f
8          else
9              forall fo ∈ co.f do
10                 if ∃fo' ∈ go.f · map_CS(fo', fo) then
11                     fo' ← compose(fo', fo)
12                 else go.f.add(fo)
13             go.f.trim(f.maxCardinality())
14     return go
15 Function compose(pf: Attribute, cf: Attribute): Attribute
16     gf ← pf.clone()
17     if pf.type is String then
18         gf.value ← pf.value.concat(cf.value)
19     else if pf.type is Numeric then
20         gf.value ← pf.value + cf.value
21     else if pf.type is Boolean then
22         gf.value ← pf.value ∨ cf.value
23     return gf
```

Given a configuration ρ, the abstract co-limit construction to build the concrete syntax model G_ρ does not detail how to compose the attribute values and references of objects. Moreover, it does not consider the cardinality of the composed fields (attributes and references), which is needed to ensure a correct resulting model. To tackle these aspects, Algorithm 1 describes how we compose a parent graphical object *po* with a child graphical object *co* (we use `CSObject` for any type in the concrete syntax model, i.e., any type in package `ConcreteSyntax` of Fig. 5). It iterates over all fields of *co* (line 3). If the field is monovalued (line 4), then its value in the child object overrides the value in the parent (line 7), unless the fields in the child and parent are mapped via map_{CS} (line 5), in which case they are composed recursively (line 6). If the field is multivalued (line 8), the same process is applied for each of its values (line 9): the values in the child are added to the parent (line 12), unless they are mapped (line 10), in which case they are composed recursively (line 11). Moreover, if the result of composing a multivalued field exceeds its maximum cardinality, it is trimmed (line 15). While function `compose` in lines 1–14 performs the composition of graphical objects, the homonymous function in lines 15–23 composes attribute values based on their type: it concatenates Strings (the child after the parent), adds numbers, and performs the *or* of booleans (composing enumerations is not allowed). In future work, we will generalise our algorithm to enable the provision of concrete composition algorithms for specific attribute types.

Example 7. Module `Time` adds a `timeStamp` to `Parts`, which should be displayed after the part name (cf. Fig. 1 (c)). Thus, module `Time` should not override the label that module `Factory` defines for `Parts`, but it needs to concatenate a value to it. Hence, both labels are mapped using map_{CS}. Then, when merging both styles, line 6 in Algorithm 1 invokes the function `compose` in line 15, which concatenates the String labels. By overloading function `compose` for

objects and attributes, we avoid the need to check the type of the mapped elements.

3.3 Overriding and graphical variants

Some variants within a language family may differ just in their concrete syntax. Such cases are handled by adding to the LPL modules that do not extend the abstract syntax, but only the concrete one.

Example 8. Language variant (d) in Fig. 1 only changes the concrete syntax to show the number of parts but do not depict the individual parts. This can be specified as Fig. 8(a) shows. The optional module `Summary` overrides the parts' compartments of `Machine` and `Conveyor` to display the number of parts. The container mappings `parts'` and `partsM'` in $G_{Summary}$ are composed with `parts` and `partsM` in $G_{Factory}$, but the `style` of the former ones prevails since `Summary` is a child module of `Factory` (cf. Fig. 8(b)). The OCL expressions in the labels of the gradient objects `mcg` and `cg` calculate the number of parts. The abstract syntax of module `Summary` includes classes `Machine` and `Conveyor` to enable their mapping to the concrete syntax $G_{Summary}$.

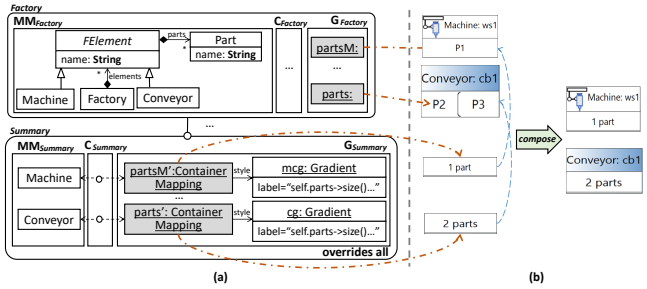

Figure 8: (a) Module `Summary` with concrete syntax variant. (b) Graphical symbols of modules `Summary` and `Factory`, and their composition (dotted blue arrows denote mapping IN_{CS}).

In this example, the concrete syntax of module `Summary` must prevail over any other, including that of `Capacities`, which adds to the parts' compartment of conveyors a grid with an indication of the free capacity (cf. Fig. 1(b)). This exemplifies a *feature interaction*: two modules, neither of which is a child of the other, compete to contribute concrete syntax elements. In our case, we need that the concrete syntax in module `Summary` is applied even if other modules that extend the compartments of `Machine` and `Conveyor` (like `Capacities`) are selected. To this aim, Def. 3.6 extends the concrete syntax of modules to enable overriding via an *override* specification OV.

Definition 3.6 (Concrete syntax overriding). Given a product line with concrete syntax LPL, a concrete syntax tuple with overriding is defined as $CS = \langle G, IN_{AC}, IN_{CS}, OV \rangle$, where:

- $\langle G, IN_{AC}, IN_{CS} \rangle$ is a concrete syntax tuple as in Def. 3.1;
- OV is a set of tuples $\langle G_O \subseteq G, M_O \subseteq LPL \rangle$ made of a (possibly empty) subgraph G_O of G, and a (possibly empty) subset M_O of modules of the LPL.

In Def. 3.6, G_O contains the elements of G that override others (all elements of G override any other if G_O is empty). Similarly, if M_O is empty, then all other modules in the LPL are overridden; otherwise, only the modules in M_O are overridden. If the set OV is empty, then there is no override specification.

Override specifications affect the composition procedure outlined in Algorithm 1 by discarding the concrete syntax of the elements within overridden modules.

Example 9. Module `Summary` in Fig. 8(a) defines the override specification $OV = \{\langle \emptyset, \emptyset \rangle\}$, so all objects in its concrete syntax model ($G_O = \emptyset$) override the graphical objects in the other modules ($M_O = \emptyset$). This is represented in the figure as "overrides all". Section 6 will describe a textual notation for override specifications.

4 ANALYSIS OF GRAPHICAL CONFLICTS

A possible mistake when specifying an LPL are modules that contribute concrete syntax elements that would be overridden by other (non-children) modules. Such modules would be in conflict because the result would depend on the order of composition.

Example 10. The left of Fig. 9 shows a schema of an example graphical conflict. For better intuition, the shapes (ellipses, rectangles) in the graphical part of modules represent the concrete syntax they define. Both M_1 and M_2 override the style for `Parts` in M. This means that, in configurations where M is selected but M_1 and M_2 are not, parts are represented as white rectangles (cf. right of Fig. 9); in configurations where M_1 is selected but M_2 is not, parts are ellipses; and when M_2 is selected but M_1 is not, parts are coloured rectangles. However, since elements can only have one style, when both M_1 and M_2 are selected, there is a conflict, since both modules override the style of parts differently.

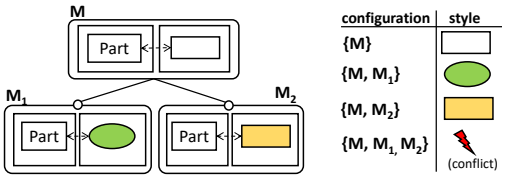

Figure 9: Example of graphical conflict. Selecting modules M, M_1 and M_2 yields a conflict in the graphical syntax.

We provide a method for detecting and signalling such situations, which then can be solved by explicitly including an override specification, as in Def. 3.6. Given an LPL, a set of modules $M_1, ..., M_n \in LPL$ are in potential graphical conflict if: (1) they all have concrete syntax elements $e_1, ..., e_n$ extending the same element e, (formally $\exists e_1 \in G(M_1), ..., \exists e_n \in G(M_n) \cdot map_{CS}(e_1, e) \wedge ... \wedge map_{CS}(e_n, e)$); (2) the modules are not direct or indirect children of each other ($M_1 \notin DEP^*(M_2) \wedge ... \wedge M_n \notin DEP^*(M_1)$); and (3) they may appear together in a configuration ($\exists \rho \in CFG(LPL) \cdot \{M_1, ..., M_n\} \subseteq \rho$). Once potential conflicts are detected, the method checks whether they are problematic: they override fields with maximum cardinality 1 (e.g., `AbstractNodeMapping.style`, cf. Fig. 5).

Given a set MC of modules in graphical conflict, the conflict is solved by setting override specifications such that, for any configuration ρ_i containing modules in MC ($\rho_i \cap MC \neq \emptyset$), there is exactly one module M_s that is not overridden, and all other modules are

overridden by another module. In practice, this amounts to defining an ordering among modules (i.e., override relations should define a total order in conflicting modules). In our implementation, if there is no override specification, the ordering is given by the insertion order of child modules and the depth-first traversal of the LPL.

Example 11. Fig. 9 contains the set $\{M_1, M_2\}$ of conflicting modules. The conflict can be solved by specifying that M_1 overrides M_2, or the other way round.

Finally, conflicts in override specifications may also occur, when there is a cycle of override relations among some of the modules in a configuration. This can be statically signalled by detecting cycles of override relations in the LPL, and then checking that the modules involved cannot appear together in a configuration.

5 GRAPHICAL LANGUAGE ASPECTS

Some graphical characteristics may be applicable across all variants of a language family, such as the use of high-contrast colours for accessibility (as in Fig. 1(e)), internationalisation (as in Fig. 1(f)), or the size of symbols. These characteristics affect all variants of a graphical language, and we refer to them as *graphical aspects*. Like aspects in programming languages [24], graphical aspects do not add localised features to the language, but they override certain graphical elements (e.g., colours, line widths, labels) defined by the LPL modules. Thus, they typically address cross-cutting concerns of a graphical language.

Technically, applying a graphical aspect creates new language variants by adding an optional child module to each module of the LPL. These new modules have constrains enforcing that, whenever the new child added to the top module is selected, the new children added to the other modules are also selected, and vice versa. Finally, the new optional modules modify the concrete syntax model of their parent as required. For example, an internationalisation aspect will translate the labels of symbols into a given language (cf. Fig. 1(f)).

Example 12. Fig. 10(a) shows the operation schema of applying the `HighContrast` graphical aspect on a fragment of the example LPL made of modules `Factory` and `Capacities`. The aspect adds the optional modules `HighContrast` and `Capacities-HighContrast` into the LPL. These modules override the concrete syntax model of their parent by changing the colour of symbols to black or white, and enlarging the font size of labels and the width of symbol lines. Moreover, the constraints in the added modules allow the aspect to be activated by just selecting the optional module under the top one (i.e., `HighContrast`). From the LPL definition, the standard feature model in Fig. 10(b) is generated. All optional modules introduced by the aspect – except the one for the top module – are set as hidden features. This way, when a user selects feature `HighContrast` in a configuration, the hidden feature `Capacities-HighContrast` will be automatically selected if `Capacities` is also selected.

In practice, aspects are specified by means of an Eclipse extension point, as we will see in the next section.

6 TOOL SUPPORT

We implemented these ideas in a tool named Capone-CS, available at https://github.com/antoniogarmendia/capone-graphical-pl.

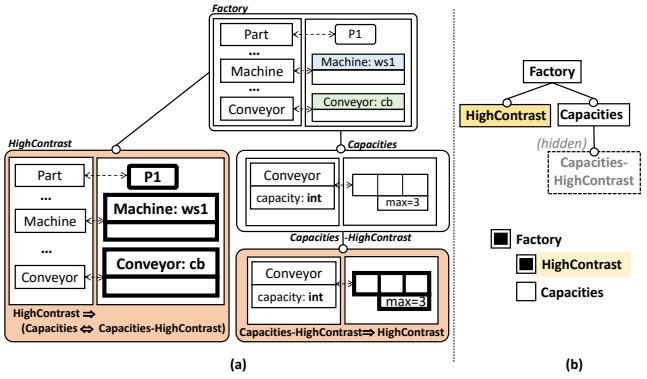

Figure 10: (a) Module injection by the `HighContrast` aspect. (b) Feature model with hidden features and configuration.

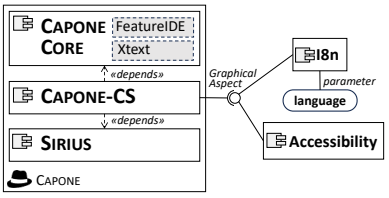

Figure 11: Architecture.

Fig. 11 shows the architecture of our tool. It is built atop Capone, an open-source Eclipse plugin made public in [9]. Capone uses the tool FeatureIDE [28] to handle the product line and the configurations, and Xtext [45] to provide a DSL to specify the modules. The figure shows our extension to Capone (Capone-CS), which relies on Sirius [40] to define the graphical concrete syntax models and generate the editors of the language variants. Capone-CS has an extension point enabling the external definition of graphical aspects. Extension point implementations may declare parameters (e.g., the internationalisation (i8n) aspect declares the language as a parameter) and must specify how to rewrite Sirius concrete syntax models (e.g., translating labels into the selected language).

Fig. 12 showcases Capone-CS. It provides a textual editor to define the modules of the LPL (label 1). We have extended the existing editor to refer to a Sirius *odesign* model (line 4 in the editor), write override specifications (line 5, cf. Def. 3.6), and compose attributes of graphical objects (e.g., labels) if needed (cf. Algorithm 1). For example, line 5 in the editor refers to two nodes in the *odesign* model by their identifier, which override their specification in all other modules. Label 2 in the figure shows two *odesign* concrete sytaxes from modules `Defects` and `MachineDefects`. The mapping IN_{CS} between *odesign* graphical objects is given by equality of identifiers. The mapping between attributes of graphical objects needs to be specified in the editor with label 1.

While an LPL is being defined, the designer can check for possible graphical conflicts, as described in Section 4. In the figure, both modules `Defects` and `MachineDefects` define a gradient shape style with different foreground color for `MachineContainer` (cf. label 2). As described in Section 4, a conflict arises because both modules may be selected together. This kind of conflicts are shown in the problems view (label 5). Currently, our implementation signals conflict sets of size 2. In the example, the problem would be solved by making one override the other. In addition, Capone-CS

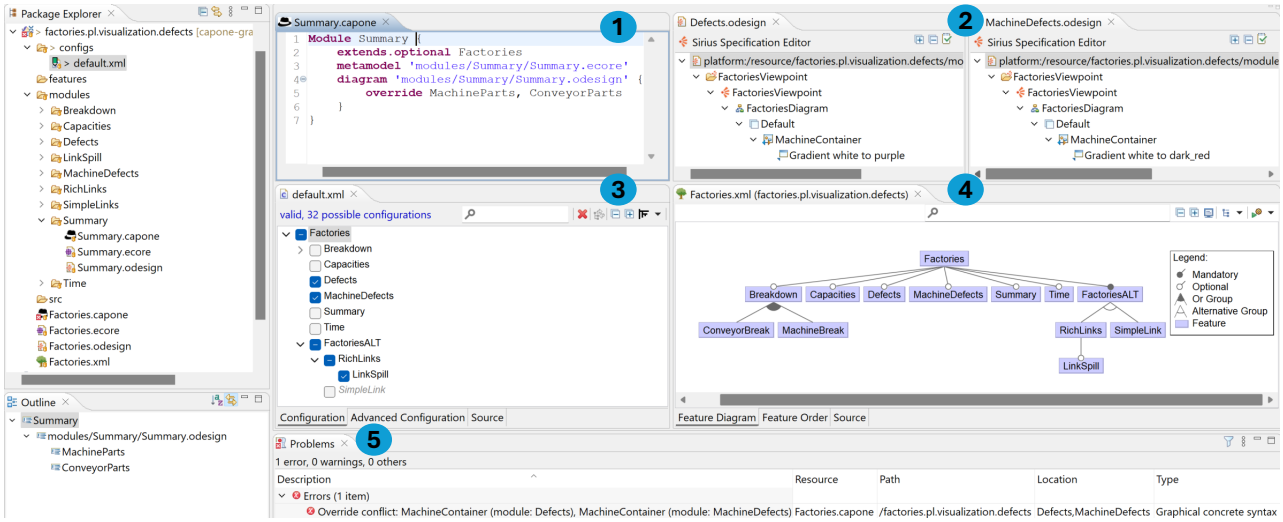

Figure 12: Capone-CS in action. (1) Specifying a module. (2) Sirius *odesign* concrete syntax models for two modules. (3) Feature model generated from the LPL. (4) Choosing a language configuration. (5) Problems view with graphical conflicts.

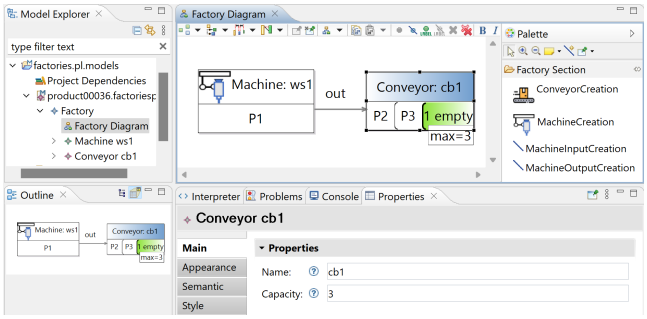

Figure 13: Generated editor for a language variant.

provides two implementations of the *Graphical Aspect* extension point for internationalisation and high-contrast accessibility. Their application to an LPL modifies its definition, as shown in Section 5.

Once the definition of the LPL is complete, Capone-CS permits generating a feature model that reflects the structure of its modules (label 3). Since the structure of an LPL is slightly more flexible than a standard feature diagram (e.g., module `Factory` in the running example has both optional and alternative children modules), the generated feature model may need to include intermediate features (e.g., `FactoriesALT`). Then, a dedicated editor permits selecting language configurations (label 4), and Capone-CS is able to build the products – the meta-model and Sirius *odesign* model – for them, as explained in Sections 2.2 and 3.

Finally, we have built a facility to extract a generated product into a dedicated Sirius project, to obtain a ready-to-use graphical editor. Fig. 13 shows a screenshot of the generated editor for one language variant. The editor features a palette with the classes and associations available in the variant, and properly handles other graphical elements that may or may not be present depending on the variant, such as layers.

7 EVALUATION

This section evaluates our approach. Firstly, we aim to assess the benefits of our proposal (LPLs) w.r.t. a direct approach where each language of a family is defined separately (case-by-case). Secondly, we aim at understanding the extensibility of our proposal, also taking a case-by-case approach as a baseline. Thus, our evaluation is designed to answer the following research questions (RQs):

RQ1: What is the specification size reduction of LPLs compared to a case-by-case approach?

RQ2: What is the typical effort for adding a feature to an LPL?

Experiment setup. We evaluated our approach on the running example and three case studies from the literature [9]: language families of networking DSLs, statecharts and Petri nets. The running example consists of the LPL in Fig. 2, plus module `Summary` in Fig. 8. Overall, the families comprise from 32 to 96 language variants.

Table 1 shows some data for these families, sorted by their number of configurations (column 2). The generated editors are available at https://github.com/antoniogarmendia/capone-graphical-pl.

Table 1: Comparison of LPLs with a case-by-case approach.

		LPL			Case-by-case		
Name	**#Cfg**	**#Mod (Overr. Rules)**	**AS size (avg)**	**CS size (avg)**	**AS size (avg)**	**CS size (avg)**	**% size red.**
`Networking`	32	8 (3)	35 (4.4)	115 (14.4)	488 (15.3)	1760 (55)	93.5%
`Statecharts`	48	12 (0)	46 (4.2)	145 (13.2)	720 (15)	2024 (42.2)	92.8%
`Petri nets`	64	13 (0)	52 (4)	188 (14.5)	864 (13.5)	4992 (78)	96.2%
`Factories`	96	10 (4)	43 (4.3)	191 (19.1)	1552 (16.1)	7296 (76)	97.4%

Results. To answer *RQ1*, we compared the specification size of the four LPLs with the case-by-case specification of each language variant. As Table 1 shows, the LPLs have between 8 and 13 modules,

with 0 to 4 overriding rules; the total size of the abstract syntax (classes, attributes and references in all meta-model fragments) ranges between 35 and 52, with an average of ≈4 elements per meta-model fragment; and the total size of the concrete syntax ranges from 115 to 191 graphical objects, with average concrete syntax model sizes between 13.2 and 19.1. In two cases, we had to define overriding rules (3 and 4). The number of rules is not large, considering the average number of graphical elements in each module (14.4 and 19.1). Instead, the case-by-case approach required creating between 32 and 96 meta-models and *odesign* models, the former with between 488 and 1552 elements in total (between 13.5 and 16.1 elements per meta-model in average), and the latter with between 1760 and 7296 graphical objects in total (between 42.2 and 78 objects per *odesign* model in average).

For *RQ2*, we looked at the typical specification sizes to create a language feature. Using LPLs, this entails adding one module with a small abstract syntax fragment (in our experiment, meta-models of less than 5 elements) and a concrete syntax model fragment (in the experiment, *odesign* models of less than 20 graphical objects). Instead, a case-by-case approach requires, in the worst case, creating n meta-models and *odesign* models (with n the number of existing features), the former with between 13.5 and 16.1 elements in our experiment, and the latter having between 55 and 78 objects.

Answering RQ1. For each considered language family, the specification size of its LPL is 1 or 2 orders of magnitude smaller than the corresponding case-by-case specification. Specifically, the LPLs resulted in a reduction in concrete syntax specification size of between 92.8% and 97.4%. The gain in specification size is typically greater the more variants a language family includes.

Answering RQ2. Extending an LPL with a new language variant requires substantially less specification effort than a case-by-case approach (one module vs. tens of meta-models and *odesign* models).

Threats to validity. The main threat to external validity is the limited number of case studies (4) of the experiment. We plan to perform a more complete experiment with bigger families in future work. Regarding internal validity, we (the authors) conducted the experiment. This does not bias our results as the evaluation is purely analytical, based on specification size. The later was used as a proxy for effort, but a study with developers would be needed to better understand the effort gain, and possible tool issues.

8 RELATED WORK

Next, we review works related to LPLs, modularity and aspects for modelling languages, and composition of graphical syntaxes.

Language product lines: Abstract syntax. Several authors have proposed product lines of meta-models. Perrouin *et al.* propose *feature model types*, an annotative product line of meta-models and their operations. Similarly, Guerra *et al.* [19] use an annotative approach to define meta-model product lines, and support well-formedness and instantiability analyses at the product line level. None of these approaches consider the concrete syntax of languages.

MMINT-PL [39] extends meta-models to support annotations at the model level and allow building product lines of models. Our approach works a meta-level higher, considering concrete syntax.

Language product lines: Concrete syntax. Similar to MMINT-PL, Verso [16] is a tool that injects variability in graphical modelling languages so that models built with those languages can be annotated with presence conditions. Instead, our approach supports the definition of product lines of graphical editors.

Concern-Oriented Language Development (COLD) [7] is a conceptual proposal that fosters reusability in language development by the notion of *language concern*. Concerns have fragments of abstract syntax, concrete syntax, and semantics, providing interfaces to support variability, customisation and use. Our modules are finer-grained, and our LPLs offer a variability interface but not a customisation interface. Moreover, our approach is fully realised, including mechanisms for merging the abstract and concrete syntaxes, analysis and graphical aspects.

Product lines of textual languages have been realised in Monticore [3, 4], Neverlang [43] and MetaDepth [31]. MontiCore components encapsulate textual syntax, integrity constraints, semantics (code generators), and can declare provided/required interfaces. Neverlang is a language workbench where modules encapsulate textual syntax and semantic actions on abstract syntax trees. In MetaDepth, meta-model templates define generic parameters with requirements specified via *concepts*. Templates can be composed via the parameters and a textual syntax model. In contrast, we target graphical syntax, proposing conflict analysis and graphical aspects.

Metrics and guidelines for de-composing textual languages were proposed by Cazzola *et al.* [6]. In future work, we aim at developing guidelines specific to meta-model-based graphical languages.

Composing graphical syntaxes. The platform Gromp [29] provides a language to compose multiple graphical languages and generate the corresponding editors. On a more conceptual level, Pedro *et al.* [36] propose rules for merging the abstract and concrete syntaxes of languages. Our approach uses composition mechanisms (cf. Sections 3.2 and 3.3), but our focus is on language families rather than on the composition of isolated languages.

Overall, to the best of our knowledge, ours is the first practical approach to define families of graphical modelling languages, combining language product lines and language engineering.

9 CONCLUSIONS AND FUTURE WORK

In this paper, we presented an approach to define families of graphical modelling languages. The approach is modular, enabling the feature-wise definition of a language family via modules that encapsulate abstract and concrete syntax. A language variant is created by selecting the desired features, and then the abstract and concrete syntax of the variant are automatically composed. The approach supports conflict analysis and *graphical syntax aspects* that inject general styles across LPLs, e.g., for accessibility. We presented an implementation and reported on an evaluation that yields substantial specification size reduction both when building a language family from scratch and when adding a new language feature.

We are currently developing methods to control the quality of the resulting graphical language variants. In particular, we are lifting the analysis of some Moody's principles for graphical notations [33] to the product line level. We also plan to enable the customisability of the composition algorithm. Finally, we will perform user studies, both about the generated editors and the specification method.

ACKNOWLEDGMENTS

This work was supported by the Spanish MICINN, with projects PID2021-122270OB-I00 and TED2021-129381B-C21.

REFERENCES

[1] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-driven software engineering in practice, Second edition.* Morgan & Claypool Publishers.
[2] Arvid Butting, Robert Eikermann, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2018. Controlled and extensible variability of concrete and abstract syntax with independent language features. In *VaMoS*. ACM, New York, NY, USA, 75–82.
[3] Arvid Butting, Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2021. *Compositional modelling languages with analytics and construction infrastructures based on object-oriented techniques—The MontiCore approach.* Springer International Publishing, Cham, 217–234.
[4] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. 2020. A compositional framework for systematic modeling language reuse. In *MoDELS*. ACM, 35–46.
[5] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. 2010. A UML/OCL framework for the analysis of graph transformation rules. *Softw. Syst. Model.* 9, 3 (2010), 335–357.
[6] Walter Cazzola and Luca Favalli. 2022. Towards a recipe for language decomposition: Quality assessment of language product lines. *Empir. Softw. Eng.* 27, 4 (2022), 82.
[7] Benoît Combemale, Jörg Kienzle, Gunter Mussbacher, Olivier Barais, Erwan Bousse, Walter Cazzola, Philippe Collet, Thomas Degueule, Robert Heinrich, Jean-Marc Jézéquel, Manuel Leduc, Tanja Mayerhofer, Sébastien Mosser, Matthias Schöttle, Misha Strittmatter, and Andreas Wortmann. 2018. Concern-oriented language development (COLD): Fostering reuse in language engineering. *Comput. Lang. Syst. Struct.* 54 (2018), 139–155.
[8] Loris D'Antoni and Margus Veanes. 2021. Automata modulo theories. *Commun. ACM* 64, 5 (2021), 86–95.
[9] Juan de Lara, Esther Guerra, and Paolo Bottoni. 2022. Modular language product lines: A graph transformation approach. In *MoDELS*. ACM, 334–344.
[10] Juan de Lara, Esther Guerra, and Paolo Bottoni. 2024. Modular language product lines: Concept, tool and analysis. *Software and Systems Modeling* to appear (2024), 29 pp. https://doi.org/10.1007/s10270-024-01179-9
[11] Juan de Lara and Hans Vangheluwe. 2008. Translating model simulators to analysis models. In *FASE (LNCS, Vol. 4961)*. Springer, 77–92.
[12] Juan de Lara and Hans Vangheluwe. 2010. Automating the transformation-based analysis of visual languages. *Form. Asp. Comput.* 22, 3 (may 2010), 297–326.
[13] Francisco Durán, Steffen Zschaler, and Javier Troya. 2012. On the reusable specification of non-functional properties in DSLs. In *SLE (LNCS, Vol. 7745)*. Springer, 332–351.
[14] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. *Fundamentals of algebraic graph transformation.* Springer.
[15] Hartmut Ehrig and Claudia Ermel. 2008. Semantical correctness and completeness of model transformations using graph and rule transformation. In *ICGT (LNCS, Vol. 5214)*, Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer (Eds.). Springer, 194–210.
[16] Antonio Garmendia, Manuel Wimmer, Esther Guerra, Elena Gómez-Martínez, and Juan de Lara. 2020. Automated variability injection for graphical modelling languages. In *GPCE*. ACM, New York, NY, USA, 15–21.
[17] GMF. (last accessed in March 2024). https://eclipse.dev/modeling/gmp/.
[18] Esther Guerra and Juan de Lara. 2018. On the quest for flexible modelling. In *MoDELS*. ACM, 23–33.
[19] Esther Guerra, Juan de Lara, Marsha Chechik, and Rick Salay. 2022. Property satisfiability analysis for product lines of modelling languages. *IEEE Trans. Softw. Eng.* 48, 2 (2022), 397–416.
[20] Felienne Hermans. 2020. Hedy: A gradual language for programming education. In *ICER*. ACM, 259–270.
[21] ISO/IEC/IEEE 42010 2022. Systems and software engineering – Architecture description. https://www.iso.org/standard/74393.html.
[22] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-oriented domain analysis (FODA) feasibility study.* Technical Report CMU/SEI-90-TR-021. Carnegie Mellon University.
[23] Nadine Kashmar, Mehdi Adda, and Mirna Atieh. 2020. From access control models to access control metamodels: A survey. In *FICC (LNNS, Vol. 70)*. Springer, 892–911.
[24] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP'97 (LNCS, Vol. 1241)*. Springer, 220–242.
[25] Dimitrios S. Kolovos, Louis M. Rose, Saad bin Abid, Richard F. Paige, Fiona A. C. Polack, and Goetz Botterweck. 2010. Taming EMF and GMF using model transformation. In *MoDELS (LNCS, Vol. 6394)*. Springer, 211–225.
[26] Saunders Mac Lane. 1971. *Categories for the working mathematician.* Springer.
[27] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. 2013. What industry needs from architectural languages: A survey. *IEEE Trans. Soft. Eng.* 39, 6 (2013), 869–891.
[28] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering software variability with FeatureIDE.* Springer.
[29] Ivan Melo, Mario E. Sánchez, and Jorge Villalobos. 2013. Composing graphical languages. In *GlobalDSL@ECOOP*. ACM, 12–17.
[30] David Méndez-Acuña, José Angel Galindo, Benoît Combemale, Arnaud Blouin, and Benoit Baudry. 2017. Reverse engineering language product lines from existing DSL variants. *J. Syst. Softw.* 133 (2017), 145–158.
[31] Bart Meyers, Antonio Cicchetti, Esther Guerra, and Juan de Lara. 2012. Composing textual modelling languages in practice. In *MPM@MoDELS*. ACM, New York, NY, USA, 31–36.
[32] Hafedh Mili, Guy Tremblay, Guitta Bou Jaoude, Eric Lefebvre, Lamia Elabed, and Ghizlane El-Boussaidi. 2010. Business process modeling languages: Sorting through the alphabet soup. *ACM Comput. Surv.* 43, 1 (2010), 4:1–4:56.
[33] Daniel L. Moody. 2009. The "physics" of notations: Toward a scientific basis for constructing visual notations in Software Engineering. *IEEE Trans. Software Eng.* 35, 6 (2009), 756–779.
[34] T. Murata. 1989. Petri nets: Properties, analysis and applications. *Proc. IEEE* 77, 4 (1989), 541–580.
[35] L. Northrop and P. Clements. 2002. *Software product lines: Practices and patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[36] Luis Pedro, Matteo Risoldi, Didier Buchs, Bruno Barroca, and Vasco Amaral. 2009. Composing visual syntax for domain specific languages. In *HCI (LNCS, Vol. 5611)*. Springer, 889–898.
[37] Gilles Perrouin, Moussa Amrani, Mathieu Acher, Benoît Combemale, Axel Legay, and Pierre-Yves Schobbens. 2016. Featured model types: Towards systematic reuse in modelling language engineering. In *MiSE@ICSE*. ACM, New York, NY, USA, 1–7.
[38] José Eduardo Rivera, Esther Guerra, Juan de Lara, and Antonio Vallecillo. 2008. Analyzing rule-based behavioral semantics of visual modeling languages with Maude. In *SLE (LNCS, Vol. 5452)*. Springer, 54–73.
[39] Alessio Di Sandro, Ramy Shahin, and Marsha Chechik. 2023. Adding product-line capabilities to your favourite modeling language. In *VaMoS*. ACM, 3–12.
[40] Sirius. (last accessed in March 2024). https://www.eclipse.org/sirius/.
[41] Harald Störrle. 2019. Modeling moods. In *MoDELS Companion*. IEEE, 468–477.
[42] Javier Troya, Antonio Vallecillo, Francisco Durán, and Steffen Zschaler. 2013. Model-driven performance analysis of rule-based domain specific visual models. *Inf. Softw. Technol.* 55, 1 (2013), 88–110.
[43] Edoardo Vacchi and Walter Cazzola. 2015. Neverlang: A framework for feature-oriented language development. *Comput. Lang. Syst. Struct.* 43 (2015), 1–40.
[44] Andrzej Wasowski and Thorsten Berger. 2023. *Domain-specific languages - Effective modeling, automation, and reuse.* Springer.
[45] Xtext. 2022. https://www.eclipse.org/Xtext/.