Adaptive modelling languages: Abstract syntax and model migration

JUAN DE LARA, Universidad Autónoma de Madrid, Spain
ESTHER GUERRA, Universidad Autónoma de Madrid, Spain

Modelling languages are heavily used in many disciplines, including software engineering. However, current languages are *rigid*, since they do not get adapted to fit the users' expertise, the modelling task, or the usage platform. This may turn some languages unsuitable for a range of users (from unexperienced to experts), goals (from informal discussion to precise specification) and platforms (from desktops to mobile phones). We claim that making languages *adaptive* to the modelling scenario would alleviate these issues and help simplifying recurring tasks such as language evolution or interoperability between the languages of a family.

In this paper, we propose the new notion of *adaptive modelling language*. This concept combines meta-modelling and product lines to support variants of a given language, and encompasses contextual conditions triggering language reconfigurations, and mechanisms for model migration across the language variants. The paper presents a theory and its realisation atop the Eclipse Modeling Framework. Our tool includes an Eclipse workbench to specify adaptive languages and produce Eclipse modelling editors with adaptation support. We report on an evaluation demonstrating the advantages of using our framework to express migrations across the variants of adaptive languages, which moreover have generally fast execution times.

CCS Concepts: • **Software and its engineering → Domain specific languages**; *Software design engineering*.

Additional Key Words and Phrases: modelling language engineering, flexible modelling, model transformation, graph transformation, model migration, software product lines.

ACM Reference Format:
Juan de Lara and Esther Guerra. 2024. Adaptive modelling languages: Abstract syntax and model migration. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (October 2024), 54 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Modelling is pervasive in software engineering [46] and essential in model-driven engineering (MDE) [12]. Models are built using modelling languages, which can be either general-purpose, like the UML [74], or domain-specific languages (DSLs) tailored for a domain and task [41].

Modelling can serve a variety of purposes, from informal discussions to precise software specification for code generation or verification [29, 83]; it is performed by users with different expertise, from novices to experts [11]; and it is supported on a variety of IDEs and devices, from computers with keyboard and mouse, to smart mobile and virtual reality devices [13, 82], or interactive multi-touch displays and whiteboards [48, 76]. However, most modelling languages are *rigid*, in the sense that they cannot be adapted to the modelling task, the target user or the modelling platform. This may hinder the language usage for a range of users or scenarios.

To alleviate the rigidity of current modelling languages, we propose the new notion of *adaptive modelling language*. An adaptive language is *flexible*, since it permits choosing between variants of

---

Authors' addresses: Juan de Lara, Universidad Autónoma de Madrid , Madrid, Spain, Juan.deLara@uam.es; Esther Guerra, Universidad Autónoma de Madrid , Madrid, Spain, Esther.Guerra@uam.es.

1049-331X/2024/10-ART1 $15.00
https://doi.org/XXXXXXX.XXXXXXX

the language that can be a better fit to different usage scenarios. Moreover, since the modelling needs may change over time, the language variant a model is being defined with can be modified dynamically. The language variant can be explicitly chosen by the user, or reconfigurations may be triggered when certain conditions specified by the language designer are met. The latter conditions may pertain, e.g., the usage or not of certain language primitives, the selection of a modelling phase in a process model, the level of expertise of the user (which can be either stated explicitly or induced automatically), the device the modelling tool is running on, or patterns found in the model, among others. Moreover, language variants are not isolated, but an adaptive language provides interoperability between them by the automated migration of models.

We have realised these ideas on a framework for creating adaptive languages based on the principles of MDE and software product lines [57]. Product lines make it possible to define highly configurable languages with hundreds or thousands of variants in a compact way [30]. In such a setting, a *naive approach* that creates migration transformations between each two language variants becomes unfeasible. Therefore, our framework reduces this burden by incorporating techniques to compose automatically those transformations out of small modules called *adapters*. In this paper, we present both a theory and a practical implementation within Eclipse, and evaluate the feasibility and advantages of our proposal based on six case studies.

Overall, this paper makes the following contributions: (i) the *novel notion* of adaptive modelling language, along with application scenarios; (ii) a *theoretical formulation* that encompasses a product line of modelling languages, language adapters that are composed on the fly to assemble migration transformations between language variants, and flexible language adaptation trigger mechanisms; (iii) techniques to *analyse* the compatibility and correctness of adapters; (iv) a *practical implementation* atop the Eclipse IDE; and (v) an *evaluation* that shows the benefits of expressing migrations across a language family using our notion of adaptive language. In particular, the evaluation aims at answering the following research questions (RQs):

RQ1: *How feasible is it to specify adaptive languages in practice?*
RQ2: *How efficient is the adaptation process at runtime?*

In turn, RQ1 is decomposed into the next follow-up RQs, which analyse the specification size reduction achieved by the use of adapters for defining migrations across language variants:

RQ1.1: *What is the specification size reduction of using adapters w.r.t. a naive approach?*
RQ1.2: *What is the specification size reduction achieved by the sequential composition of adapters?*

In the following, Section 2 overviews adaptive modelling languages and their usage scenarios. Next, Section 3 gives background on meta-models, models, graph transformation, and language product lines. Then, Section 4 defines a theory for adaptive modelling languages, with mechanisms (called adapters) to reduce the effort needed to define migration transformations between language variants. Sections 5 and 6 present techniques to compose and analyse adapters. Next, Section 7 describes tool support, and Section 8 evaluates our proposal. Finally, Section 9 compares with related work, and Section 10 presents the conclusions and lines for future work. Appendix A provides details of the theory, including proofs of the lemmas, propositions and theorems.

2 OVERVIEW AND SCENARIOS OF ADAPTIVE MODELLING LANGUAGES

This section provides an intuitive notion of adaptive modelling languages, describing scenarios where they are useful (Section 2.1). Then, it overviews our approach to the definition and use of adaptive modelling languages, explaining briefly its building blocks (Section 2.2).

2.1 Intuition and Usage Scenarios

A modelling language is made of abstract syntax (the primitives of the language, their properties and relations), concrete syntax (how the primitives are rendered, typically graphically or textually), and semantics (what models mean, often realised via code generators or simulators). These language parts are typically fixed and unchanging. Instead, we define an adaptive modelling language as:

> *A language with variants, along with mechanisms to trigger dynamic adaptations between them – based on the modelling context – and for automated model migration across the language variants.*

Supporting a coordinated use of variants of a language and automating the migration of models across those variants is useful in several scenarios, like:

- *Languages that adapt to the user.* The *cognitive fit* principle for visual language design [51] states that users with different expertise in a language can benefit from different language versions. Beginners could use simple language variants, which become more complete as they learn. For instance, novice users of UML could use simpler versions without composition, inheritance or navigation decorators in associations, and experienced users could use more sophisticated UML versions. This can be useful in education, where increasingly sophisticated language versions (called gradual languages [32]) can guide the learning process[1], or in lowcode platforms [62], which need to support citizen developers with a diverse range of skills. While user adaptation is a desirable language feature, most notations exhibit *visual monolinguism*, as they use a single visual notation for all purposes [51]. Thus, the design of user-oriented language variants must consider their concrete syntax representation, as well as their abstract syntax.
- *Languages that adapt to the IDE.* According to Moody [51], different representational media for the modelling task may require the design of different language variants. For example, devices with a reduced screen size (e.g., mobile devices [13]) or sketch-based interaction (e.g., digital whiteboards [48] or tablets [44]) may employ simple language variants, while traditional computers with wide screens, mouse-based interaction, and high computational power can use more complete languages. Likewise, different variants of a concrete syntax (e.g., tabular vs graphical) could be used to maximise the information presented in reduced spaces.
- *Languages that adapt to the process.* In software engineering, early development phases benefit from informal modelling as a vehicle for discussion and problem understanding. As a project progresses, precise models may be needed to enable system analysis or code generation. To transition between both operation modes, the discussion phase could rely on permissive variants of a modelling language, and later phases could use more constrained variants [29].
  Figure 1 shows an example of this scenario that will be used to illustrate our proposal throughout the paper. In the figure, a modelling process goes through three stages: *analysis*, *design* and *detailed design*. Each stage uses a different variant of class diagrams. The analysis phase employs a simple variant without methods, compositions or aggregations. The design phase uses another variant that considers these elements. Since the implementation language is Java, the detailed design phase uses single inheritance and interfaces. The figure depicts that, whenever the phase changes, a model adaptation occurs, which transforms the current model into the language variant of the next phase.
- *Language/model co-evolution.* In this scenario [78], a language evolves into a new version, and the existing models must be migrated to remain compatible with the new version. This is a special case of adaptive languages where each language variant corresponds to a different language

---

[1] The term gradual language was proposed in [32], where different versions of Python were created to help children in learning programming.

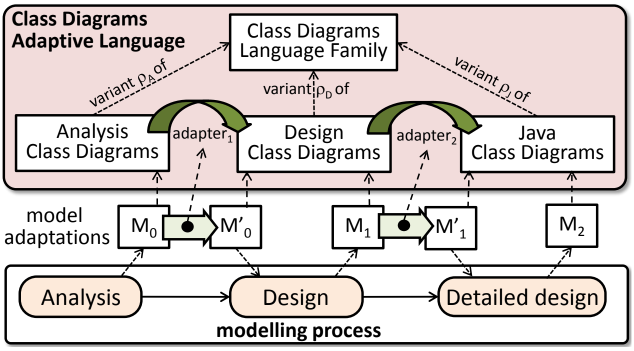


Fig. 1. Class diagrams as an adaptive language that adapts to the modelling phase.

version. This way, the model adaptation mechanisms of adaptive languages can be used for co-evolving models.

- *Language families.* A language family is a group of related languages, and can be included within the usage applications of adaptive languages. Examples of language families include the more than 120 variations of architectural languages reported in [47], and the many variants of Petri nets [52], access control languages [40] and symbolic automata [20]. An interesting scenario here is to start modelling with a language variant of the family (e.g., black and white Petri nets), and then switching to a more expressive variant as modelling progresses and new needs arise (e.g., when the modeller needs to use inhibitor arcs).

All these scenarios require being able to migrate models between the language variants employed. In Figure 1, the adaptive language provides facilities to migrate from the analysis to the design language variant, and from the design to the detailed design variant. Even though this example considers three variants only, an adaptive language may comprise many. Hence, mechanisms that avoid the explicit creation of migration transformations between each language variant would be most helpful. Our notion of adaptive language includes mechanisms – called adapters – to specify the migration in "pieces", which are combined depending on the source and target language variant.

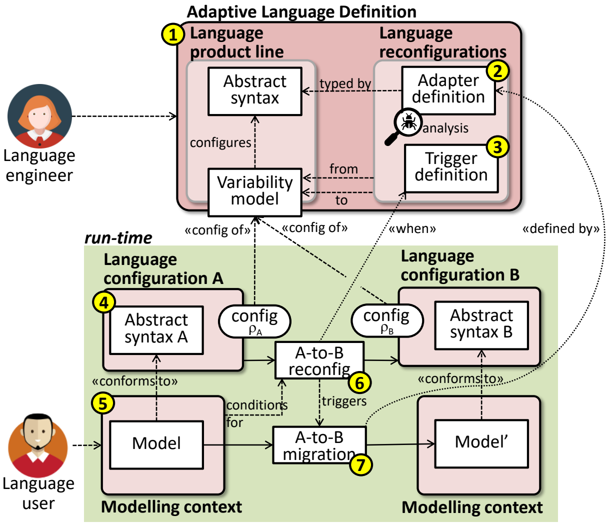
Another issue is the adaptation trigger. In the simplest case, the user selects the new language variant, causing the adaptation (i.e., the migration) of the current model to the new language variant. In addition, we foresee scenarios where adaptation is triggered automatically based on the language features (un-)used by the current user, or on the preferred language variants of like-minded users (i.e., using collaborative filtering recommendation techniques [3, 71]). Our notion of adaptive language considers a general triggering mechanism that can accommodate these scenarios.

This paper focuses on the abstract syntax of adaptive languages, as it is the basis for defining the concrete syntax and semantics. However, adapting the concrete syntax is also meaningful to provide more or less sophisticated visualisations depending on the screen size (e.g., to accommodate the cognitive fit principle [51]), or even moving between graphical, textual, tabular, or conversational syntaxes [54]. Similarly, adapting the semantics can also be of interest, e.g., to select between the different semantics of Statecharts [77]. These two topics are left for future work.

Overall, the MDE community has done extensive work in language syntax and semantics, but their *pragmatics* – how languages are used – is not so explored [69]. Adaptive modelling languages aim at making pragmatics a first-class citizen in software language engineering.

2.2 Overview

Figure 2(a) shows the main ingredients of our approach. The specification of an adaptive language is responsibility of a language engineer. It involves defining a language product line (label 1 in the figure) and a set of language reconfigurations that include a set of model migration rules (label 2) and triggers stating the circumstances for reconfiguration between language versions (label 3).



(a)

Concept	Description	Def.
Feature model	Description of the variability of the language family, via features and relations between them	3.13
Feature configuration	A set of features consistent with the feature model	3.15
150% meta-model (150MM)	Abstract syntax of all language variants in an overlapped way	3.17
Presence condition (PC)	Boolean formula over the features, which annotates elements of the 150MM	3.17
Language product line (LPL)	A family of languages, made of a feature model and a 150MM with PCs	3.17
Diff	A tuple specifying feature changes and feature invariants	4.1
Graph transformation system (GTS)	A set of rules, with an execution policy, and a meta-model	3.11
Language adapter	A GTS with a diff stating feature conditions for its selection	4.9
Adaptive language	An LPL with a set of adapters	4.15
Migration transformation between configs	A GTS built by selecting adapters compatible with the source and target configurations	4.17
Triggered adaptive language (TAL)	An adaptive language with trigger conditions for adaptation	4.19
Contextual adaptive model	Model with context, conforming to one configuration of the TAL	4.22

(b)

Fig. 2. (a) Schema of our approach to define and use adaptive modelling languages. (b) Key concepts.

A language product line is a compact specification of a set of language variants. As previously stated, we focus on the abstract syntax of the language only. Thus, in our approach, a language product line is made of: a meta-model specifying the abstract syntax of all language variants in an overlapped way (so-called negative variability [65]), a variability model describing the features of the allowed language variants (so-called language configurations), and conditions on the presence or absence of the meta-model elements in each language configuration[2]. Section 3.3 will introduce language product lines.

A reconfiguration from a source to a target language configuration is defined by means of adapters. These are sets of transformation rules that specify model migration piecewise. As Sections 4.2 and 5 will show, adapters are defined based on the available language features, and then get automatically selected and composed depending on the features of the source and target language configurations. We use graph transformation [28] to express model transformations, but other approaches could be used as well. In addition, we propose techniques for the language engineer to analyse the local correctness and compatibility of adapters, and the reachability of configurations using meaningful migration transformations (cf. Section 6).

Triggers are conditions evaluated on the current model or its context, specifying when an adaptation into a new language variant should occur. Context models [13] may include information

---

[2] Our approach can be easily extended to include a model-based definition of the concrete syntax of a language family (e.g., using Sirius *odesign* models [66]). Selecting a configuration would then produce the abstract and concrete syntax definitions for the language variant. At runtime, a framework that interprets the model-based concrete syntax definitions (e.g., Sirius) would enable replacing the concrete syntax based on the language variant. We will consider concrete syntax in future work.

regarding the model construction history or the modelling activity (e.g., properties of the modelling device or the current user, time of modelling, position of the device). Some adaptive languages may also allow the user to freely select the desired target language variant, while ensuring that such a language reconfiguration is allowed. Section 4.4 will explain triggers.

The bottom of Figure 2(a) depicts the usage schema of an adaptive language at run-time. A user is editing a model with the language variant A, given by the configuration ρ_A of the adaptive language (label 4). The environment is monitoring the model and the context of interest (label 5). When a reconfiguration trigger into the language variant B occurs (label 6), a migration transformation is composed on the fly out of the defined adapters (label 7). This transformation is executed, so that the model is migrated and the user can continue modelling using the language variant B.

As a reference for the reader, Figure 2(b) provides a brief description of the key concepts that will be introduced throughout the paper, and a pointer to their formal definition.

3 PRELIMINARIES

This section provides some background for the notion of adaptive modelling language. Section 3.1 starts defining the concepts of meta-model, model and model mapping. Section 3.2 introduces graph transformation, as we will use it to express migrations across language variants. Then, Section 3.3 presents language product lines, over which adaptive languages are defined.

3.1 Models and Meta-models

Our theory requires a notion of model and meta-model, for which we use a representation based on graphs. For convenience, we use a slight simplification of the notion of E-Graph defined in [28] to represent both models and meta-models.

Definition 3.1 (E-Graph). An *E-Graph* $G = \langle V, D, E, A, src, tar, owner, val \rangle$ consists of the sets:

- V of graph vertices, D of data values, E of graph edges, and A of attributes

and the functions:

- $src\colon E \rightarrow V$, $tar\colon E \rightarrow V$ providing a source and target vertex to each graph edge
- $owner\colon A \rightarrow V$, $val\colon A \rightarrow D$ providing an owner vertex and a value to each attribute

Remark 3.2. Given an E-Graph G, we write V, E, A to denote its sets of vertices, edges and attributes, when no confusion can arise. When considering several graphs (e.g., M, MM) then we use subindices for these sets (e.g., M_V, M_E, M_A, MM_V, MM_E, MM_A).

Models can be encoded as E-Graphs by using the set V to represent the objects, A the attributes, D the attribute values, and E the links between objects. E-Graphs are often enriched with an algebra over a data signature [64] that describes the attribute data types (string, integer, boolean). Such graphs are called *attributed graphs*, and the set D is then defined as the union of the carrier sets of the algebra [28]. Meta-models can be encoded using the same structure, but in this case, attributes specify a data type and do not hold values. This way, meta-models are attributed graphs over a final signature, where the carrier set of each sort has just one element [28]. Richer meta-model formalisations have been proposed, e.g., considering inheritance [21] or cardinalities [72]. Instead, we opt for a simpler formulation as it serves better to illustrate our ideas.

Example 3.3. Figure 3(a) depicts a meta-model MM and a model M as per Definition 3.1. The meta-model MM contains one vertex Class with a reflexive edge parent and an attribute name of type String (being String the only element of MM_D). The model M has two vertices (person and emp) connected via an edge parent and giving values to attribute name ("Person" and "Employee"). Figure 3(b) depicts the same meta-model and model using the UML notation, which we will use from now on.

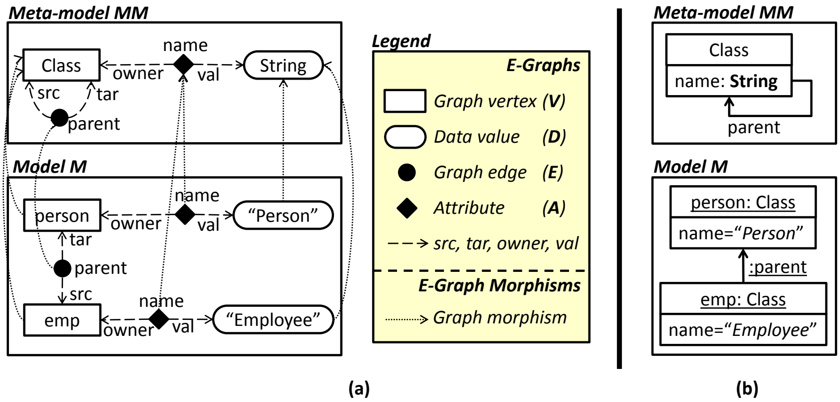


Fig. 3. A model M typed over meta-model MM using (a) Definitions 3.1 and 3.4, and (b) the UML notation.

We use graph morphisms [28] to express relations between graphs, like the type relationship between model M and meta-model MM in Figure 3(a). A graph morphism is a tuple of commuting functions mapping the sets V, D, E and A in both graphs.

Definition 3.4 (E-Graph morphism). Given two E-Graphs $G_i = \langle V_i, D_i, E_i, A_i, src_i, tar_i, owner_i, val_i \rangle$ for $i \in \{1, 2\}$, an *E-Graph morphism* $f \colon G_1 \longrightarrow G_2 = \langle f_V, f_D, f_E, f_A \rangle$ is made of a tuple of set functions $f_X \colon X_1 \longrightarrow X_2$ (for $X \in \{V, D, E, A\}$) commuting with functions *src*, *tar*, *owner*, and *val*, i.e., $f_V \circ src_1 = src_2 \circ f_E$, $f_V \circ tar_1 = tar_2 \circ f_E$, $f_V \circ owner_1 = owner_2 \circ f_A$, and $f_D \circ val_1 = val_2 \circ f_A$.

Example 3.5. Figure 3(a) shows an E-Graph morphism $f \colon M \longrightarrow MM$, which maps person and emp to Class (i.e., $f_V(\text{person}) = f_V(\text{emp}) = \text{Class}$). It is valid according to Definition 3.4 since all functions commute. For example, the source vertex of parent in M is emp, which is mapped to Class (i.e., $f_V(src_M(\text{parent})) = \text{Class}$); and commutatively, we get the same result by first obtaining the mapping of edge parent in M, which is edge parent in MM, and then taking the source vertex of this latter edge (i.e., $src_{MM}(f_E(\text{parent})) = \text{Class}$).

Given a meta-model MM, we define the set $SEM(MM) = \{M \mid \exists f \colon M \longrightarrow MM\}$ of all models typed by MM. We also say that $M \in SEM(MM)$ is a typed graph.

3.2 Graph Transformation

Changing the language variant in use entails the migration of the current model to the new variant. We use graph transformation [28] for this task. This is a rule-based declarative transformation approach with a formal basis. Next, we introduce the basic concepts that we will use in our proposal, and refer to [28] for more details.

The theory of graph transformation works with graphs and morphisms (like those in Definitions 3.1 and 3.4) and has been generalised to work with more abstract structures [28]. Conceptually, rules have a *left-hand side* graph[3] (LHS) describing a pattern to be found on a model, a *right-hand side* graph (RHS) defining the changes to perform to the model, and an intermediate *gluing graph* K with the common parts of the LHS and the RHS. In addition, rules can define a set of *negative application conditions* (NACs) stating forbidden conditions on the model for the rule to be applicable.

Definition 3.6 (Graph transformation rule). A *graph transformation rule* $tr = \langle L \xleftarrow{l} K \xrightarrow{r} R, NACS = \{L \xrightarrow{n_i} N_i\}_{i \in I} \rangle$ is made of:

---

[3] In the paper, we use the terms graph and model interchangeably.

- Three (typed) graphs L (called the left-hand side, LHS), K (called the gluing graph), and R (called the right-hand side, RHS), with two injective morphisms l and r between them
- A set $NACS$ of negative application conditions made of a collection of graphs N_i (for $i \in I$) and injective morphisms n_i from L to each such graph

Example 3.7. The top of Figure 4 shows an example rule *tr* that creates a parent class named *Parent* for two classes that lack a parent class. The morphisms l and r are defined by equality of identifiers (e.g., morphism l maps node c1 in graph K to c1 in graph L). The rule has two NACs, given by morphisms $n_0 \colon L \longrightarrow N_0$ and $n_1 \colon L \longrightarrow N_1$, where N_0 and N_1 are isomorphic. The figure shows morphisms n_0 and n_1 explicitly as mappings, since they map differently c1 and c2. The NACs forbid applying the rule if either Class identified by L has a parent.

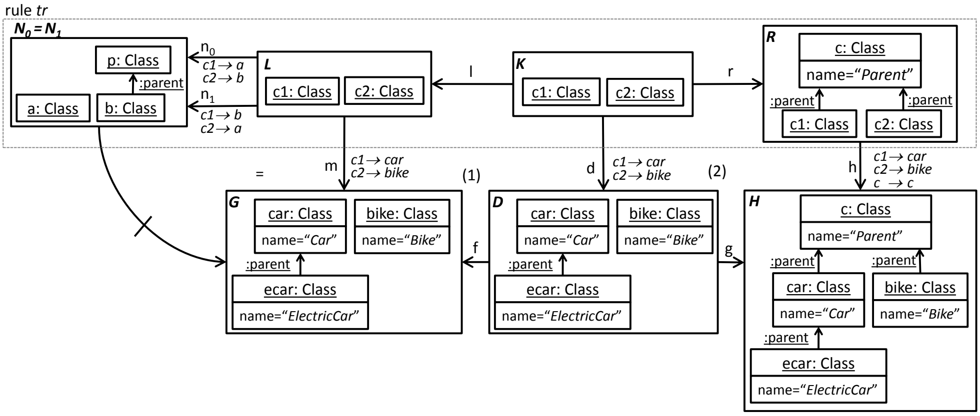


Fig. 4. Example rule (top) and rule application to a graph G yielding graph H.

A rule is applicable on a model if the model contains an occurrence (i.e., a *match*) of the LHS, no occurrence of the NACs (i.e., the model does not include any of the graphs N_i), and the rule application yields a valid model. The rule application deletes the elements present in the LHS but not on the RHS ($L \setminus l(K)$)[4], and adds those present in the RHS but not in the LHS ($R \setminus r(K)$). The resulting graph is valid if the match satisfies the *dangling edge* and the *identification* conditions. The former states that if a node is deleted, all its incident and outgoing edges should be deleted as well to avoid dangling edges without source or target. The identification condition states that if two elements in the LHS are identified into a single element in the model (via a non-injective match), then the rule does not specify contradictory actions for them (i.e., deleting one and preserving the other) [28].

Definition 3.8 (Rule application [28]). Given a rule $tr = \langle L \xleftarrow{l} K \xrightarrow{r} R, NACS = \{L \xrightarrow{n_i} N_i\}_{i \in I} \rangle$ and a graph G, tr is *applicable* on G via the *match* morphism $m \colon L \to G$, written $G \models_m tr$, if:

- There is no injective morphism $m_i \colon N_i \longrightarrow G$ from any negative application condition in $NACS$, s.t. the triangle to the left of Figure 5 commutes (i.e., $\nexists m_i \cdot m_i \circ n_i = m$)
- *Dangling edge condition*: the nodes in L whose image under m are the source or target of an edge in G that is not mapped by m, are preserved by tr (cf. Definition 3.9 in [28])
- *Identification condition*: if two nodes or edges in L have the same image under m, they are preserved by tr (cf. Definition 3.9 in [28])

---

[4] $L \setminus l(K)$ are the elements (vertices and edges) belonging to L that $l(K)$ does not map.

$$\begin{tikzcd} N_i & L \arrow[l, "n_i"'] \arrow[d, "m"] \arrow[dl, phantom, "="] \\ & G \arrow[from=1-1, "m_i"'] \end{tikzcd} \qquad \begin{tikzcd} L \arrow[d, "m"] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "d"] \arrow[dr, phantom, "(2)"] & R \arrow[d, "h"] \\ G & D \arrow[l, "g"] \arrow[r, "f"'] & H \end{tikzcd}$$

Fig. 5. Satisfaction of NACs (left). Rule application (right).

Given a rule tr, a graph G, and a match m s.t. $G \models_m tr$, then tr is applied to G yielding graph H, written $G \overset{tr,m}{\Longrightarrow} H$, by the double pushout diagram to the right of Figure 5, where (1) and (2) are pushouts. We write $G \overset{tr^*}{\Longrightarrow} H$ for zero or more consecutive applications of tr, yielding graph H.

A pushout [28] is a gluing construction that merges two graphs (e.g., D, R) via a common subgraph (e.g., $D \overset{d}{\leftarrow} K \overset{r}{\rightarrow} R$). A rule application (cf. right of Figure 5) calculates first a pushout complement graph D, which is a graph that makes the square (1) a pushout. Intuitively, it is a graph equal to G, but deprived of the elements that are in L and not in K ($m(L \setminus l(K))$). A second pushout (square (2)) adds to D the elements in $R \setminus r(K)$, yielding graph H.

Example 3.9. Figure 4 shows an example rule application. The rule is applied to graph G, on a match identifying c1 to car and c2 to bike. This is allowed since neither car nor bike have a parent[5]. Instead, identifying c1 or c2 to ecar is not possible because ecar has a parent, which violates the NACs. The rule does not delete anything (D is isomorphic to G), but it creates a Class named *Parent* connected to car and bike. The created elements are those belonging to $R \setminus r(K)$ (i.e., the node c and the two edges). The pushout of square (2) performs this creation, merging graphs R and D via the common elements in K, to yield graph H. In the rest of the paper, rules will omit graph K and morphisms l, r and n_i, as they can be deduced by the equality of object identifiers in L, R and N_i.

Given a set RS of transformation rules and a graph G, we use the predicate $terminal(G, RS) \triangleq \forall tr_i \in RS, \nexists m \colon L_i \rightarrow G \cdot G \models_m tr_i$ to denote that no rule in RS is applicable to G. We write $G \overset{RS^*}{\Longrightarrow} H$ for zero or more consecutive applications of the rules within RS starting from graph G.

Our notion of transformation system requires the concept of trace of a derivation, defined next.

Definition 3.10 (Derivation trace). Given a set RS of rules, a graph G, and a derivation $d : G \overset{tr_i}{\Longrightarrow} G_1 \ldots \overset{tr_n}{\Longrightarrow} G_n$, the function $trace(d) = tr_i \ldots tr_n$ yields the sequence of rules applied within d.

A graph transformation system is made of rules where L, K, R and N_i are typed by a common meta-model MM. We consider transformation units [43] to control the rule execution order. These consist of regular expressions over rules, which can include parenthesis for grouping, and use tr^* to denote 0 or more applications of the rule tr, tr^+ for 1 or more applications of tr, $tr_0 + tr_1$ for the application of tr_0 or tr_1, and $tr_0; tr_1$ for the sequential application of tr_0 and tr_1. Given a regular expression C, we write $LAN(C)$ to denote the language it defines.

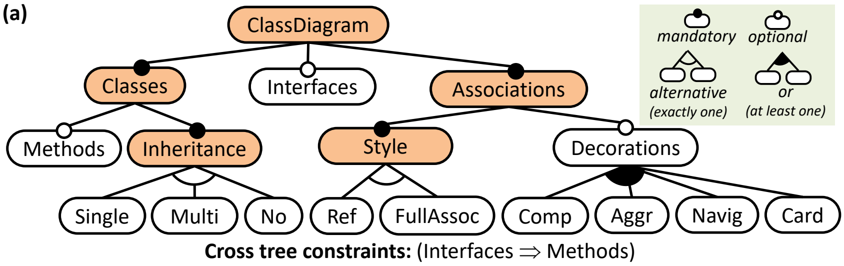
Definition 3.11 (Graph transformation system). A *graph transformation system* $GTS = \langle RS, MM, C \rangle$ contains a set RS of rules typed over meta-model MM, and a regular expression C over the rules in RS.

Finally, we define the semantics of a graph transformation system, which is given by all terminal graphs produced by derivations whose trace belongs to the language of the regular expression.

---

[5] Another valid injective match from L to G exists, identifying c1 to bike and c2 to car, as well as two other non-injective matches, identifying c1 and c2 to car, and c1 and c2 to bike.

(a)



(b) $FM = (F = \{\text{ClassDiagram, Classes, Methods, Inheritance, Single, ...}\},$

$\psi = \text{ClassDiagram} \wedge \text{Classes} \wedge \text{Associations} \wedge \text{Inheritance} \wedge \text{Style} \wedge$
$((\text{Decorations} \Leftrightarrow (\text{Comp} \vee \text{Aggr} \vee \text{Navig} \vee \text{Card})) \wedge$
$((\text{Single} \wedge \neg \text{Multi} \wedge \neg \text{No}) \vee (\neg \text{Single} \wedge \text{Multi} \wedge \neg \text{No}) \vee (\neg \text{Single} \wedge \neg \text{Multi} \wedge \text{No})) \wedge$
$((\text{Ref} \wedge \neg \text{FullAssoc}) \vee (\neg \text{Ref} \wedge \text{FullAssoc})) \wedge$
$(\text{Interfaces} \Rightarrow \text{Methods})$

Fig. 6. Feature model for the class diagrams adaptive language represented using: (a) the feature diagram notation, and (b) Definition 3.13.

Definition 3.12 (Application of graph transformation system). Given a graph transformation system $GTS = \langle RS, MM, C \rangle$, and a graph G typed over MM, its semantics $SEM_G(GTS) = \{H \mid \exists d : G \overset{RS^*}{\Longrightarrow} H \wedge terminal(H, RS) \wedge trace(d) \in LAN(C)\}$ consists of all terminal graphs H produced by 0 or more applications of the rules in RS, such that the trace of the derivation belongs to the language of the regular expression C.

3.3 Language Product Lines

As a first step to define an adaptive language, we build on works [30, 55] that propose combining meta-models and software product lines to create families of modelling languages in a compact way. This way, each variant of an adaptive language corresponds to a language of the family.

We define the variability space of an adaptive language by means of a *feature model*, which represents all features an adaptive language may have and restricts how they can be combined. While feature diagrams [39] are a popular notation for them, we use a formalisation to facilitate the precise definition of adaptive language and related concepts in the next section.

Definition 3.13 (Feature model [30]). A *feature model* $FM = (F, \Psi)$ consists of a set of variables $F = \{f_1, ..., f_n\}$ called *features*, and a propositional formula Ψ over the variables in F.

Example 3.14. Figure 6 shows the variability in the adaptive class diagrams language of our running example, represented using the feature diagram notation in part (a), and Definition 3.13 in part (b). The feature model allows choosing whether classes have methods; the supported kind of inheritance (single, multiple or none); whether interfaces are supported; the style for associations (unidirectional references or full associations); and the available decorations for association ends (composition, aggregation, navigation, and cardinality). The cross-tree constraint ensures that when interfaces are present in a language variant, so are methods.

A specific selection of features that is compatible with the feature model is called a *configuration*.

Definition 3.15 (Feature configuration). Given a feature model $FM = (F, \Psi)$, a *configuration* $\rho \subseteq F$ is a *partition* of F into two subsets of selected ($F^+ = \rho$) and unselected ($F^- = F \setminus \rho$) features that satisfy Ψ, i.e., $\Psi[true/F^+, false/F^-]$ evaluates to true when each $f \in F^+$ is substituted by true, and each $f \in F^-$ by false. We write $CFG(FM)$ for the set of all configurations of FM.

Example 3.16. The feature model in Figure 6 admits 288 configurations. Three of them are ρ_A = {Multi, FullAssoc, Decorations, Card}, ρ_D = {Methods, Multi, FullAssoc, Decorations, Comp, Aggr, Navig, Card}, and ρ_J = {Methods, Single, Ref, Interfaces, Decorations, Comp, Aggr, Navig, Card}. We will use these configurations to obtain the analysis, design and Java variants of the class diagrams adaptive language in our running example (cf. Figure 1). For simplicity, the configurations only list the selected features with white background in Figure 6, since the shaded features (e.g., ClassDiagram, Classes) are mandatory and must be selected in any configuration.

A *language product line* (LPL) [30] comprises a feature model and a so-called 150% meta-model (*150MM*). The latter overlaps the meta-models of all language variants, and its elements attach a boolean formula – called presence condition (PC) – stating the variants the element belongs to.

Definition 3.17 (Language product line). A *language product line* is a tuple $LPL = \langle FM, MM, \Phi \rangle$ consisting of:

- A feature model $FM = (F, \Psi)$
- A meta-model MM, called the 150% meta-model (*150MM*)
- A tuple $\Phi = \langle \Phi_V, \Phi_E, \Phi_A \rangle$ of functions $\Phi_X \colon X \longrightarrow \mathbf{Prop}_F$ (for $X \in \{V, E, A\}$) assigning presence conditions (PCs) to the *150MM* elements. $\mathbf{Prop}_F$ is the set of all propositional formulae over the features in F, and $\Phi(x)$ is called the PC of x[6]

such that the following conditions hold:

- The PC of each attribute $a \in A$ must be stronger than that of its owning class: $\Phi(a) \Rightarrow \Phi(owner(a))$
- The PC of each reference $r \in E$ must be stronger than that of its source and target classes: $(\Phi(r) \Rightarrow \Phi(src(r))) \wedge (\Phi(r) \Rightarrow \Phi(tar(r)))$

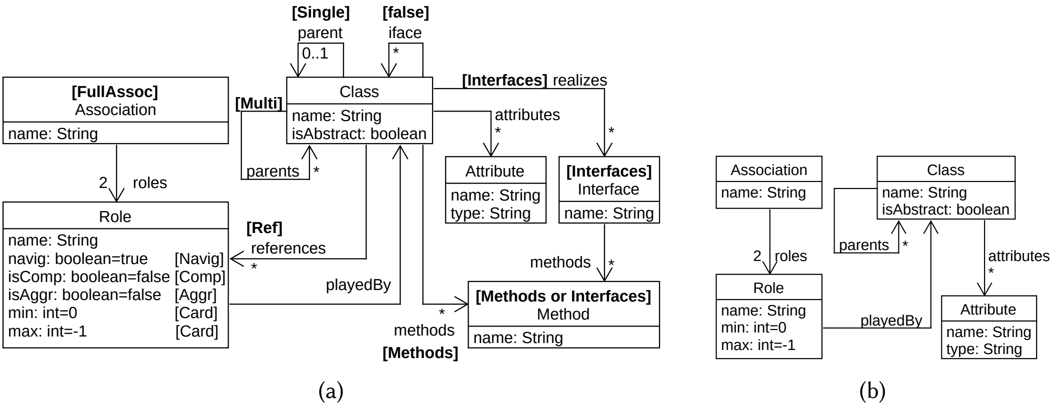
Example 3.18. Figure 7(a) shows the *150MM* for the example. It displays the PCs between square brackets, omitting those equal to true. For instance, the PC of class Interface is Interfaces. This PC is a propositional formula that uses features (Interfaces) as variables. Hence, selecting feature Interfaces makes this formula true, while not selecting it makes the formula false. In the figure, the PC of Role is true so the figure does not show it. By convention, the figures assume that the PC of *fields* (attributes and references) is conjoined with that of their owning class. For instance, the PC of Interface.methods is Interfaces, the PC of Role.navig is Navig, and the one of Method.name is Methods ∨ Interfaces (i.e., Method.name will be present in any language variant that selects either Methods or Interfaces). This simplifies the definition of the LPLs and ensures the required implication from the PC of fields to the PC of their owner classes (e.g., Φ(Interface.name) ⇒ Φ(Interface)). Elements with PC false (like reference Class.iface) are auxiliary elements used by the migration transformations (cf. Section 4) but absent from any language variant. This avoids polluting the individual meta-models of the language variants with these auxiliary elements. Finally, the figure shows cardinalities in references, but since the notion of meta-model of Definition 3.1 does not consider them, these are displayed for explanatory purposes only.

Given a configuration, we can *derive* a meta-model variant (i.e., a *product*) by removing from the *150MM* the elements whose PC evaluates to false when substituting the features in their PC by their value in the configuration.

Definition 3.19 (Derivation). Given $LPL = \langle FM, MM = \langle V, D, E, A, src, tar, owner, val \rangle, \Phi \rangle$ and a configuration $\rho \in CFG(FM)$, a meta-model product $MM_\rho = \langle V_\rho, D, E_\rho, A_\rho, src_\rho, tar_\rho, owner_\rho, val_\rho \rangle$ is *derived* by deleting from the *150MM* those elements whose PC evaluates to false in configuration

---

[6] For simplicity, given $x \in V \cup E \cup A$, we use $\Phi(x)$ (instead of, e.g., $\Phi_V(x)$) when no confusion can arise.



(a) (b)

Fig. 7. (a) 150% meta-model for the class diagrams adaptive language. (b) Meta-model product MM_{ρ_A}.

ρ, i.e., $X_\rho = \{x \in X \mid \Phi(x)[true/F^+, false/F^-] = true\}$, for $X \in \{V, E, A\}$, and restricting the functions: $src_\rho = src|_{E_\rho}$, $tar_\rho = tar|_{E_\rho}$, $owner_\rho = owner|_{A_\rho}$, $val_\rho = val|_{A_\rho}$.

Example 3.20. Figure 7(b) shows the meta-model derived from the *150MM* of Figure 7(a) using the configuration $\rho_A = \{$Multi, FullAssoc, Decorations, Card$\}$ (i.e., the analysis class diagrams meta-model). According to Definition 3.19, the derivation deletes all classes, attributes and references whose PC evaluates to false for the given configuration. The derivation does not delete elements of D, i.e., data types like `String` or `int`. The unused data types are simply ignored.

4 ADAPTIVE MODELLING LANGUAGES

This section builds on LPLs to introduce the new notion of *adaptive modelling language*. This extends LPLs with support for model migration between the language variants of a family.

A major concern in this proposal is to avoid the explicit specification of migration transformations between every two variants derivable from the LPL, since the cost may be prohibitive (e.g., the running example would imply defining 288·287 = 82 656 transformations). To this aim, we provide means to define smaller transformation pieces (called *language adapters*) that take care of the migration tasks needed upon changing individual language features (or a small set of them). A language adapter declares a set of feature *diff*erences (features changes and feature invariants), plus a set of in-place transformation rules stating how models should be changed to accommodate those diffs. This way, an adapter is directed to bridge the gap between a (typically reduced) set of language features. When moving from a source to a target language configuration, their feature diffs are identified, and a suitable migration transformation is constructed on the fly by combining adapters compatible with such diffs. As we will see later, a transformation from MM_{ρ_s} to MM_{ρ_t} will include all adapters having a diff consistent with the configuration diff between ρ_t and ρ_s.

Next, Section 4.1 describes configuration diffs as a way to express changes in configurations. Then, Section 4.2 uses them to build language adapters that permit modularising model migration transformations feature-wise. Section 4.3 defines adaptive languages as LPLs equipped with language adapters that ensure the interoperability between language variants. Finally, Section 4.4 extends adaptive languages with adaptation triggers. For readability, part of the theory and the proofs of the lemmas and propositions can be found in Appendix A.

4.1 Diffs and Configuration Diffs

We start defining diffs, which represent changes and invariants in the selection values of a set of features. A diff is a tuple made of a difference δ (the features that modify their selection value) and a context C (the features that preserve their selection value).

Definition 4.1 (Diff). Given a feature model $FM = \langle F, \Psi \rangle$, a *diff* $\Delta = \langle \delta, C \rangle$ contains:

- A tuple $\delta = \langle F^{+-}, F^{-+} \rangle$ called *difference*, with sets $F^{+-} \subseteq F$ of features changing from selected to unselected, and $F^{-+} \subseteq F$ of features changing from unselected to selected
- A tuple $C = \langle F^{++}, F^{--} \rangle$ called *context*, with sets $F^{++} \subseteq F$ of features remaining selected, and $F^{--} \subseteq F$ of features remaining unselected

such that the four sets F^{+-}, F^{-+}, F^{++}, F^{--} are disjoint.

The union of the feature sets within a diff is not required to yield the complete set of features F, but diffs may describe just a few changes in a configuration, like (un)selecting one feature. These are called *partial* diffs, and we use them to specify the conditions for including an adapter in a migration transformation. In contrast, *configuration* diffs consider all features within a feature model, and we use them to describe the difference between two configurations.

Example 4.2. The diff $\Delta_1 = \langle \delta = \langle \{\text{Multi}\}, \{\text{Single}\} \rangle, C = \langle \{\text{Methods}\}, \{\} \rangle \rangle$ states that Multi changes to unselected, Single to selected, and Methods remains selected. The features not included in the diff can change or retain their value. As shown in Figure 8(a), Δ_1 is a partial diff as it uses a subset of the features of the feature model, describing some feature changes and contextual conditions that remain invariant. We will attach this type of diff to adapters. Instead, Δ_{DJ} in Figure 8(b) is a configuration diff[7] that captures how all features change or retain their value when moving from ρ_D to ρ_J. We will define compatibility conditions between diffs that will enable selecting adapters with diffs like Δ_1 when assembling a migration transformation from ρ_D to ρ_J.

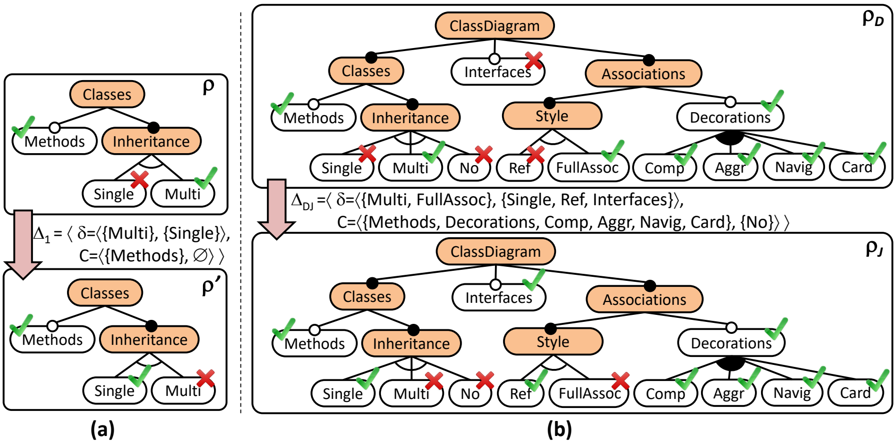


Fig. 8. (a) A partial diff Δ_1 expressing a few changes and contextual conditions. (b) A configuration diff Δ_{DJ} expressing the difference between ρ_D and ρ_J. As Definition 4.7 will show, Δ_1 is compatible with Δ_{DJ}.

Not any diff is meaningful, but the features included in their difference and context need to be compatible with the feature model. As the next definition states, in a well-formed (*wff*) diff,

---

[7] For readability, configuration diffs like Δ_{DJ} omit all mandatory features that are always selected, like ClassDiagram.

the initially selected ($F^{+-} \cup F^{++}$) and unselected ($F^{-+} \cup F^{--}$) features, and the finally selected ($F^{-+} \cup F^{++}$) and unselected ($F^{+-} \cup F^{--}$) features, need to be compatible with the feature model.

Definition 4.3 (Well-formed diff). A diff Δ is *well-formed* (*wff*) w.r.t. $FM = \langle F, \Psi \rangle$ if:

(1) the pre-state (i.e., the initial feature values) is wff:
$\Psi[true/(F^{+-} \cup F^{++}), false/(F^{-+} \cup F^{--})] \neq false$
(2) the post-state (i.e., the final feature values) is wff:
$\Psi[true/(F^{-+} \cup F^{++}), false/(F^{+-} \cup F^{--})] \neq false$

Condition (1) in Definition 4.3 requires that, when taking the features that the diff assumes true (F^{+-}, F^{++}) and false (F^{-+}, F^{--}), there is no contradiction with the feature model. In our example, a diff assuming both Multi and Single to be true would not be wff. Condition (2) states that the features that become (or stay) true (F^{-+}, F^{++}) and false (F^{+-}, F^{--}) after the diff application should not be contradictory with the feature model. For example, a diff selecting Single (F^{-+}) and assuming that Multi stays selected (F^{++}) would not be wff.

Example 4.4. The diff $\Delta_2 = \langle \delta = \langle \{\text{Multi, Single}\}, \{\} \rangle, C = \langle \{\}, \{\} \rangle \rangle$ is not wff for the feature model of the running example, since both Multi and Single cannot be true at the same time, so the pre-state is not wff. Conversely, the diff $\Delta_3 = \langle \delta = \langle \{\text{Multi}\}, \{\} \rangle, C = \langle \{\}, \{\} \rangle \rangle$ is wff. Even if it does not specify that either Single or No should become selected (since Multi is deselected), the changes in Δ_3 do not contradict the feature model.

Appendix A.1 shows that diffs can be used to transform configurations [28]. However, we are rather interested in their use to express the difference between two configurations (a *configuration diff*, cf. Definition 4.5), and then check if partial diffs are compatible with that difference (using notions of *diff inclusion* and *consistency*, cf. Definition 4.7). Next, Definition 4.5 uses diffs to record all feature values that are modified and preserved when moving from one configuration to another.

Definition 4.5 (Configuration diff). Given $\rho_i, \rho_j \in CFG(FM)$, the *configuration diff* $\rho_j - \rho_i$ (which records the feature changes and invariants when moving from ρ_i to ρ_j) is given by the diff $\Delta_{ij} = \langle \delta_{ij} = \langle F_i^+ \cap F_j^-, F_i^- \cap F_j^+ \rangle, C_{ij} = \langle F_i^+ \cap F_j^+, F_i^- \cap F_j^- \rangle \rangle$.

Example 4.6. Given the configurations ρ_D and ρ_J in Example 3.16, the configuration diff $\rho_J - \rho_D$, which corresponds to moving from configuration ρ_D to configuration ρ_J, is $\Delta_{DJ} = \langle \langle \{\text{Multi, FullAssoc}\}, \{\text{Single, Ref, Interfaces}\} \rangle, \langle \{\text{Methods, Decorations, Comp, Aggr, Navig, Card}\}, \{\text{No}\} \rangle \rangle$ (cf. Figure 8(b)). Hence, features Multi and FullAssoc change to unselected; Single, Ref and Interfaces change to selected; and the others preserve their selection value.

Next, we define diff inclusion and consistency, which enable checking if a partial diff is compatible with another, "bigger" diff (like a configuration diff). Later, in Section 4.2, we will define language adapters with diffs, and exploit the notion of diff consistency to compose full migration transformations out of adapters.

Definition 4.7 (Diff inclusion and consistency). Given two diffs $\Delta = \langle \langle F^{+-}, F^{-+} \rangle, \langle F^{++}, F^{--} \rangle \rangle$ and $\Delta' = \langle \langle F'^{+-}, F'^{-+} \rangle, \langle F'^{++}, F'^{--} \rangle \rangle$, we say that:

- Δ is *included* in Δ' (written $\Delta \subseteq \Delta'$) if $F^X \subseteq F'^X$, for $X = \{+-, -+, ++, --\}$
- Δ is *pre-consistent* with Δ' (written $\Delta \sqsubseteq_{pre} \Delta'$) if $F^{+-} \subseteq F'^{+-}$, $F^{-+} \subseteq F'^{-+}$, $F^{++} \subseteq (F'^{++} \cup F'^{+-})$, and $F^{--} \subseteq (F'^{--} \cup F'^{-+})$
- Δ is *post-consistent* with Δ' (written $\Delta \sqsubseteq_{post} \Delta'$) if $F^{+-} \subseteq F'^{+-}$, $F^{-+} \subseteq F'^{-+}$, $F^{++} \subseteq (F'^{++} \cup F'^{-+})$, and $F^{--} \subseteq (F'^{--} \cup F'^{+-})$

Diff inclusion requires the feature sets in Δ to be included in those of Δ'. Diff consistency is more permissive as the context may be satisfied in the pre- or post-states. That is, the delta features of Δ must be included in those of Δ', but the context of Δ can either be guaranteed by the context of Δ' or be satisfied at the initial (for pre-consistency) or final (for post-consistency) configurations. If $\Delta \subseteq \Delta'$, then $\Delta \sqsubseteq_{pre} \Delta'$ and $\Delta \sqsubseteq_{post} \Delta'$.

Example 4.8. In our example, $\Delta_1 = \langle\langle\{\textsf{Multi}\}, \{\textsf{Single}\}\rangle, \langle\{\textsf{Methods}\}, \{\}\rangle\rangle$, and $\Delta_{DJ} = \langle\langle\{\textsf{Multi}, \textsf{FullAssoc}\}, \{\textsf{Single}, \textsf{Ref}, \textsf{Interfaces}\}\rangle, \langle\{\textsf{Methods}, \textsf{Decorations}, \textsf{Comp}, \textsf{Aggr}, \textsf{Navig}, \textsf{Card}\}, \{\textsf{No}\}\rangle\rangle$ (cf. Figure 8). Then, $\Delta_1 \subseteq \Delta_{DJ}$, since every set in Δ_1 is included in the corresponding set of Δ_{DJ}. On the contrary, $\Delta = \langle\langle\{\textsf{Multi}\}, \{\textsf{Single}\}\rangle, \langle\{\textsf{Ref}\}, \{\}\rangle\rangle \not\subseteq \Delta_{DJ}$, since Ref is not in the positive context of Δ_{DJ}. However, $\Delta \sqsubseteq_{post} \Delta_{DJ}$, since $\textsf{Ref} \in F_{DJ}^{-+}$.

4.2 Language Adapters

A language adapter associates a graph transformation system to a diff. Intuitively, the transformation encodes how to adapt a model when the language variant changes according to the diff. Adapters typically manage changes in a single language feature, or a reduced set of them. This way, they enable defining migration transformations feature-wise.

Definition 4.9 (Language adapter). Given a language product line $LPL = \langle FM, MM, \Phi\rangle$, a *language adapter* $a = \langle \Delta, GTS\rangle$ is made of a diff Δ over FM, and a graph transformation system $GTS = \langle RS, MM, C\rangle$.

Remark 4.10. The rules in RS are typed over the *150MM* of the LPL, so they can use any element of the language, including the auxiliary ones.

Example 4.11. Figure 9 shows three language adapters for the running example. Their rules are typed over the *150MM* in Figure 7(a). Adapter InhByDelegation transforms from multiple to single inheritance when the feature Ref remains selected, as specified by the adapter diff Δ. The adapter has two rules: multiBySingle and inhByRef. The adapter's regular expression C specifies that these rules are to be applied randomly as long as possible. The first rule changes a link parents (used for multiple inheritance) by a link parent, provided that the child class has no other parents (checked by the NACs). Instead, if the child class already has a parent, then the second rule substitutes the link parents by a reference. This rule also creates an auxiliary link iface, which other adapters may process (in particular, adapter InhByDelegationInterface). As Definition 4.17 will show, after applying all suitable adapters to a model, a subsequent step removes from the model all elements that do not belong to the target language meta-model (e.g., link iface, or attributes tar.min and tar.max if the target configuration does not select feature Card). This way, by setting values that can be removed if not needed, a single rule can address several similar cases.

Adapter InhByDelegationInterface is to be used when moving from Multi to Single inheritance, and features Interfaces and Methods remain selected (positive context of the diff). It comprises two rules to be applied randomly as long as possible. The first one creates an Interface for each class pointed by an iface link, if the interface does not exist yet (ensured by the NAC). The second rule creates suitable Method objects in the interface and the source class of the iface link. The rules do not need to delete the iface links, but this is deferred to the final deletion step mentioned above.

Finally, adapter AssocByRef transforms full associations into references. It declares two rules to be randomly applied as long as possible: addNavigRole, which creates a reference for each navigable association role, and removeNonNavigRole, which deletes non-navigable roles. The rules do not delete the Association objects, but the final deletion step will take care of that. Note that the adapter's Δ does not include feature Navig in its positive context, even though the rules make use of attribute

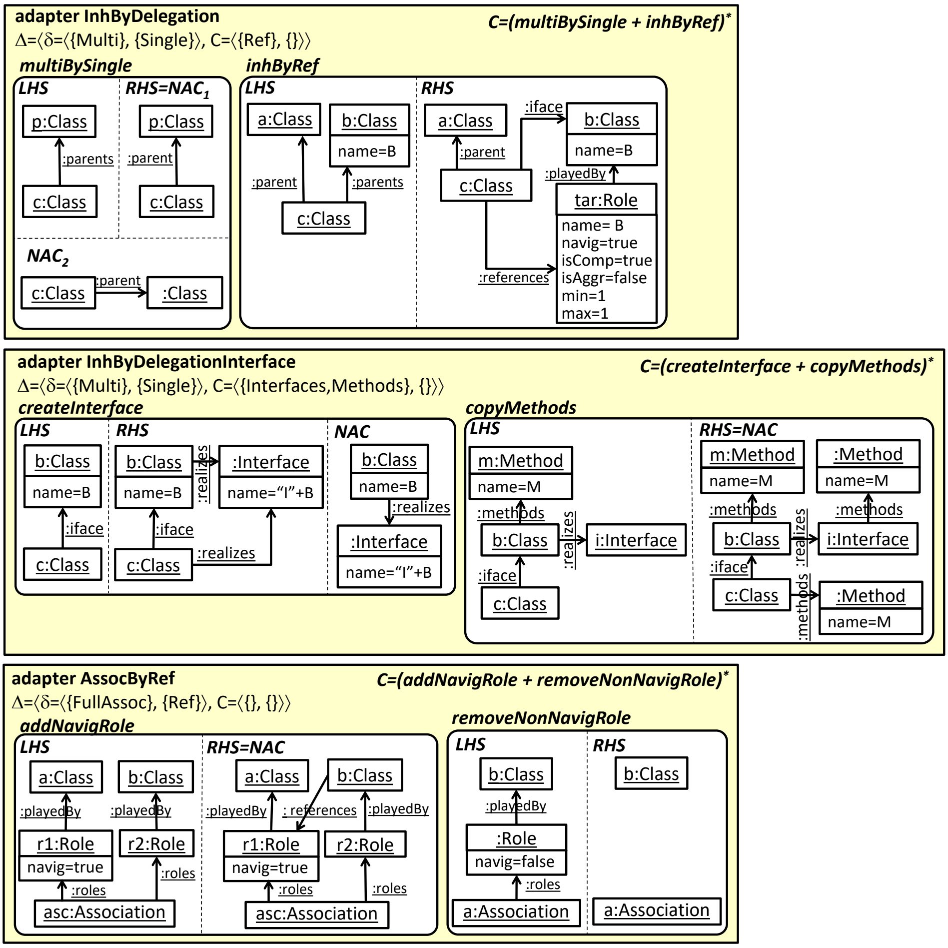


Fig. 9. Three language adapters for the running example (cf. feature model and *150MM* in Figures 6 and 7(a)).

navig. This is allowed since rules are typed by the *150MM*. Moreover, as Definition 4.17 will show, any model adaptation will start by making it conform to the *150MM*, adding any missing fields with their default value (e.g., adding navig with value false to the Role objects if they lack this attribute). This avoids having two sets of rules, for the cases that the Navig feature is or is not selected.

Next, we define a set of predicates ($create$, $delete$, $preserve$, $forbid$, $read$) characterising the actions that a rule performs on the objects of types activated by a set of selected (FS^+) and unselected (FS^-) features. For example, a rule tr satisfies predicate $create(FS^+, FS^-, tr)$ if the rule creates an object o whose type $type(o)$ has a PC that: (1) uses some of the features in FS^+ or FS^-, and (2) is satisfied when substituting the features in FS^+ by true, and those in FS^- by false. Similarly, tr satisfies $forbid$, if any of its NACs contains an object of a type activated by the predicate features. Definition 4.17 and Algorithm 2 employ these predicates to choose the adapters used to build migration transformations, e.g., to avoid selecting those that create elements whose type is not present in the target configuration, and those that delete elements whose type is not present in the source configuration.

Definition 4.12 (Rule-feature interaction). Given two disjoint feature sets FS^+ and FS^-, and a rule $tr = \langle L \xleftarrow{l} K \xrightarrow{r} R, NACS = \{L \xrightarrow{n_i} N_i\}_{i \in I}\rangle$, we define the following predicates[8]:

$$\begin{aligned}
create(FS^+, FS^-, tr) &\triangleq \exists x \in (R \setminus r(K)) \cdot ActiveType(type(x), FS^+, FS^-)\\
delete(FS^+, FS^-, tr) &\triangleq \exists x \in (L \setminus l(K)) \cdot ActiveType(type(x), FS^+, FS^-)\\
preserve(FS^+, FS^-, tr) &\triangleq \exists x \in K \cdot ActiveType(type(x), FS^+, FS^-)\\
forbid(FS^+, FS^-, tr) &\triangleq \exists n_i \colon L \longrightarrow N_i, \exists x \in (N_i \setminus n_i(L)) \cdot ActiveType(type(x), FS^+, FS^-)\\
read(FS^+, FS^-, tr) &\triangleq delete(FS^+, FS^-, tr) \vee preserve(FS^+, FS^-, tr) \vee forbid(FS^+, FS^-, tr)
\end{aligned}$$

with

$$ActiveType(t, FS^+, FS^-) \triangleq TermOf(FS^+ \cup FS^-, \Phi(t)) \wedge \Phi(t)[true/FS^+, false/FS^-] = true$$

where $TermOf(F, \Phi)$ holds if the formula Φ uses some of the literals in the set F.

In the definition, predicate $ActiveType(t, FS^+, FS^-)$ holds if the PC of type t is true and uses some feature in the sets FS^+ or FS^-. Next, we generalise some of these predicates for adapters.

Definition 4.13 (Adapter-feature interaction). Given two disjoint sets FS^+ and FS^- of features, and an adapter $a = \langle \Delta, GTS \rangle$, we define the following predicates:

$$\begin{aligned}
create(FS^+, FS^-, a) &\triangleq \exists tr \in RS \cdot create(FS^+, FS^-, tr)\\
delete(FS^+, FS^-, a) &\triangleq \exists tr \in RS \cdot delete(FS^+, FS^-, tr)\\
read(FS^+, FS^-, a) &\triangleq \exists tr \in RS \cdot read(FS^+, FS^-, tr)
\end{aligned}$$

Example 4.14. Rule inhByRef in Figure 9 creates a Role object and links of type iface, playedBy and references. Hence, predicate $create(\{\text{Ref}\}, \{\}, \text{inhByRef})$ is true, since the PC of references is Ref, and this PC evaluates to true. On the contrary, predicate $delete(\{\text{Ref}\}, \{\}, \text{inhByRef})$ is false as the rule does not delete elements of types whose PC includes feature Ref. At the adapter level, predicate $create(\{\text{Ref}\}, \{\}, \text{InhByDelegation})$ is true, but $delete$ evaluated with the same parameters is false.

4.3 Adaptive Languages

An *adaptive modelling language* is defined as a language product line plus a set of language adapters.

Definition 4.15 (Adaptive modelling language). An *adaptive modelling language* $AL = \langle LPL, A \rangle$ is made of a language product line LPL and a set A of language adapters over LPL.

Example 4.16. Our example adaptive language comprises the LPL made of the feature model in Figure 6 and the *150MM* in Figure 7(a), and the seven language adapters in Figures 9 and 10. In Figure 10, adapter SingleToMulti replaces single by multiple inheritance, and so, its only rule swaps link parent by parents. Adapter SingleToNo replaces single by no inheritance, and its diff Δ requires Ref in its positive context (F^{++}). It has just one rule that swaps link parent by a reference. Adapter RefByAssoc replaces references by full associations. It has two rules that create Association objects, one handling the case of classes connected via opposite references, and the other handling unidirectional references. Finally, adapter InterfacesToNo deals with the case of deselecting the Interfaces feature, and assumes both Multi and Methods. It has two rules, one creating an abstract class for each interface, and the other copying the interface methods to the created class.

Next, Definition 4.17 describes the process for migrating a model from a source to a target language variant. First, the model – typed by the source language variant – is retyped to the *150MM*.

---

[8] In the following, given a graph G, we use $x \in G$ as a shortcut for $x \in (G_V \cup G_E \cup G_A)$.

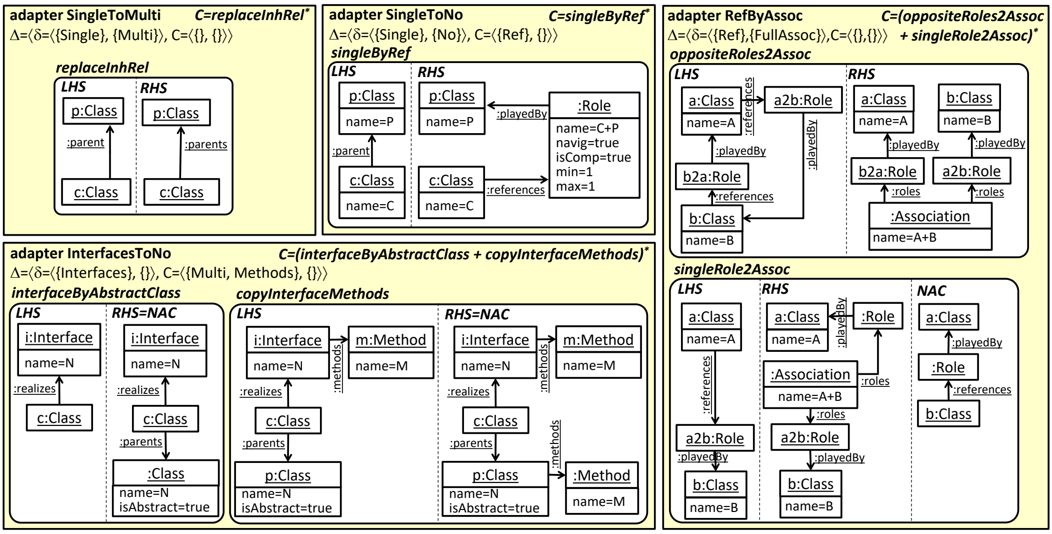


Fig. 10. Remaining adapters for the running example (cf. feature model and *150MM* in Figures 6 and 7(a)).

With our notion of meta-model and typing (cf. Section 3.1), a valid model of any language variant is also a valid model of the *150MM*. However, to fit in with the usual notion of conformance of objects to their types – which requires objects to have as many attributes as specified in their type – objects are added the non-instantiated attributes from their types, using their default values. If no default value is specified, they take the default value of their datatype (0 for numbers, false for Boolean, or the empty String). Then, in a second step, a graph transformation system is automatically assembled out of the adapters consistent with the language reconfiguration, taking their rules and the star-iterated sum of their regular expressions (i.e., the adapters are applied in random order, until none is applicable anymore). This transformation is applied to the model. Finally, in a third step, the elements not typed by the meta-model of the target language variant are removed from the migrated model.

Definition 4.17 (Migration between language variants). Given $AL = \langle\langle FM, MM, \Phi\rangle, A\rangle$ and two different configurations $\rho_s, \rho_t \in CFG(FM)$, the *migration* of a model M_s conforming to MM_{ρ_s} into a model M_t conforming to MM_{ρ_t} proceeds in three steps (cf. Figure 11):

(1) Model *augmentation*: M_s is retyped w.r.t. MM. Every object $o \in M_{s_V}$ is completed with new attributes typed by the attributes in $type(o)$ (if not already defined), using their default values. This yields model M'_s.

(2) Model *transformation*: The set of adapters consistent with Δ_{st} is collected (cf. Definition 4.7):

$$\begin{aligned} AD = \{a_k \in A \mid\ & \Delta_k \subseteq \Delta_{st} \vee \\ & (\Delta_k \sqsubseteq_{pre} \Delta_{st} \wedge \neg create(F_k^{++} \setminus F_{st}^{++}, F_k^{--} \setminus F_{st}^{--}, a_k)) \vee \\ & (\Delta_k \sqsubseteq_{post} \Delta_{st} \wedge \neg delete(F_k^{++} \setminus F_{st}^{++}, F_k^{--} \setminus F_{st}^{--}, a_k))\} \end{aligned}$$

This set is used to build the graph transformation system:

$$GTS_{st} = \langle \bigcup_{a_k \in AD} RS_k, MM, (\sum_{a_k \in AD} C_k)^* \rangle$$

GTS_{st} is applied on model M'_s, which yields model $M'_t \in SEM_{M'_s}(GTS_{st})$.

(3) Model *restriction*: M'_t is deleted the elements typed by $MM \setminus MM_{\rho_t}$, yielding model M_t.

$$\begin{tikzcd}
 & & MM & & \\
MM_{\rho_s} \arrow[urr, hook] & & (2) & & MM_{\rho_t} \arrow[ull, hook'] \\
M_s \arrow[u] \arrow[uurr, dashed, "(1)"] \arrow[r, hook] & M'_s \arrow[uur, dashed] \arrow[rr, Rightarrow, "GTS_{st}"] & & M'_t \arrow[uul, dashed] & M_t \arrow[u] \arrow[uull, dashed, "(3)"'] \arrow[l, hook']
\end{tikzcd}$$

Fig. 11. Model migration scheme from MM_{ρ_s} to MM_{ρ_t}.

Step 2 in Definition 4.17 collects all adapters whose diff is included in Δ_{st}, all pre-consistent adapters that do not create elements activated by the adapters' context but not by Δ_{st}'s context, and all post-consistent adapters that do not delete elements activated by the adapters' context but not by Δ_{st}'s context. This precludes selecting pre-consistent adapters creating elements of non-existent types in the target language variant (they would be removed in the third step of the migration), as well as post-consistent adapters deleting elements of non-existent types in the source variant.

Example 4.18. Figure 12 shows the migration of a model M_s from configuration ρ_A to configuration ρ_J (defined in Example 3.16). The first step (augmentation) retypes M_s w.r.t. MM (i.e., w.r.t. the *150MM* of the LPL). This produces a model M'_s, in which the two Role objects are added attributes navig, isComp and isAggr, to make them conform to class Role in MM (labels 1 and 2 in the figure).

The second step (transformation) creates a transformation system containing the rules of the adapters consistent with the language reconfiguration. The consistent adapters are InhByDelegation, AssocByRef and InhByDelegationInterface (cf. Figure 9). The first one removes multiple inheritance, the second converts full associations into references, and the third uses the auxiliary iface links created by InhByDelegation to add interfaces to the classes from which multiple inheritance is removed. Adapter AssocByRef is selected because its diff is included in $\Delta_{AJ} = \langle\langle\{\text{Multi, FullAssoc}\}, \{\text{Single, Ref, Interfaces, Methods, Comp, Aggr, Navig}\}\rangle, \langle\{\text{Decorations, Card}\}, \{\text{No}\}\rangle\rangle$. Adapter InhByDelegation is selected because it is post-consistent with Δ_{AJ} (its context requires Ref, which is available in ρ_J but not in ρ_A), and does not delete elements with PC Ref. Similarly, InhByDelegationInterface is post-consistent with Δ_{AJ} (its context requires Interfaces and Methods, only available in ρ_J) and does not delete elements with PC Interfaces or Methods. The regular expression of the resulting transformation system is the iterated sum of the regular expressions of the three adapters, which is equivalent to randomly applying the rules of the adapters for as long as possible.

Figure 12 applies the transformation system over model M'_s to yield model M'_t, which is terminal (no rules can be applied to it). The figure shows this transformation in two steps. The first one depicts the execution of rules multiBySingle and inhByRef, both from adapter InhByDelegation. The rules replace the links parents by links parent and iface, and create a Role object (labels 3 and 4 in the figure). Next, the transformation executes rules addNavigRole (twice) and createInterface. The first rule adds roles r1 and r2 to object c1, and the second rule creates an interface. Since the rules are applied randomly, other rule execution orders than the one in the example are possible.

The last step (restriction) removes from model M'_t the elements whose type does not belong to MM_t (i.e., the iface link and the Association object). The result is model M_t, which is typed by MM_{ρ_J}.

4.4 Adaptation Triggers

Triggered adaptive languages extend adaptive languages with triggers that unleash a change in the language variant in use, and migrate the current model accordingly. Triggers may consider not

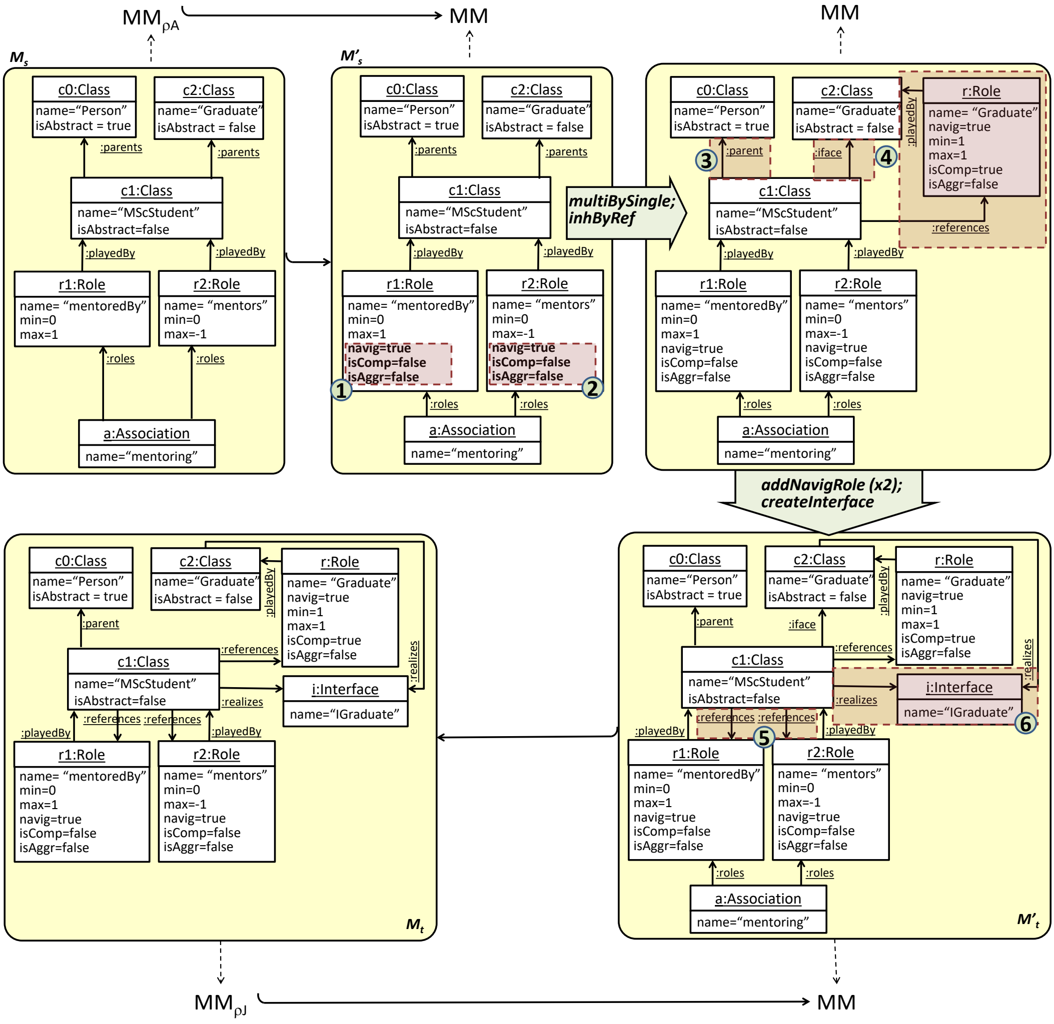


Fig. 12. Migrating a model from MM_{ρ_A} to MM_{ρ_J}.

only the model, but also contextual information such as user actions performed at modelling-time, or conditions about the modelling environment. For example, Figure 1 assumes the existence of a process model, and the user explicitly triggers the transition to the next phase by clicking on a button of the modelling IDE. Other scenarios may trigger language reconfigurations upon the occurrence of certain conditions in the model (e.g., expressed in OCL), the repetition of certain user errors, or the use of devices with different screen sizes, among many other possibilities.

Figure 13(a) depicts the working scheme of our approach, which involves three ingredients:

- A *triggered adaptive language*, which consists of an adaptive language, plus a state transition system whose states are configurations of the adaptive language. The triggered language may transition from one configuration to another when certain events (from a set Λ of relevant language events) occur.

- A *contextual adaptive model* that enriches models with a context and the current language configuration. The context captures relevant information for the modelling experience, and is represented as a sequence of timed events.
- *Adaptation triggers*, which are generated by a function called *eval*. The function receives a timed event from the context, and the current model and language configuration. Then, if appropriate, it generates a trigger that causes a reconfiguration of the adaptive language.

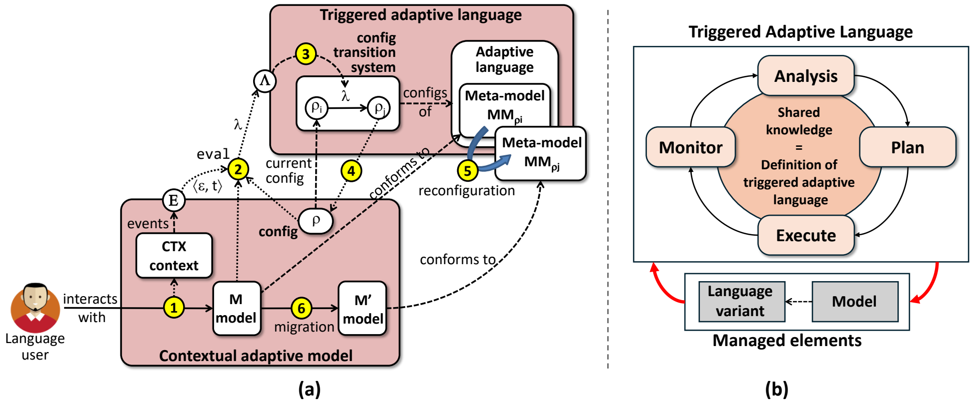


Fig. 13. (a) Working scheme of triggered adaptive modelling languages. (b) Adaptation MAPE-K loop.

As depicted in Figure 13(a), the user interacts with the model M (label 1). The context captures this interaction via a sequence of timed events, and may produce other events that consider further elements besides the model. When any of these events occurs, function *eval* (the adaptation trigger, label 2) evaluates whether the event is relevant for the current model state and language configuration. If so, the function forwards a new event λ to the triggered adaptive language (label 3). The language's configuration transition system determines whether, given the language configuration in use and the received event λ, a language reconfiguration should occur. In such a case, the new language configuration is stored in the contextual adaptive model (label 4), and the model M is migrated to become conformant with the new configuration (labels 5 and 6).

This way, similar to many self-adaptive [15] and autonomous software systems [42], triggered adaptive languages manage their adaptation using a MAPE-K (Monitor-Analyze-Plan-Execute over a shared Knowledge) loop, but tailored to languages as follows (cf. Figure 13(b)):

- *Monitor:* Triggered adaptive modelling languages monitor the context for relevant events. These events may include actions like saving or editing the model (to analyse constraints on it), explicit validation requests, or the explicit selection of language reconfigurations.
- *Analysis:* The function *eval* analyses the context event and the current model state and language configuration, and then forwards a reconfiguration event to the triggered adaptive language.
- *Plan:* The configuration transition system plans the target language variant to adapt to, based on the reconfiguration event produced by the *eval* function, and the current language configuration.
- *Execute:* The language is adapted to the new variant, and the model is migrated to this variant.
- *Knowledge:* This is the definition of the triggered adaptive language, comprising the *150MM*, the feature model, the adapters, and the configuration transition system.

We start defining a triggered adaptive modelling language as an adaptive language equipped with a configuration transition relation (a transition system over the set of all configurations, labelled over a set Λ of possible language events) and an initial configuration.

Definition 4.19 (Triggered adaptive modelling language). Given a set Λ of language events, a *triggered adaptive modelling language* over Λ is a tuple $TAL_\Lambda = \langle AL, CF, \rho_{init} \rangle$ made of:

- An adaptive modelling language $AL = \langle LPL = \langle FM, MM, \Phi \rangle, A \rangle$ as in Definition 4.15
- A configuration transition system $CF \subseteq CFG(FM) \times \Lambda \times CFG(FM)$, which is a deterministic labelled transition system having the language configurations as states and labels over Λ. Being deterministic, for every $\rho_s \in CFG(FM)$ and for every $\lambda \in \Lambda$, there is at most one $\rho_t \in CFG(FM)$ s.t. $(\rho_s, \lambda, \rho_t) \in CF$
- An initial configuration $\rho_{init} \in CFG(FM)$

Example 4.20. Without any restriction, the full variability space of the triggered adaptive language of our running example would yield a transition system with 288 language configurations as states, and 82 656 transitions between them. The set Λ of language events contains all tuples $\langle \rho_i, \rho_j \rangle$, with $\{\rho_i, \rho_j\} \subseteq CFG(FM)$. Hence, $CF = \{\langle \rho_i, \langle \rho_i, \rho_j \rangle, \rho_j \rangle \mid \rho_i \neq \rho_j \wedge \{\rho_i, \rho_j\} \subseteq CFG(FM)\}$. The initial configuration ρ_{init} is set to be ρ_A (cf. Example 3.16).

Remark 4.21. A triggered adaptive language may omit transitions between some language variants. For instance, in educational applications, the language designer may not allow reconfigurations into language variants that are simpler than the current one. Hence, in practice, the variability space of interest may be much smaller than the space of all possible configurations (e.g., Figure 1 comprises just three language variants and two transitions). Thus, there is no need to define adapters from one language configuration to another that is not reachable in the transition system.

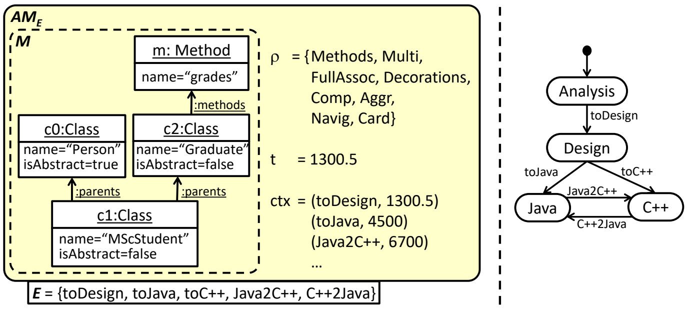
Next, we define *contextual adaptive models*, which store the current language configuration ρ and an instance model of the current meta-model MM_ρ. They are embedded in a *context* where the modelling activity aspects relevant for language reconfiguration purposes are represented as a sequence of timed events. For example, in a language that adapts to the IDE, the context may populate events when the screen size changes; in a language adaptive to a modelling process, the context may inform about the current phase; and in a language that adapts to the user knowledge, the context may store static background information about the user (e.g., years of modelling experience) or infer the expertise dynamically by counting the user errors when creating the model.

Definition 4.22 (Contextual adaptive model). Given a triggered adaptive language TAL_Λ and a set E of context events, a *contextual adaptive model* $AM_E = \langle \rho, M, type\colon M \longrightarrow MM_\rho, ctx, t \rangle$ is made of:

- A configuration $\rho \in CFG(FM)$, called the *current configuration*
- A model M typed over MM_ρ via morphism *type*
- A sequence $ctx \in (E \times \mathbb{R})^*(\{\bot_e\} \times \mathbb{R})$ of all relevant past and future context events, where $\bot_e \notin E$ is the final event, and the second component ($\mathbb{R}$) is the timestamp
- The current time t

Remark 4.23. The sequence ctx contains a (potentially infinite) succession of timestamped events from E, ending in a final event $\bot_e$. We use $ctx(i)$ to refer to its *i-th* element.

Example 4.24. Figure 14(a) shows a contextual adaptive model for our running example. It contains the current configuration ρ, a model typed by MM_ρ, the current time t, and a sequence ctx of context events. The current configuration corresponds to ρ_D, which configures the class diagram language for the design phase. The process model in Figure 14(b) specifies the possible project phases and how to transition between them. We assume that the modelling IDE generates the events in the process model transitions (i.e., toDesign, toJava, toC++, Java2C++, C++2Java) when the user selects the next modelling phase. This means that, effectively, the transition system of interest for our example (cf. Definition 4.19) comprises four language variants and five reconfigurations.



(a) (b)

Fig. 14. (a) Example of contextual adaptive model. (b) Process model that is used as context.

The last component is the adaptation trigger. A function *eval* produces the triggers based on the occurrence of context events, the current model, and the current configuration. If the context event is deemed relevant, the function returns a language event $\lambda \in \Lambda$ of the triggered adaptive language; otherwise, the function returns an event $\perp_\Lambda$ that does not belong to the language and is ignored.

Definition 4.25 (Adaptation trigger). Given a triggered adaptive modelling language TAL_Λ, and a contextual adaptive model AM_E, an *adaptation trigger* is a function $eval\colon E \times \mathbb{R} \times CFG(FM) \times SEM(MM) \longrightarrow \Lambda \cup \{\perp_\Lambda\}$. The input of the function is an event $e \in E$, the current time $t \in \mathbb{R}$, the current configuration $\rho \in CFG(FM)$, and the current model $M \in SEM(MM)$. The output of the function can be either a language event $\lambda \in \Lambda$ or an event $\perp_\Lambda \notin \Lambda$.

Example 4.26. The adaptation trigger of our example uses the set E = {toDesign, toJava, toC++, Java2C++, C++2Java}. Its function *eval*, defined below, translates events pertinent to the context (in this case a process model) into language events of the triggered adaptive language:

$$eval(e, t, \rho, M) = \begin{cases} \langle \rho_A, \rho_D \rangle & \text{if } e = toDesign \\ \langle \rho_D, \rho_J \rangle & \text{if } e = toJava \\ \langle \rho_D, \rho_C \rangle & \text{if } e = toC{+}{+} \\ \langle \rho_J, \rho_C \rangle & \text{if } e = Java2C{+}{+} \\ \langle \rho_C, \rho_J \rangle & \text{if } e = C{+}{+}2Java \\ \perp_\Lambda & \text{otherwise} \end{cases}$$

where ρ_C is a configuration like ρ_J (for Java, cf. Example 3.16), but enabling multiple inheritance.

Algorithm 1 implements the MAPE-K feedback loop that adapts a contextual adaptive model when context events occur. The algorithm receives as input a triggered adaptive language TAL_Λ, an adaptation trigger *eval*, and a contextual adaptive model AM_E. The latter may have been just initialised (with the empty model, the initial configuration ρ_{init} of TAL_Λ, and the current time 0) or be an existing model previously saved.

The algorithm modifies the input model as follows. Line 1 sets i (an index over the context events) to the first event with a timestamp equal to or greater than the current time t of the model. For models just created, the current time is 0, hence i is set to 0. Line 2 selects the next context event in the sequence (produced by an editing command or any other means). Lines 3–7 iteratively process the context events in the sequence while they are not final. Specifically, line 4 calls function *eval*, which returns a language event in Λ if the context event is relevant in the current configuration, and checks if the language's configuration transition system has a transition from the current

Algorithm 1 Adaptation of contextual adaptive models upon the occurrence of context events

```
Input: $TAL_\Lambda = \langle AL, CF, \rho_{init} \rangle$        ▷ Triggered adaptive language as in Def. 4.19
Input: $eval\colon E \times \mathbb{R} \times CFG(FM) \times SEM(MM) \longrightarrow \Lambda \cup \{\perp\}$        ▷ Adaptation trigger as in Def. 4.25
Input: $AM_E = \langle \rho, M, type\colon M \longrightarrow MM_\rho, ctx, t \rangle$        ▷ Contextual adaptive model as in Def. 4.22
1: $i \longleftarrow min\{j \mid \langle \epsilon, t' \rangle = ctx(j) \ \wedge \ t' \geq t\}$        ▷ search the next event to be processed
2: $\langle \epsilon, t \rangle \longleftarrow$ ctx(i)
3: while $\epsilon \neq \perp_e$ do
4:     if $\exists \langle \rho, eval(\epsilon, t, \rho, M), \rho' \rangle \in CF$ then
5:         $AM_E \longleftarrow \langle \rho', M', type'\colon M' \longrightarrow MM_{\rho'}, ctx, t \rangle$        ▷ with $M'$, $type'$ as in Def. 4.17
6:     $i \longleftarrow i + 1$
7:     $\langle \epsilon, t \rangle \longleftarrow$ ctx(i)
8: return $AM_E$
```

configuration labelled with that language event. If so, line 5 performs a language reconfiguration into ρ' (the target configuration of the identified transition), migrating the model as described in Definition 4.17, so that it becomes typed over $MM_{\rho'}$.

Our approach makes it possible to use the same triggered adaptive language with different contexts and adaptation triggers. This enables scenarios where the user explicitly selects a language reconfiguration (e.g., via a process model, as in the running example), or where reconfigurations are automatically applied when some conditions are met (e.g., evaluating OCL expressions on the current model when it is saved, whose satisfaction can trigger different language events).

5 SEQUENTIAL COMPOSITION OF ADAPTERS

Definition 4.17 assembles migration transformations out of adapters that tackle orthogonal language features (e.g., inheritance and associations in Figure 9). Still, further mechanisms are needed to avoid the combinatorial nature of feature interactions. In product lines, a feature interaction occurs when the behaviour of a feature is influenced by the presence of another one [67]. This section presents an optimisation to reduce the number of adapters required in an adaptive language definition, which is especially useful to tackle feature interactions within the language family.

In Figure 9, the adapter InhByDelegation deletes multiple inheritance assuming references. This assumption is needed because the adapter rules create references. However, if a language reconfiguration needs to delete multiple inheritance when the source and target configurations use full associations, then the language engineer would have to create another adapter for that case. The new adapter would tackle the change from Multi to Single inheritance assuming feature FullAssoc. Its rules would be like those of InhByDelegation, but creating full associations instead of references.

Figure 15 shows part of the example feature diagram, and represents the adapters as arrows indicating the feature changes they bridge. It can be noticed that, instead of defining another adapter to bridge Multi to Single when FullAssoc, it would be possible to apply InhByDelegation and then RefByAssoc (which replaces the created references by associations). This feature interaction happens because there are two mandatory, alternative feature sets (Inheritance and Style), and an adapter bridging two features of the first set needs to create elements of the second set. This sequential composition of adapters can also reduce the number of adapters needed to bridge features within an alternative set. For example, as Figure 15 shows, there is no need to define an adapter from Multi to No, but it suffices to apply first InhByDelegation and then SingleToNo.

This section extends the second step in Definition 4.17 (which collects the adapters compatible with a configuration diff) to select adapters that can be composed sequentially in a meaningful way,

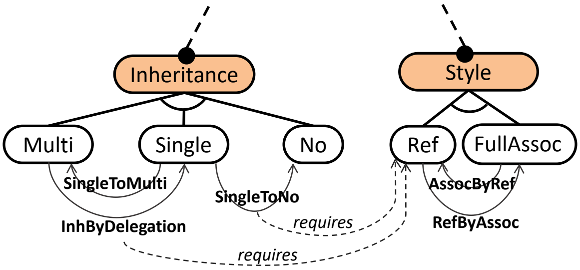


Fig. 15. Feature interactions between language adapters.

covering feature changes that individual adapters do not cover. For this purpose, we start defining the sequential composition of diffs. Two diffs Δ_1 and Δ_2 can be composed if the post-state of Δ_1 is coherent with the pre-state of Δ_2. For instance, a feature that changes to unselected in Δ_1 cannot change to unselected also in Δ_2, nor be assumed selected by Δ_2. The delta of the composed diff is the union of the changes of the first and second diffs, excluding the changes undone by the second diff and those that are synchronised. The context is the union of both contexts, excluding the features that the delta of the other diff changes, and including the features that the deltas synchronise (e.g., the features changed from + to − by Δ_1 and from − to + by Δ_2 are added to F_{12}^{++}).

Definition 5.1 (Sequential composition of diffs). Given diffs Δ_1 and Δ_2 s.t.

$$(F_1^{--} \cup F_1^{+-}) \cap (F_2^{++} \cup F_2^{+-}) = \emptyset \text{ and } (F_1^{++} \cup F_1^{-+}) \cap (F_2^{--} \cup F_2^{-+}) = \emptyset$$

their *sequential composition* is given by

$$\begin{aligned}\Delta_1;\Delta_2 =& \langle \delta_{12} = \langle (F_1^{+-} \setminus F_2^{-+}) \cup (F_2^{+-} \setminus F_1^{-+}), (F_1^{-+} \setminus F_2^{+-}) \cup (F_2^{-+} \setminus F_1^{+-}) \rangle,\\ & C_{12} = \langle (F_1^{++} \setminus F_2^{+-}) \cup (F_2^{++} \setminus F_1^{-+}) \cup (F_1^{+-} \cap F_2^{-+}), (F_1^{--} \setminus F_2^{-+}) \cup (F_2^{--} \setminus F_1^{+-}) \cup (F_1^{-+} \cap F_2^{+-}) \rangle \rangle\end{aligned}$$

Remark 5.2. We use predicate *composable*(Δ_1, Δ_2) to denote that diffs Δ_1 and Δ_2 can be composed according to Definition 5.1.

Example 5.3. Given diffs $\Delta_1 = \langle \delta_1 = \langle \{\text{Multi}\}, \{\text{Single}\} \rangle, C_1 = \langle \{\text{Ref}\}, \{\} \rangle \rangle$ and $\Delta_2 = \langle \delta_2 = \langle \{\text{Single}\}, \{\text{No}\} \rangle, C_2 = \langle \{\}, \{\text{Methods}\} \rangle \rangle$, their sequential composition is $\Delta_1;\Delta_2 = \langle \delta_{12} = \langle \{\text{Multi}\}, \{\text{No}\} \rangle, C_{12} = \langle \{\text{Ref}\}, \{\text{Single, Methods}\} \rangle \rangle$. The first diff changes from Multi to Single, and the second changes from Single to No, so their composition changes from Multi to No. As for the context, the resulting diff contains the union of the positive and negative contexts of the two diffs. In addition, Single is added to the negative context because it belongs to $F_1^{-+} \cap F_2^{+-}$.

The next lemma states that the sequential composition of two diffs yields a diff, and gives the conditions to obtain a wff diff out of the sequential composition of two diffs.

Lemma 5.4 (Wff diff composition). *Given diffs Δ_1 and Δ_2 s.t.* composable(Δ_1, Δ_2)*:*

- $\Delta_1;\Delta_2$ *is a diff*
- *If equations (1) and (2) below are satisfied, then $\Delta_1;\Delta_2$ is a wff diff*

$$\begin{aligned}\Psi[&true/(F_1^{+-} \setminus F_2^{-+}) \cup (F_2^{+-} \setminus F_1^{-+}) \cup (F_1^{++} \setminus F_2^{+-}) \cup (F_2^{++} \setminus F_1^{-+}) \cup (F_1^{+-} \cap F_2^{-+}),\\ &false/(F_1^{-+} \setminus F_2^{+-}) \cup (F_2^{-+} \setminus F_1^{+-}) \cup (F_1^{--} \setminus F_2^{-+}) \cup (F_2^{--} \setminus F_1^{+-}) \cup (F_1^{-+} \cap F_2^{+-})] \neq false\end{aligned} \tag{1}$$

$$\begin{aligned}\Psi[&true/(F_1^{-+} \setminus F_2^{+-}) \cup (F_2^{-+} \setminus F_1^{+-}) \cup (F_1^{++} \setminus F_2^{+-}) \cup (F_2^{++} \setminus F_1^{-+}) \cup (F_1^{+-} \cap F_2^{-+}),\\ &false/(F_1^{+-} \setminus F_2^{-+}) \cup (F_2^{+-} \setminus F_1^{-+}) \cup (F_1^{--} \setminus F_2^{-+}) \cup (F_2^{--} \setminus F_1^{+-}) \cup (F_1^{-+} \cap F_2^{+-})] \neq false\end{aligned} \tag{2}$$

Remark 5.5. If equations (1) and (2) in Lemma 5.4 are satisfied, then both Δ_1 and Δ_2 are wff. However, the converse is not true in general. We use predicate $\mathit{wffComposable}(\Delta_1, \Delta_2)$ to denote that diffs Δ_1 and Δ_2 are composable, and their composition is wff according to Lemma 5.4.

In Appendix A.3, we show that applying a composite diff yields the same result as applying each diff in sequence. Now, we define adapter composition. Given two adapters a and b whose diffs can be composed into a wff diff (i.e., $\mathit{wffComposable}(\Delta_a, \Delta_b)$), their composition yields an adapter *a; b* with diff $\Delta_a; \Delta_b$, containing the rules of both adapters, and whose regular expression concatenates the regular expressions of both adapters.

Definition 5.6 (Adapter composition). Given an adaptive language $AL = \langle LPL, A\rangle$, and two adapters $a, b \in A$ s.t. $\mathit{wffComposable}(\Delta_a, \Delta_b)$, the *composition* of a and b yields the adapter $a; b = \langle \Delta_a; \Delta_b, GTS = \langle MM, RS_a \cup RS_b, C_a; C_b\rangle\rangle$.

Example 5.7. Composing adapters InhByDelegation (with diff $\langle\langle\{\mathsf{Multi}\}, \{\mathsf{Single}\}\rangle, \langle\{\mathsf{Ref}\}, \{\}\rangle\rangle$) and SingleToNo (with diff $\langle\langle\{\mathsf{Single}\}, \{\mathsf{No}\}\rangle, \langle\{\mathsf{Ref}\}, \{\}\rangle\rangle$) yields adapter InhByDelegation; SingleToNo with diff $\langle\langle\{\mathsf{Multi}\}, \{\mathsf{No}\}\rangle, \langle\{\mathsf{Ref}\}, \{\mathsf{Single}\}\rangle\rangle$, the rules of both adapters, and the regular expression $(\mathsf{multiBySingle}+\mathsf{inhByRef})^*; (\mathsf{singleByRef})^*$. This expression executes first the rules of the first adapter as long as possible, followed by the rules of the second adapter. This phased execution is needed since rule multiBySingle creates parent links, which rule singleByRef of the second adapter deletes. Hence, the concatenation of the adapters' regular expressions avoids interferences between rules working on the same element types.

Definition 5.6 defines adapter composition for adaptive languages. The composition for triggered adaptive languages works the same way, by applying this definition to the adapters of the adaptive language within the triggered language. Note also that the *sequential* composition of adapters (using ";" in regular expressions) is complementary to their *parallel* composition in the migration transformation built in step 2 of Definition 4.17 (using "+" and star-iteration in regular expressions).

We could now modify the migration process in Definition 4.17 by searching sequential adapter compositions that bridge feature changes for which no specific adapter exists. However, this search can be expensive. Instead, we propose two adapter composition patterns able to solve the problems identified in Figure 15: *context fixers*, which handle dependencies between two alternative feature sets (e.g., Inheritance and Style), and *completers*, which bridge features in the same alternative set for which no adapter exists (e.g., Multi and No). As we will see later, these patterns are enough to organise transformations around *pivot features*, avoiding the creation of similar adapters.

Completers. A completer for a diff Δ_a within a diff Δ_{st} is a diff Δ_b such that the sequential composition $\Delta_a; \Delta_b$ yields a diff compatible with Δ_{st}. We distinguish *completers* from *soft completers*. The latter yield a diff that may not be compatible with the context of Δ_{st}, however, they are still useful because that context may be fixed with a context fixer (explained later).

Definition 5.8 (Completer). Given three diffs Δ_{st}, Δ_a and Δ_b, we define the predicates *SoftCompleter* and *Completer* as follows:

$$\begin{aligned} \mathit{SoftCompleter}(\Delta_a, \Delta_b, \Delta_{st}) \triangleq\ & \mathit{wffComposable}(\Delta_a, \Delta_b) \wedge \\ & (F_a^{+-} \subseteq F_{st}^{+-}) \wedge (F_a^{--} \subseteq F_{st}^{--}) \wedge \\ & (F_a^{-+} \setminus F_{st}^{-+} = F_b^{+-}) \wedge (F_b^{-+} \subseteq F_{st}^{-+}) \quad \textit{(Δ_b deactivates Δ_a's extra activations)} \end{aligned}$$

$$\begin{aligned} \mathit{Completer}(\Delta_a, \Delta_b, \Delta_{st}) \triangleq\ & \mathit{SoftCompleter}(\Delta_a, \Delta_b, \Delta_{st}) \wedge \\ & (F_a^{++} \subseteq F_{st}^{++}) \wedge (F_b^{++} \subseteq F_{st}^{++}) \wedge (F_b^{--} \subseteq F_{st}^{--}) \quad \textit{(contexts are compatible with Δ_{st})} \end{aligned}$$

We say that Δ_b is a *(soft) completer* for Δ_a within Δ_{st}.

Example 5.9. Figure 16 shows a completer for Δ_a (which moves from Multi to Single) within Δ_{st} (which changes feature No from unselected to selected). The first column displays whether each feature is initially selected (+) or not (-), and the subsequent columns depict the result of applying a diff. The completer Δ_b moves from Single to No, and is both a completer and a soft completer. Taking this into account, there is no need to build an adapter to move from Multi to No, but instead, we can sequentially compose an adapter whose diff goes from Multi to Single, with an adapter whose diff is the completer Δ_b.

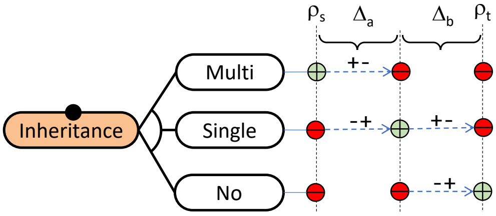


Fig. 16. Composing a diff Δ_a with a completer Δ_b to go from configuration ρ_s to ρ_t.

The next lemma states that completers do their job, that is, composing them yields a compatible diff with the given Δ_{st}.

LEMMA 5.10 (COMPOSING COMPLETERS). *Given diffs Δ_{st}, Δ_a and Δ_b s.t. $Completer(\Delta_a, \Delta_b, \Delta_{st})$, then $\Delta_a; \Delta_b \subseteq \Delta_{st}$.*

Context fixers. A context fixer for a diff Δ_a within a diff Δ_{st} is a diff Δ_b that repairs the context of Δ_a to make the resulting context of the sequential composition $\Delta_a; \Delta_b$ compatible with that of Δ_{st}. For this notion, we define a predicate *ContextFixer*, and three auxiliary ones: *FixerApplicable*, *PositiveFixer* and *NegativeFixer*. *FixerApplicable* checks if the delta of Δ_{st} includes Δ_a, the context of Δ_{st} includes Δ_b, and the deltas of Δ_a and Δ_b are independent. *PositiveFixer* checks if Δ_b can fix the positive context of Δ_a, i.e., unselects the features that Δ_a assumes positively but Δ_{st} does not. Conversely, *NegativeFixer* checks that Δ_b can fix the negative context of Δ_a.

Definition 5.11 (Context fixer). Given diffs Δ_{st}, Δ_a and Δ_b, predicate *ContextFixer* is defined as:

$$\begin{aligned} ContextFixer(\Delta_a, \Delta_b, \Delta_{st}) \triangleq\, & FixerApplicable(\Delta_a, \Delta_b, \Delta_{st}) \wedge \\ & (PositiveFixer(\Delta_a, \Delta_b, \Delta_{st}) \vee NegativeFixer(\Delta_a, \Delta_b, \Delta_{st})) \end{aligned}$$

with:

$$\begin{aligned} FixerApplicable(\Delta_a, \Delta_b, \Delta_{st}) \triangleq\, & wffComposable(\Delta_a, \Delta_b) \wedge \\ & (F_a^{+-} \cup F_a^{-+}) \cap (F_b^{+-} \cup F_b^{-+}) = \emptyset \wedge \;\; \textit{(delta of } \Delta_a \textit{ and } \Delta_b \textit{ are independent)} \\ & F_a^{+-} \subseteq F_{st}^{+-} \wedge F_a^{-+} \subseteq F_{st}^{-+} \wedge \;\; \textit{(delta of } \Delta_a \textit{ is included in } \Delta_{st}\textit{)} \\ & F_b^{++} \subseteq F_{st}^{++} \wedge F_b^{--} \subseteq F_{st}^{--} \;\; \textit{(context of } \Delta_b \textit{ is included in } \Delta_{st}\textit{)} \end{aligned}$$

$$\begin{aligned} PositiveFixer(\Delta_a, \Delta_b, \Delta_{st}) \triangleq\, & F_a^{--} \subseteq F_{st}^{--} \wedge \;\; \textit{(negative context of } \Delta_a \textit{ is included in } \Delta_{st}\textit{)} \\ & (F_a^{++} \setminus F_{st}^{++}) \subseteq F_b^{+-} \subseteq F_{st}^{--} \wedge \;\; \textit{(}\Delta_b \textit{ deactivates } \Delta_a\textit{'s extra positive context)} \\ & F_b^{-+} \subseteq F_{st}^{++} \;\; \textit{(}\Delta_b\textit{'s activations are compatible with required positive context)} \end{aligned}$$

$$\begin{aligned} NegativeFixer(\Delta_a, \Delta_b, \Delta_{st}) \triangleq\, & F_a^{++} \subseteq F_{st}^{++} \wedge \;\; \textit{(positive context of } \Delta_a \textit{ is included in } \Delta_{st}\textit{)} \\ & (F_a^{--} \setminus F_{st}^{--}) \subseteq F_b^{-+} \subseteq F_{st}^{++} \wedge \;\; \textit{(}\Delta_b \textit{ activates } \Delta_a\textit{'s extra negative context)} \\ & F_b^{+-} \subseteq F_{st}^{--} \;\; \textit{(}\Delta_b\textit{'s deactivations are compatible with required negative context)} \end{aligned}$$

We say that Δ_b is a *context fixer* for Δ_a within Δ_{st}.

Example 5.12. Given diffs $\Delta_{st} = \langle\langle\{\text{Multi}\}, \{\text{Single}\}\rangle, \langle\{\text{FullAssoc, Interfaces, Methods}\}, \{\text{Ref}\}\rangle\rangle$, $\Delta_a = \langle\langle\{\text{Multi}\}, \{\text{Single}\}\rangle, \langle\{\text{Ref}\}, \{\}\rangle\rangle$, and $\Delta_b = \langle\langle\{\text{Ref}\}, \{\text{FullAssoc}\}\rangle, \langle\{\}, \{\}\rangle\rangle$, we have that Δ_b is a context fixer for Δ_a within Δ_{st}. This is so as: (i) Δ_a and Δ_b can be composed ($wffComposable(\Delta_a, \Delta_b)$); (ii) the changes of Δ_a and Δ_b are disjoint; (iii) the delta of Δ_a is included in the delta of Δ_{st}; (iv) the context of Δ_b is included in the context of Δ_{st} (and so $FixerApplicable(\Delta_a, \Delta_b, \Delta_{st})$); (v) the negative context of Δ_a is included in the negative context of Δ_{st}; (vi) the positive context of Δ_a that Δ_{st} does not guarantee (Ref) is exactly F_b^{+-}, which is compatible with Δ_{st}'s negative context; and (vii) F_b^{-+} activates a feature (FullAssoc) in the positive context of Δ_{st} (and so $PositiveFixer(\Delta_a, \Delta_b, \Delta_{st})$).

In this example, the composition $\Delta_a; \Delta_b$ yields $\langle\langle\{\text{Multi, Ref}\}, \{\text{Single, FullAssoc}\}\rangle, \langle\{\}, \{\}\rangle\rangle$, which unselects Multi and Ref, and selects Single and FullAssoc. However, $\Delta_a; \Delta_b \nsubseteq \Delta_{st}$, since $\{\text{Multi, Ref}\} \nsubseteq \{\text{Multi}\}$, and $\{\text{Single, FullAssoc}\} \nsubseteq \{\text{Single}\}$. This is to be expected, since we are trying to apply Δ_a in an initial situation where the positive context of Δ_{st} (FullAssoc) is violated by Δ_a (which assumes Ref). Thus, an *implicit diff injector* $\Delta_{\vec{a}}$ of the form $\langle\langle\{\text{FullAssoc}\}, \{\text{Ref}\}\rangle, \langle\{\}, \{\}\rangle\rangle$ is needed. Figure 17 illustrates this situation, where Δ_a is not applicable to ρ_s since ρ_s does not have Ref initially selected. Hence, Δ_a is pre-composed with an *implicit injector* $\Delta_{\vec{a}}$, and post-composed with the *context fixer* Δ_b. Overall, Δ_b reverses the actions of $\Delta_{\vec{a}}$, but fixes the context of Δ_a.

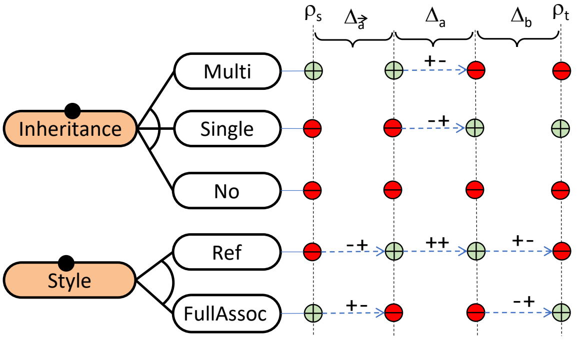


Fig. 17. Composing a diff Δ_a with its context fixer Δ_b and its implicit injector $\Delta_{\vec{a}}$.

The following lemma states the usefulness of context fixers, and introduces implicit injectors. A context fixer Δ_b for a diff Δ_a within Δ_{st} repairs the contextual expectations of Δ_a, so that, when pre-concatenated with the implicit injector $\Delta_{\vec{a}}$, we have $\Delta_{\vec{a}}; \Delta_a; \Delta_b \subseteq \Delta_{st}$.

Lemma 5.13 (Composing context fixers). *Given diffs Δ_{st}, Δ_a and Δ_b s.t. $ContextFixer(\Delta_a, \Delta_b, \Delta_{st})$, then $\Delta_{\vec{a}} = \langle\langle F_b^{-+}, F_b^{+-}\rangle, \langle\{\}, \{\}\rangle\rangle$ is the implicit diff injector of Δ_a.*

At this point, we need a mechanism for finding adapters whose diffs are context fixers for the diffs of other adapters. Given adapters a and b, and a diff Δ_{st} s.t. Δ_b is a context fixer for Δ_a within Δ_{st}, we use the notation $\vec{a} = (\Delta_{\vec{a}}, GTS = \langle MM, \{\}, \epsilon\rangle)$ for the empty injector adapter, which has $\Delta_{\vec{a}}$ as diff, and a graph transformation system without rules. In practice, we use an empty injector adapter when the first adapter a does not read elements of types activated by the features in $F_a^{++} \setminus F_{st}^{++}$ (for positive context fixers) or $F_a^{--} \setminus F_{st}^{--}$ (for negative ones). However, the adapter a is allowed to create elements of such types, since the second adapter b will take care of them. In our example, given adapter InhByDelegation and its context fixer RefByAssoc, we can use an empty injector adapter, since InhByDelegation creates references links, which have PC Ref (which is in $F_a^{++} \setminus F_{st}^{++}$). On the contrary, if the adapter a does not create, but reads or deletes elements of types activated by features in the unsatisfied context, then an empty injector is not enough, but it is necessary to find and apply an existing adapter *c* (with same diff as the implicit adapter) instead. This is so, as adapter *c* will introduce the elements activated by features in $F_a^{++} \setminus F_{st}^{++}$, so that adapter *a* can use them.

Algorithm 2 (migrAlg) uses completers and context fixers to provide an optimised version of the procedure to select suitable adapters for model migrations (step 2 in Definition 4.17). The algorithm receives an adaptive language and two (source and target) configurations as input, and returns the set of adapters to use in the migration transformation between both configurations as output. First (lines 1–3), the algorithm selects the set AD of all adapters consistent with the diff of the source and target configurations, just like in Definition 4.17. It also stores the features deactivated (U^{+-}) and activated (U^{-+}) by Δ_{st}, but which are not covered by the selected adapters (lines 4–5). Next, a loop traverses each adapter *a* not selected yet (lines 6–15). The loop first checks if all feature changes are covered (U^{+-} and U^{-+} are empty), in which case, the algorithm returns the current set AD of selected adapters (line 7). Otherwise, the loop searches for *context fixers* (lines 8–10), *completers* (lines 11–12) and *soft completers* that can be fixed (lines 13–15).

Algorithm 2 Extended migration generation using context fixers and completers (migrAlg)

```
Input: $AL = \langle LPL = \langle FM, MM, \Phi \rangle, A \rangle$                      ▷ Adaptive language as in Def. 4.15
Input: $\rho_s, \rho_t \in CFG(FM)$                                              ▷ Two configurations of FM
Output: Set(Adapter)                                  ▷ Set of adapters for migrating from $\rho_s$ to $\rho_t$
 1: $AD = \{a_k \in A \mid \Delta_k \subseteq \Delta_{st} \vee$
 2:         $(\Delta_k \sqsubseteq_{pre} \Delta_{st} \wedge \neg create(F_k^{++} \setminus F_{st}^{++}, F_k^{--} \setminus F_{st}^{--}, a_k)) \vee$
 3:         $(\Delta_k \sqsubseteq_{post} \Delta_{st} \wedge \neg delete(F_k^{++} \setminus F_{st}^{++}, F_k^{--} \setminus F_{st}^{--}, a_k))\}$      ▷ As in Def. 4.17
 4: $U^{+-} = F_{st}^{+-} \setminus \bigcup_{a_k \in AD} F_k^{+-}$                          ▷ Remaining + to - changes
 5: $U^{-+} = F_{st}^{-+} \setminus \bigcup_{a_k \in AD} F_k^{-+}$                          ▷ Remaining - to + changes
 6: for ($a \in A \setminus AD$) do
 7:     if ($U^{+-} = \emptyset \wedge U^{-+} = \emptyset$) then return $AD$
 8:     else if ($F_a^{+-} \subseteq U^{+-} \wedge F_a^{-+} \subseteq U^{-+} \wedge$               ▷ Looks for context fixers
 9:         $\exists b \in A \setminus AD \cdot$ ContextFixer($\Delta_a, \Delta_b, \Delta_{st}$) $\wedge$
10:         (inj=getInjector(a, b, $\Delta_{st}$)) $\neq$ null) then Update(inj;a;b)
11:     else if ($F_a^{+-} \subseteq U^{+-} \wedge F_a^{--} \subseteq F_{st}^{--} \wedge F_a^{-+} \nsubseteq U^{-+} \wedge$      ▷ Looks for completers
12:         $\exists b \in A \setminus AD \cdot$ Completer($\Delta_a, \Delta_b, \Delta_{st}$)) then Update(a;b)
13:     else if ($\exists b \in A \setminus AD \cdot$ SoftCompleter($\Delta_a, \Delta_b, \Delta_{st}$)$\wedge$      ▷ Looks for soft completers
14:             $\exists c \in A \setminus AD \cdot$ ContextFixer($\Delta_a; \Delta_b, \Delta_c, \Delta_{st}$)$\wedge$
15:             (inj=getInjector(a;b, c, $\Delta_{st}$)) $\neq$ null) then Update(inj;a;b;c)
16: return AD
17: function GETINJECTOR($a, b : Adapter, \Delta : Diff$) : Adapter
18:     if ($\neg$read($F_a^{++} \setminus F^{++}, F_a^{--} \setminus F^{--}$, a)) then return $(\langle\langle F_b^{-+}, F_b^{+-}\rangle, \langle\{\},\{\}\rangle\rangle, \langle MM, \{\}, \epsilon\rangle)$
19:     if ($\exists c \in AD \cdot (F_c^{+-} = F_b^{-+}) \wedge (F_c^{-+} = F_b^{+-}) \wedge (F_b^{++} = F_b^{--} = \emptyset)$) then return c
20:     else return null
21: function UPDATE($a : Adapter$) : void
22:     $AD = AD \cup \{a\}$
23:     $U^{+-} = U^{+-} \setminus F_a^{+-}$
24:     $U^{-+} = U^{-+} \setminus F_a^{-+}$
```

To check for *context fixers*, if the delta of the considered adapter *a* fits within U^{+-} and U^{-+} (line 8), there is a context fixer for it (line 9), and there is a suitable injector adapter (line 10), then the sequential composition of the injector, the adapter and the context fixer is added to the current adapter set AD, and the uncovered activated and deactivated features are updated (function Update in lines 21–24). Function getInjector is used to check for suitable injectors (lines 17–20). The function returns an empty injector if the adapter does not read context elements (line 18, checked using predicate *read* in Definition 4.13); otherwise, it returns an existing adapter with the same diff as the implicit injector (line 19), or null if none exists (line 20).

To check for *completers*, if the considered adapter a only fails in the activation F_a^{-+} (line 11) and there is a completer for it (line 12), then the sequential composition of the adapter and the completer is added to the current adapter set AD, and the uncovered activated and deactivated features are updated (line 12). If no completer exists, but a soft completer (line 13) for which there is a context fixer (line 14) and an injector (line 15), then the composed adapter is added to the adapter set AD, and the uncovered features are updated as before (line 15). Overall, the algorithm complexity is cubic on the number of adapters.

Remark 5.14. The algorithm checks for adapter compositions of length two (for context fixers and completers) or three (for soft completers). While this could be generalised to find longer compositions, it is enough to deal with feature interactions, and permits organising the transformations conceptually around "pivot features". A pivot feature is a feature in an alternative set of a feature diagram that: (a) has adapters to migrate to all other features in the same alternative set, and (b) the adapters of other alternative sets use the feature for their migrations. In our example, Ref is a pivot feature within Style since there are adapters to transform from Ref to FullAssoc, and the adapters handling the Inheritance alternative set use Ref. This permits context-fixing those adapters, if needed, with the adapter transforming from Ref to FullAssoc. However, the limitation on the composition length forces to organise the transformations within an alternative set in two steps. For example, in Figure 15, No is reachable from Multi in two steps. In general, this is always possible by choosing a pivot feature that is reachable, and can reach, all other features in one transformation step. In our example, Single is a pivot feature within Inheritance (there is no adapter from No to Single, but this is because there are no inheritance relationships to migrate).

Example 5.15. Given configurations $\rho_s = \{\text{Multi, FullAssoc}\}$ and $\rho_t = \{\text{No, FullAssoc}\}$, lines 1–5 of Algorithm 2 build sets $AD = \{\}$, $U^{+-} = \{\text{Multi}\}$, and $U^{-+} = \{\text{No}\}$. Lines 8–12 do not find context fixers or completers. Lines 13–15 find a soft completer (SingleToNo) for adapter InhByDelegation. It is a soft completer because, even though the positive context of none of the adapters is satisfied (since they require Ref), line 14 finds a context fixer (AssocByRef). Adapters InhByDelegation and SingleToNo create references (with PC Ref), but do not read them, so $\neg read(\{\text{Ref}\}, \{\}, \text{InhByDelegation})$ and $\neg read(\{\text{Ref}\}, \{\}, \text{SingleToNo})$. Therefore, method getInjector returns an empty injector inj in line 18, and the composition inj; InhByDelegation; SingleToNo; AssocByRef is added to AD. At this point, U^{+-} and U^{-+} are empty and the algorithm returns AD.

Overall, without Algorithm 2, the language engineer would need to manually define five adapters more: versions of InhByDelegation and SingleToNo assuming FullAssoc, a version of InterfacesToNo for single inheritance, an adapter from Multi to No assuming Ref, and a similar one assuming FullAssoc. Instead, our algorithm synthesises those adapters by composition of other adapters.

6 ANALYSIS

Next, we present analyses to check the correctness of adapters (Section 6.1); to measure the coverage of the set of possible migrations by the defined adapters (Section 6.2); and to assert whether a configuration is reachable from another one via non-empty migrations (Section 6.2).

6.1 Correctness of Adapters

Our migration transformation scheme yields models that are syntactically well-typed, since the model elements that are not typed by MM_t are removed in the last migration step (cf. Definition 4.17).

Nonetheless, the language designer may create rules that use elements of types that do not belong to all language configurations where the rule is applicable. As discussed in Example 4.11, these rules are syntactically correct as they are typed over the *150MM*. However, this may indicate a design error in the rule or in the adapter's diff. This section presents an analysis technique to detect these cases. We start defining the compatibility of model elements and models w.r.t. a diff (Definition 6.1), and then use this notion to define the compatibility at the rule level (Definition 6.3).

Definition 6.1 (Diff-model compatibility). Given a language product line $LPL = \langle FM, MM, \Phi\rangle$, a diff Δ over FM, a model M typed by MM via morphism $type$, and an element $x \in M$, we say that:

- x is *source-compatible* with Δ, written $src\text{-}compat_\Delta(x, M)$, if:
  $\Phi(type(x)) = false \ \vee \ \Phi(type(x))[true/(F^{+-} \cup F^{++}), false/(F^{-+} \cup F^{--})] = true$
- x is *target-compatible* with Δ, written $tar\text{-}compat_\Delta(x, M)$, if:
  $\Phi(type(x)) = false \ \vee \ \Phi(type(x))[true/(F^{-+} \cup F^{++}), false/(F^{+-} \cup F^{--})] = true$
- x is *compatible* with Δ, written $compat_\Delta(x, M)$, if: $src\text{-}compat_\Delta(x, M) \vee tar\text{-}compat_\Delta(x, M)$
- M is *source-compatible* with Δ, written $src\text{-}compat_\Delta(M)$, if: $\forall x \in M \cdot src\text{-}compat_\Delta(x, M)$
- M is *target-compatible* with Δ, written $tar\text{-}compat_\Delta(M)$, if: $\forall x \in M \cdot tar\text{-}compat_\Delta(x, M)$
- M is *compatible* with Δ, written $compat_\Delta(M)$, if: $\forall x \in M \cdot compat_\Delta(x, M)$

Remark 6.2. Definition 6.1 admits elements whose type's PC is false. This allows considering the case of auxiliary elements in meta-models, as is the case of the iface reference in our example, which is an auxiliary element for the transformation.

A model M source-compatible with Δ is ensured to be well-typed w.r.t. any meta-model derivable by any configuration in which Δ is applicable. Conversely, a target-compatible model M is well-typed w.r.t. any meta-model derivable by any configuration that may result from applying Δ. A compatible model M can have elements typed by meta-models of the source or target configurations.

Next, we define compatibility for rules and adapters. A rule compatible with a diff Δ has NACs whose elements are compatible with either the source or target configurations, may delete elements from the source configuration, preserves elements of any of the source or target configurations, and may create elements of the target configuration. An adapter is compatible with Δ, if all its rules are.

Definition 6.3 (Rule and adapter compatibility). Given an adaptive language $AL = \langle LPL, A\rangle$, an adapter $a \in A$, and a rule $tr = \langle L \xleftarrow{l} K \xrightarrow{r} R, NACS = \{L \xrightarrow{n_i} N_i\}_{i \in I}\rangle$ of a, we say that tr is *compatible* with a diff Δ, written $compat_\Delta(tr)$, if:

$$\begin{aligned}&(\forall n_i \colon L \rightarrow N_i \in \ NACS \cdot compat_\Delta(N_i)) \wedge \\ &src\text{-}compat_\Delta(L \setminus l(K)) \wedge compat_\Delta(K) \wedge tar\text{-}compat_\Delta(R \setminus r(K))\end{aligned}$$

The adapter a is *compatible* with a diff Δ', written $compat_{\Delta'}(a)$, iff $\forall tr \in RS \cdot compat_{\Delta'}(tr)$.

If a rule's K (which contains the preserved elements) is not compatible with Δ, then it may not be applicable in every configuration compatible with Δ (since the rule expects elements that cannot be present in the source or target configurations). If a rule's NAC N_i is not compatible with Δ, then it will always succeed (becoming useless), since N_i will never be present in the model. For the same reasoning, the elements deleted by the rule ($L \setminus l(K)$) should be source-compatible, and the elements created by the rule ($R \setminus r(K)$) should be target-compatible.

Example 6.4. Consider rule multiBySingle in Figure 9, defined by adapter InhByDelegation with $\Delta = \langle\langle\{\text{Multi}\}, \{\text{Single}\}\rangle, \langle\{\text{Ref}\}, \{\}\rangle\rangle$. The rule preserves objects p and c, of type Class, which has PC true, and so $compat_\Delta(K)$. The rule forbids elements with types Class and parent. The latter has PC Single, which evaluates to true in target configurations, and so $compat_\Delta(N_i)$. The rule deletes a parents reference (present in source configurations), and so, $src\text{-}compat_\Delta(L \setminus l(K))$. Finally, the rule creates a parent reference (present in target configurations), and so, $\textit{tar-compat}_{\Delta}(R \setminus r(K))$. Hence, overall, we have $\textit{compat}_{\Delta}$(multiBySingle).

Our compatibility notion is a heuristic to rule out errors, but a non-compatible rule may still be the intention of the language engineer. For instance, in Figure 9, rule inhByRef of adapter InhByDelegation creates a Role object and gives value to its attributes navig (with PC Navig), min and max (with PC Card), isComp (with PC Comp) and isAggr (with PC Aggr). Hence, we have $\neg\textit{tar-compat}_{\Delta}(R \setminus r(K))$. However, the rule is as intended, because giving value to these attributes avoids creating additional rules for cases where those features are individually selected. Instead, if these features are not selected, the last step of the migration will delete the corresponding attribute.

Next, we characterise the global correctness of our migration procedure, based on the local correctness of the adapters (compatibility with its Δ). The next lemma states that, if an adapter is compatible with its diff Δ, then it will be compatible with any diff Δ_{st} that makes the adapter be selected by the migration transformation of Definition 4.17.

Lemma 6.5 (Migration compatibility). *Let $AL = \langle LPL, A\rangle$ be an adaptive language; $\rho_s, \rho_t \in CFG(FM)$ be two configurations; and $a = \langle \Delta, GTS\rangle \in A$ be an adapter of LPL s.t. $compat_{\Delta}(a)$. Then:*

$$\begin{aligned}
\Delta \subseteq \Delta_{st} &\implies compat_{\Delta_{st}}(a)\\
\Delta \sqsubseteq_{pre} \Delta_{st} \wedge \neg create(F^{++} \setminus F^{++}_{st}, F^{--} \setminus F^{--}_{st}, a) &\implies compat_{\Delta_{st}}(a)\\
\Delta \sqsubseteq_{post} \Delta_{st} \wedge \neg delete(F^{++} \setminus F^{++}_{st}, F^{--} \setminus F^{--}_{st}, a) &\implies compat_{\Delta_{st}}(a)
\end{aligned}$$

Finally, Theorem 6.6 states that, if each adapter is compatible with its diff, then it is compatible with the diff of any source and target configurations over which it is selected by Algorithm 2.

Theorem 6.6 (Extended migration compatibility). *Let $AL = \langle LPL, A\rangle$ be an adaptive language s.t. $\forall a_k \in A \cdot compat_{\Delta_k}(a)$; and $\rho_s, \rho_t \in CFG(FM)$ be two configurations. Then, any adapter a_i returned by Algorithm 2 for ρ_s and ρ_t is compatible with Δ_{st} (i.e., $compat_{\Delta_{st}}(a_i)$).*

6.2 Migration Coverage and Configuration Reachability

Our migration approach can bridge any two language configurations even if the transformation between them lacks adapters, due to the initial model augmentation and final model restriction steps (cf. Definition 4.17). However, given a triggered adaptive language, it is important to understand which transitions within a configuration transition system CF use non-empty migration transformations (called *covered* transition system), and which ones use adapters that altogether cover all feature changes between their source and target configurations (called *totally covered*).

Definition 6.7 (Configuration transition system coverage). Given a triggered adaptive language $TAL_{\Lambda} = \langle AL, CF, \rho_{init}\rangle$, we define:

- *Covered transition system*: $C_{TAL_{\Lambda}} = \{(\rho_i, \lambda_{ij}, \rho_j) \in CF \mid migrAlg(AL, \rho_i, \rho_j) \neq \emptyset\}$
- *Totally covered transition system*: $TC_{TAL_{\Lambda}} = \{(\rho_i, \lambda_{ij}, \rho_j) \in CF \mid migrAlg(AL, \rho_i, \rho_j) = A_{ij} \wedge total(A_{ij}, \Delta_{ij})\}$

where $migrAlg$ corresponds to Algorithm 2, and predicate $total$ receives a set of adapters A and a diff Δ, and is defined as $total(A, \Delta) \triangleq (F^{+-} = \bigcup_{a_k \in A} F^{+-}_k) \wedge (F^{-+} = \bigcup_{a_k \in A} F^{-+}_k)$.

The analysis of configuration transition system coverage can help detecting missing adapters by uncovering migration transformations that are empty ($CF \setminus C_{TAL_{\Lambda}}$) or partial ($CF \setminus TC_{TAL_{\Lambda}}$). Hence, given a configuration ρ, one can obtain which configurations ρ_j can only be reached from ρ with empty migrations: $\{\rho_j \mid (\rho, \lambda, \rho_j) \in CF \setminus C_{TAL_{\Lambda}}\}$. Please note that, given a triggered language, its totally covered system is a subset of the covered one: $TC_{TAL_{\Lambda}} \subseteq C_{TAL_{\Lambda}} \subseteq CF$.

Example 6.8. The full (unrestricted) configuration transition system of our running example has 82 656 transitions. Our seven adapters cover 60 672 transitions (73.4%), and totally cover 960. Conversely, 21 984 transitions apply empty migrations (26.6%). If we restrict to the four configurations and five transitions in Figure 14(b), we can check that Design is reachable from Analysis using an empty migration; Java and C++ are reachable from Design using covered migrations; and Java and C++ are reachable from each other using totally covered migrations. Since Design only adds features Methods, Comp, Aggr, and Navig to configuration Analysis, it makes sense for the migration from Analysis to Design to be empty.

Next, we provide a way to analyse the coverage of feature changes by a set of adapters. It provides a global view of the reachability space via non-empty migrations, which is more compact than the previous analysis based on reachable configurations, since the number of configurations may be exponential on the number of features. Specifically, for each feature f, we collect the set of adapters whose diff requires the feature activation ($cov^{+-}(f)$) or deactivation ($cov^{-+}(f)$), and then calculate the percentage of covered activations and deactivations.

Definition 6.9 (Feature coverage). Given an adaptive language $AL = \langle LPL, A\rangle$ and a feature f, we define the *adapter coverage sets* for f as $cov^{+-}(f) = \{a \in A \mid f \in F^{+-}\}$ and $cov^{-+}(f) = \{a \in A \mid f \in F^{-+}\}$. The *feature coverage* of AL is then a percentage given by:

$$\frac{\sum_{f \in F}\left(nonEmpty(cov^{+-}(f)) + nonEmpty(cov^{-+}(f))\right)}{2 \times |F|} \times 100.0$$

where $nonEmpty(S) = 1$ if $|S| > 0$ and 0 otherwise.

Example 6.10. Our example has 12 selectable features, and so, 24 feature changes are possible (i.e., each feature can be individually selected or unselected). Our adapters cover 10 of these changes, which yields a feature coverage of 41.7%. On inspection, we note that no adapter activates features Methods, Decorations or their children Comp, Aggr, Navig, and Card. This is to be expected, since adding or removing methods or association decorations has no impact on migrations.

7 ARCHITECTURE AND TOOL SUPPORT

We have implemented our approach to adaptive languages atop the Merlin tool [25, 30], which allows defining LPLs (cf. Section 3.3). The new tool, called Merlin-a, extends Merlin to support adaptive languages, including the definition of language adapters, their analysis and composition, the synthesis of migration transformations, and the generation of adaptive modelling editors. The website http://miso.es/tools/merlin-adaptive/ permits downloading the tool, and includes installation and use instructions, as well as the case studies used in the evaluation of Section 8.

Merlin-a provides automation to build and use adaptive modelling languages using the process depicted in Figure 18, which involves the next steps, to be performed by the language engineer:

(1) *Define language variability.* First, the language variability is designed as a feature diagram. For this purpose, the language designer can use FeatureIDE [50].
(2) *Design language syntax.* As a second step, the abstract syntax of the LPL is defined via a *150MM*. This is just a regular Ecore meta-model, where the PCs are defined as annotations on the meta-model elements. The notion of meta-model that the tool supports is more expressive than the one in Definition 3.1, allowing cardinalities, inheritance and OCL constraints. Any Ecore editor could be used to define the *150MM*, but we recommend OCLinEcore[9] as it simplifies editing OCL constraints and annotations.

---

[9] https://wiki.eclipse.org/OCL/OCLinEcore

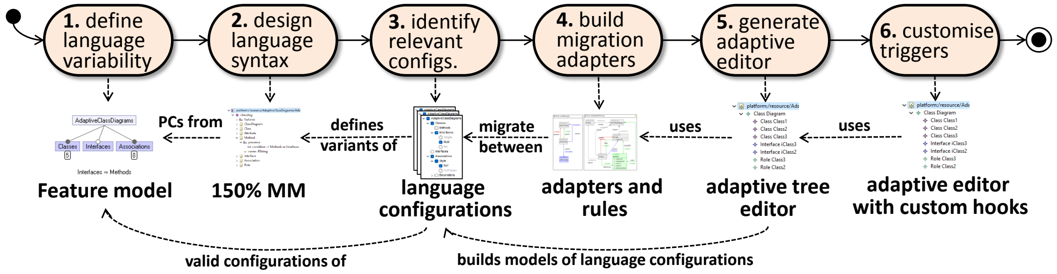


Fig. 18. Steps for generating an editor for an adaptive modelling language.

(3) *Identify relevant configurations.* The language designer specifies the subset of configurations that are relevant for the language, or alternatively, selects all configurations. In the former case, the individual configurations can be defined using FeatureIDE.

(4) *Build migration adapters.* The adapters to migrate between the relevant configurations are created. Merlin-a provides a textual DSL for their specification, and relies on the transformation language Henshin [5] to define the rules. At this stage, the language designer can use the analysis methods described in Section 6 to analyse: (i) the correctness of the adapters, and (ii) the coverage of the configurations of interest and the language features by the adapters. Section 7.1 will provide further details on the DSL and the supported analyses, and Section 7.2 will display screenshots of their use within Merlin-a.

(5) *Generate adaptive editor.* At this point, Merlin-a can automatically generate a modelling editor for the adaptive language. The editor permits creating models of the selected language variants, and migrates the models when the language variant in use changes.

(6) *Customise triggers.* Optionally, the language engineer can customise the editor with hook methods on GUI events, to trigger language reconfigurations. To facilitate regenerating the editor (step 5), but still preserve the manually added code, this manual code is encapsulated into event classes (e.g., `OnEdit`) with protected regions to prevent it from being overwritten.

This process does not need to be sequential, but may have iterations. For example, languages with many variants are typically developed iteratively, adding one or a few variants and their adapters in different iterations. As the following subsections will explain, our use of code generation techniques allows for quick editor re-generation while preserving any manually added code.

In the remainder of this section, we describe the tool architecture (Section 7.1), the facilities for defining adaptive languages (Section 7.2, steps 1–4), and those for generating and using the adaptive modelling language editors (Section 7.3, steps 5–6).

7.1 Architecture

Figure 19 shows the architecture of Merlin-a, which is an Eclipse plugin. It uses the Eclipse Modeling Framework (EMF) [68] as the underlying (meta-)modelling technology, Henshin [5] for creating the adapter rules, FeatureIDE [50] for defining and handling the language variability via feature models, and Xtext[10] to support specifying the adapters using a textual DSL.

Components 1 to 4 in Figure 19 support the definition of adaptive languages. Our tool relies on FeatureIDE (label 1) to handle the language variability. Merlin-a provides an extension to FeatureIDE that enables the definition of LPLs (label 2). Hence, in the first place, the language

---

[10] http://www.eclipse.org/Xtext/

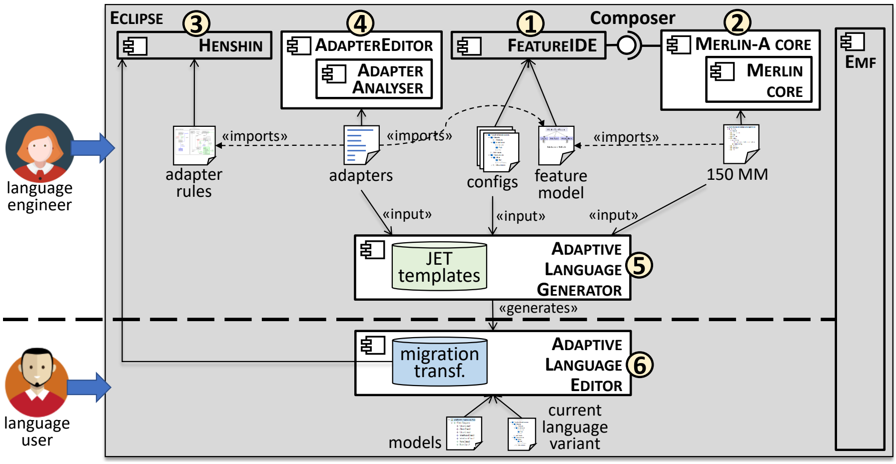


Fig. 19. Architecture of Merlin-A.

engineer needs to create a FeatureIDE project selecting the Merlin-A extension, define a feature model, and specify the feature configurations corresponding to the allowed language variants.

Then, the engineer must define the *150MM* with all language variants superimposed. The *150MM* is a regular Ecore meta-model, where the PCs are specified as annotations on the meta-model elements. Merlin-A relies on Merlin to validate the correctness of the specified *150MM*, both syntactically (e.g., no language variant has inheritance cycles) and semantically (e.g., all PCs are satisfiable, the OCL constraints in all variants are satisfiable). See [25, 30] for more details.

Next, the language engineer defines the adapter rules using Henshin (label 3), and the adapters themselves using a dedicated textual DSL (label 4). Figure 20 shows the meta-model of this DSL, whereby an AdapterModel has a name, stores the path of the ecore and Henshin files with the *150MM* and the rules, and comprises a collection of adapters. Each Adapter has a name, a set of rules, and a configuration diff (context and delta). Section 7.2 provides more details about the editor.

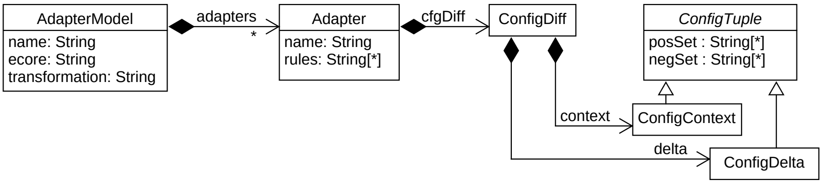


Fig. 20. Meta-model of the textual DSL for adapter definition.

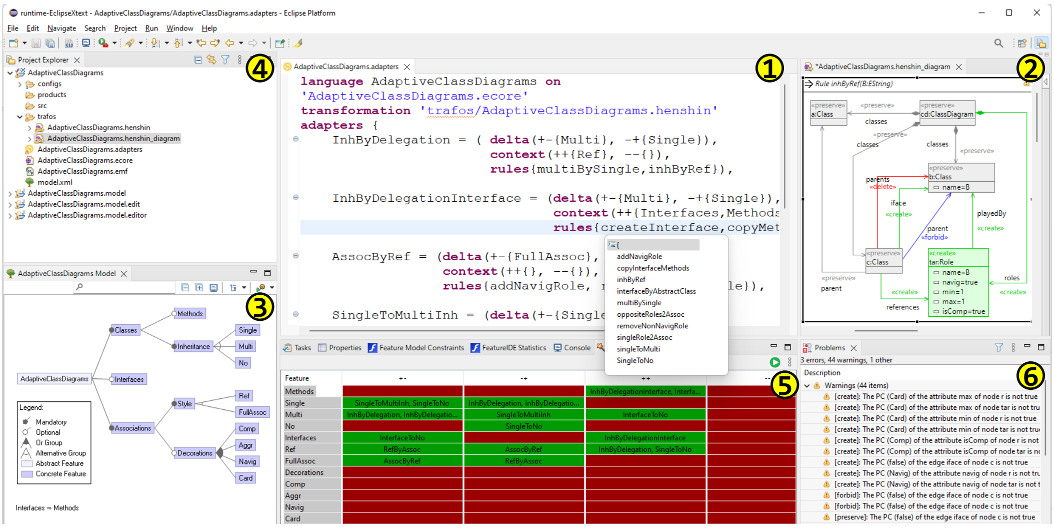
Merlin-A integrates an analyser (label 4 of Figure 19) that reveals non-compatible rules and the reasons for non-compatibility, as described in Section 6.1. In addition, the analyser reports on the adapters that use each language feature. This report is divided into deactivated (+-), activated (-+), positive (++) and negative context (--), depending on where the feature appears. This way, the analysis can be used to understand the *coverage* of feature (de)activation by adapters, as described in Section 6.2. If no adapter covers the activation of a feature, then a migration into a configuration where the feature is activated will not create elements whose type is guarded by the feature. Conversely, if no adapter covers the deactivation of a feature, then a migration into a configuration where the feature is deactivated will not handle elements whose type is guarded by the feature.

By default, these elements will be deleted by the migration. If a feature is not covered, it does not mean there is an error, but coverage serves to trace the language features explicitly considered by the migrations.

As Section 7.3 details, our tooling also integrates a generator of adaptive editors (label 5 of Figure 19). This is built atop the EMF generation facility for tree-based modelling editors. The generated editors (label 6) support language reconfiguration and model migration.

7.2 Tool Support: Definition of Adaptive Languages

Figure 21 shows Merlin-A being used to define the adaptive language of the running example. The panel with label 1 corresponds to the editor of the adapter definition DSL. The displayed listing specifies the language name (AdaptiveClassDiagrams) in line 1, the ecore file with the *150MM* in line 2, the Henshin file containing the rules in line 3, and then the adapters including their diff and the name of their Henshin rules. The editor features code completion on possible rule names (those defined in the Henshin file, cf. label 2) and feature names (those defined in the feature diagram, cf. label 3). It also integrates validators for the diffs, e.g., checking their well-formedness.



```
language AdaptiveClassDiagrams on
'AdaptiveClassDiagrams.ecore'
transformation 'trafos/AdaptiveClassDiagrams.henshin'
adapters {
    InhByDelegation = ( delta(+-{Multi}, -+{Single}),
                        context(++{Ref}, --{}),
                        rules{multiBySingle,inhByRef}),

    InhByDelegationInterface = (delta(+-{Multi}, -+{Single}),
                               context(++{Interfaces,Methods
                               rules{createInterface,copyMet

    AssocByRef = (delta(+-{FullAssoc},
                  context(++{}, --{}),
                  rules{addNavigRole, r                le}),

    SingleToMultiInh = (delta(+-{Single
```

Feature	+-	-+	++	--
Methods			InhByDelegationInterface, Interfa...	
Single	SingleToMultiInh, SingleToNo	InhByDelegation, InhByDelegatio...		
Multi	InhByDelegation, InhByDelegatio...	SingleToMultiInh	InterfaceToNo	
No		SingleToNo		
Interfaces	InterfaceToNo		InhByDelegationInterface	
Ref	RefByAssoc	AssocByRef	InhByDelegation, SingleToNo	
FullAssoc	AssocByRef	RefByAssoc		
Decorations				
Comp				
Aggr				
Navig				
Card				

```
3 errors, 44 warnings, 1 other
Description
Warnings (44 items)
    [create]: The PC (Card) of the attribute max of node r is not true
    [create]: The PC (Card) of the attribute max of node tar is not tru
    [create]: The PC (Card) of the attribute min of node r is not true
    [create]: The PC (Card) of the attribute min of node tar is not tru
    [create]: The PC (Comp) of the attribute isComp of node r is not
    [create]: The PC (Comp) of the attribute isComp of node tar is no
    [create]: The PC (false) of the edge iface of node c is not true
    [create]: The PC (Navig) of the attribute navig of node r is not tru
    [create]: The PC (Navig) of the attribute navig of node tar is not t
    [forbid]: The PC (false) of the edge iface of node c is not true
    [forbid]: The PC (false) of the edge iface of node c is not true
    [preserve]: The PC (false) of the edge iface of node c is not true
```

Fig. 21. Merlin-A in use for specifying the Class Diagrams adaptive language.

The panel with label 2 displays the Henshin editor. It allows creating the migration rules, which are typed by the *150MM*. FeatureIDE provides an editor for the feature diagram (label 3), another to create valid configurations, and tools to analyse the feature diagram. As the project explorer shows (label 4), these artefacts are stored within a FeatureIDE project.

The view with label 5 provides coverage information. It displays a matrix where the rows are the non-mandatory features, and the columns are possible uses of the feature within a diff (+-, -+, ++, --). Each cell shows the adapters that use the feature. Finally, the view with label 6 displays errors and warnings detected by the compatibility analysis of Definition 6.3.

7.3 Tool Support: Generation and Usage of Adaptive Language Editors

EMF provides built-in support to generate tree editors for Ecore-based languages by means of a model-to-text template language called Java Emitter Templates (JET)[11]. In particular, EMF provides

---

[11] https://projects.eclipse.org/projects/modeling.m2t.jet

a set of predefined JET templates that generate Java code implementing the editor for a given regular (i.e., non-adaptive) Ecore meta-model.

In Merlin-a, we have included a generator (label 5 in Figure 19) that overwrites those templates to extend the generated tree editor with support for language adaptation. The generator is invoked using a contextual menu. It receives a *150MM*, an adapter specification, and a set of feature configurations of interest, and synthesises a tree editor for the adaptive language together with migration transformations between the language variants corresponding to the given configurations.

The language users can use the generated editor to build models in the selected language variant (label 6 in Figure 19). The editor is an Eclipse plugin, and has a menu to select the language variant in use. This selection triggers the migration of the current model to the new language version. The editor dynamically inspects the current language variant and adapts its behaviour accordingly, hiding the menus and fields for creating and editing objects and features unsupported in the current language variant, and omitting the checking of the cardinality and OCL constraints absent from the current configuration. The editor includes *hook* methods that are called upon certain events, like saving or editing the model. The language designer can use these hooks to specify triggering conditions for language reconfigurations, e.g., based on the analysis of the user editing actions or the result of OCL queries evaluated on the model. Technically, we generate separate template hook classes (OnEdit, OnSave) with a common interface (IHook). To activate a hook, the language designer needs to fill in a method of these classes – which is generated empty – to perform actions when the event occurs. The common IHook interface has *default* methods with useful functionality, which can be called from the implementing classes. For example, it provides methods to execute OCL queries – passed as *Strings* – on particular objects or resources. The hook classes have *protected regions* that prevent overwriting the manually created code if the editor is regenerated.

Figure 22 displays some screenshots of the generated tree editor for the running example, where no hook code has been manually added. Label 1 shows the model-creation wizard, which extends the standard one with a combo-box to select the initial language configuration (Analysis in the figure). Label 2 shows the tree editor, which is used in the standard way to create models of the selected configuration. Our generator modifies the file name displayed in the top node of the model (after platform:) to display the current language version (Analysis). When modelling, the hooks are evaluated in the background and may trigger language reconfigurations. In addition, the editor includes by default a contextual menu Adaptation that permits changing to a different language configuration. When a language configuration is selected, the migration transformation is executed and the model updated (label 3). As an example, the figure shows the adaptation depicted in Figure 12 from Analysis to Java.

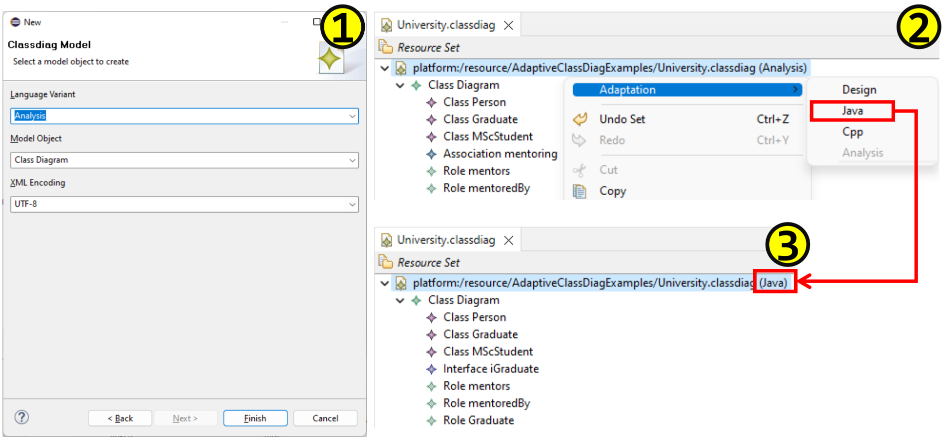
8 EVALUATION

Next, we evaluate the approach to answer the following research questions (RQs):

RQ1: *How feasible is it to specify adaptive languages in practice?*
RQ2: *How efficient is the adaptation process at runtime?*

To dig into RQ1, we compare the number of rules required by our approach, w.r.t. the number of rules required by a *naive approach* where each migration transformation is specified separately in an explicit way. Moreover, we analyse the reduction in the number of rules that our sequential composition of adapters brings. Hence, we study these two follow-up RQs:

RQ1.1: *What is the specification size reduction of using adapters w.r.t. a naive approach?*
RQ1.2: *What is the specification size reduction achieved by the sequential composition of adapters?*



New
Classdiag Model
Select a model object to create
Language Variant: Analysis
Model Object: Class Diagram
XML Encoding: UTF-8

Fig. 22. Generated adaptive (tree-based) model editor for the running example.

To answer RQ2, we measure the adaptation time of models of increasing size, for migrations between different variants of an adaptive language.

In the following, Sections 8.1 and 8.2 answer the RQs, and Section 8.3 discusses threats to validity.

8.1 RQ1: Specification Size of Adaptive Languages

8.1.1 Experiment design. To evaluate RQ1, we developed six case studies, available at https://miso.es/tools/merlin-adaptive/examples.html. They are families of well-known notations, variants of which have been reported in the literature, but never as adaptive languages.

- *Adaptive class diagrams.* This is the running example. It considers variants of class diagrams with/without interfaces, associations and methods, as well as variants with multiple, single, and no inheritance. The adaptation in this case is useful when using the language in different project phases (e.g., analysis, design, detailed design) or within a learning scenario. For the adaptation, we have designed adapters that bridge the different types of inheritance (using interfaces and delegation when moving from multiple to single inheritance, and interfaces are available), replace associations by simple references and vice versa, and substitute interfaces by abstract classes when the former are not available in a language variant.
- *Adaptive Petri nets.* The purpose of this adaptive language is to adapt the Petri net model to the user needs, moving to variants with sophisticated primitives when requiring a more expressive language, and to simpler variants when analysis capabilities are required. The language considers Petri nets [52] with tokens represented either as objects or as an integer attribute; arcs with/without weights; transitions with/without priority; variants with/without inhibitor, read and reset arcs; variants with/without bounded places; and variants with/without hierarchy. We have defined three sets of adapters. The first set moves from a complex to a simple variant, by expressing one primitive (e.g., read arcs) in terms of patterns of simpler primitives (e.g., parallel simple arcs in each direction). Hence, this set of adapters removes read arcs, weights from arcs, inhibitor arcs, bounded places, and the net hierarchy. The second set of adapters replaces patterns of a simple language variant by a primitive of a more sophisticated language variant. They detect arc loops to create read arcs, and parallel arcs to create weighted arcs. The third adapter set moves between alternative language realisations: tokens as objects or as attributes.

- *Adaptive process modelling.* We have built an adaptive process modelling language to fit different modelling scenarios. The language has variability on the available gateway types (parallel split, synchronisation, simple merge, exclusive choice, and multi-choice), the task types (hierarchical, initial and final, where the two latter can be mandatory or optional), and the representation of flows between elements either as intermediate objects or references. The adaptation capabilities enable changing the language style (with/without mandatory initial and final states, with flows represented as objects or relations) and the level of support for gateway types. Similar to the Petri nets case, we have defined adapters into simpler language variants, which replace complex gateways by patterns of primitives of simpler language variants.
- *Adaptive relational databases.* This adaptive language permits specifying database schemas, and optionally, their content data. The language has variants with/without primary and foreign keys; indices; and default values, unique values, and value auto-increment for columns. It also considers variants with either a closed set of data types, or an open set of data types represented as objects or attributes. The adapters bridge variants with open and closed data types. They also infer whether a column can be null from the available data, or be declared as unique.
- *Adaptive state machines.* This adaptive language has variants with a choice of the following features: transitions that are timed, have event triggers, or are immediate; guarded transitions and actions; hierarchical states, concurrent states, and states with entry, exit or do actions; pseudostates of types condition, (deep) history, and forks/joins; and executable machines. The adaptiveness permits moving between language variants tailored to the expressive power required at a certain moment. The defined adapters replace primitives by patterns: when exit actions are not available for states, these are moved to the output transitions (and similar for entry actions); condition pseudostates are replaced by standard transitions (concatenating the incoming and outgoing transitions); immediate transitions are replaced either by timed or event transitions depending on availability; and hierarchy is flattened when no longer available.
- *Adaptive multi-level modelling.* Multi-level modelling [27] permits modelling using any number of meta-levels, and not just two (meta-model and model). This results in simpler models in some scenarios [7]. Researchers have proposed different realisations of this approach [35], each with their own meta-modelling facilities and variants of them. To allow their inter-operability, we have designed an adaptive language which encompasses variants of the most common primitives within those multi-level proposals, provides different degrees of flexibility, and enables moving between variants depending on the modelling needs. For example, one may start using the primitives of one tool (e.g., Melanee [6]) and then change to another (e.g., with leap potency, as in MetaDepth [22]) when needed. At any point, the language can be adapted back (e.g., to Melanee), so that the adapters will express the unavailable primitives in terms of the available ones. Overall, the language allows choosing different degrees of conformance flexibility (e.g., cardinality checks, objects with abstract type), mechanisms for information extension (e.g., inheritance between objects, untyped objects and features), different flavours of potency (e.g., range [58], leap [26]), and the possibility to have multiple classifiers for objects, abstract classifiers, or assigning levels to models. The language adapters express abstract clabjects by using 0 potency; create appropriate subclasses to emulate multi-typing when multiple classification is no longer available; calculate model levels and element potency when those features become available; express leap potency with normal potency; and create proper types for untyped elements if these are disabled.

8.1.2 *Results.* Table 1 reports some metrics on the structure of the defined adaptive languages. The first column shows the language name; the next four columns report the size of the *150MM* in terms of the number of classes, attributes, references and PCs; and the last three columns characterise the language variability by the number of features of the feature model (in parenthesis, the number of non-mandatory ones, i.e., those that are *selectable*), alternative feature sets, and valid configurations. Overall, the *150MM* sizes range from 7 to 16 classes[12], from 1 to 14 attributes, from 7 to 15 references, and from 14 to 19 PCs. The feature models have between 14 and 26 features, leading to languages with 256 to 27 648 variants. Four adaptive languages have alternative feature sets. The class diagrams language has 2 alternative sets (cf. Figure 6), and the other languages have 0, 1 or 3.

Table 1. Metrics for the case studies: Structure.

	Size of 150MM				**Feature Model**		
Language	**Class**	**Attrs.**	**Refs.**	**PCs**	**Features (selectable)**	**Alternative feature sets**	**Configs.**
Class diagrams	7	13	14	16	17 (12)	2	288
Petri nets	7	6	15	18	14 (9)	1	256
Process modelling	11	1	7	14	21 (15)	3	1 920
Relational DDBB	9	12	14	19	16 (11)	1	576
State machines	16	12	7	19	21 (17)	0	12 288
Multi-level modelling	8	14	11	19	26 (19)	0	27 648

Table 2 focusses on the adaptiveness specifications of the languages. For each adaptive language, the first three columns show the number of language adapters, the total number of defined rules (in parenthesis, the average number of rules per adapter), and the feature coverage (percentage of activated or deactivated individual (selectable) features for which there is an explicit adapter, cf. Section 6.2). Then, the next two columns provide metrics on our mechanism for the sequential composition of adapters, counting the total number of context fixers and completers that this mechanism discovers (in parenthesis, the fixers and adapters that are unique, cf. Section 5). Finally, the last five columns report the total number of possible migration transformations between language variants (i.e., to go from each language variant to each other language variant), the migration transformations that are unique as a result of Algorithm 2, the average number of adapters per transformation, the average number of rules per transformation, and the total number of rules in the unique transformations.

Table 2. Metrics for the case studies: Adaptiveness.

	Language Adapters			**Seq. Composition**		**Migration Transformations**				
Language	**Adapt.**	**Rules (avg.)**	**Feature cover.**	**Fixers (unique)**	**Complet. (unique)**	**Possible**	**Unique**	**Average adapters**	**Average rules**	**Total rules**
Class diagrams	7	12 (1.7)	45.4%	3 (3)	2 (2)	82 656	22	1.4	3.0	66
Petri nets	10	18 (1.8)	61.1%	40 (4)	0 (0)	65 280	117	2.7	6.2	726
Process modelling	14	18 (1.3)	46.7%	4 064 (9)	0 (0)	3 684 480	2 609	5.4	13.4	34 961
Relational DDBB	6	11 (1.8)	31.8%	0 (0)	8 (2)	331 200	28	1.9	3.8	107
State machines	6	13 (2.2)	14.7%	0 (0)	0 (0)	150 982 656	139	3.3	7.2	1 001
Multi-level mod.	10	19 (1.9)	26.3%	0 (0)	0 (0)	764 384 256	319	4.0	5.5	1 755

The number of unique migration transformations is much lower than the total number of possible migrations (which is the number of configuration pairs). Transformations between pairs of configurations are identical if they select the same adapters. This is so if changes in some features, or the fact that some features remain selected or unselected, are irrelevant for the migration task. For instance, in our running example, it does not matter whether feature Aggr is selected or not, as

---

[12] The table reports one extra class and four extra references for the adaptive class diagrams compared to Figure 7(a). It corresponds to the root class that is customary in EMF meta-models, and the composition references this class defines.

migrations do not need to do anything special. Thus, migration transformations will be the same between two pairs of configurations that only differ in the selection value of feature Aggr.

We can observe in Table 2 that all adaptive languages required a moderate number of adapters (between 6 and 14) and rules (between 11 and 19), independently on the number of language configurations. We used our tool MERLIN-A to produce migration transformations between every two configurations. In the first four case studies, our optimised algorithm for sequential adapter composition generated between 2 and 9 unique context fixers or completers, which were reused from 2 to 4 064 times. These high numbers for Process modelling (9 unique context fixers, reused 4 064 times) is explained because this adaptive language has the largest number of alternative sets (3), and from the 4 languages with alternative sets, it has the highest number of configurations (1 920). This way, our migration mechanism was able to bridge many pairs of language configurations, ranging between 65 280 and more than 764 million. For this purpose, the algorithm generated between 22 and 2 609 unique transformations, by using between 1.4 and 5.4 adapters in average. In average, these transformations contain between 3 and 13.4 rules.

8.1.3 *Answering RQ1.*

Next, we answer RQ1 and its follow-up questions.

RQ1: How feasible is it to specify adaptive languages in practice? The effort required to specify both the structure and adaptiveness of the adaptive languages is moderate. For the former, the overall size of the *150MMs* ranged from 33 to 54 elements (including classes, attributes, references and PCs). Regarding adaptiveness, the language specifications had between 6 and 14 adapters, and between 11 and 19 rules.

RQ1.1: What is the specification size reduction of using adapters w.r.t. a naive approach? The effort reduction of using adapters compared to the naive approach of defining each migration transformation by hand is considerable. For the case studies, the naive approach requires defining between 22 and 2 609 transformations, with an overall number of rules between 66 and 34 961. Instead, we created between 6 and 14 adapters per language, and an overall number of rules between 11 and 19.

RQ1.2: What is the specification size reduction achieved by the sequential composition of adapters? The composition mechanism created either context fixers or completers for the first four case studies, which were the cases with alternative feature sets. In these cases, our approach saved the construction of between 2 to 9 adapters. Defining those adapters manually would have meant an increase between 66.7% and 85.7% on the number of adapters defined.

8.2 RQ2: Adaptation Efficiency at Runtime

8.2.1 *Experiment design.*

To address this RQ, we measured the model migration time between variants of the same adaptive language, for models of increasing size. Specifically, we considered the adaptive class diagrams running example, and the five migrations between configurations Analysis, Design, Java and C++ depicted in Figure 14(b). For each configuration, we created 10 random models with 10, 50, 100, 200, 500 and 1 000 objects (10 models of each size). To ensure realism, we used probability distributions for the number of objects per type (classes, attributes, methods, interfaces, roles, associations), as reported in language usage studies for meta-models [8]. Additionally, 25% of the classes were randomly assigned between 1 and 3 parent classes (only 1 if the language variant did not support multiple inheritance, as is the case for Java). Similarly, 25% of the classes were randomly set to implement between 1 to 3 interfaces, if permitted by the configuration.

The experiments were executed on a Windows 11 machine with Intel iCore 9 CPU and 32Gb of RAM. To reduce possible effects of non-determinism (e.g., rule matches, operating system processes), we repeated each execution 10 times, restarting the tool, and taking the median of the times [38]. The raw data are available at: https://miso.es/tools/merlin-adaptive/runtimeEval.html.

8.2.2 Results. Table 3 shows the adaptation time, in milliseconds, for each migration and model size (more precisely, the medians of the migration execution time for the median of the 10 executions of the 10 models of each size). This time includes loading and executing the transformation, as well as the model augmentation/restriction steps (cf. Definition 4.17). The time does not include the generation of the migration transformations, as our implementation pre-computes and caches these transformations for each configuration transition of interest. Figure 23 shows the results graphically.

Table 3. Adaptation time (in ms) for models of increasing size of the adaptive class diagrams language.

	Model size					
Migration	**10**	**50**	**100**	**200**	**500**	**1 000**
Analysis-Design	4.75	2.75	2	3	3.5	4
Design-Java	5.75	9	15.75	108	2 937	39 170.75
Design-C++	3.5	3	5.75	14	175	1 225.5
Java-C++	2.25	2	2.5	2.75	3.25	3.75
C++-Java	4	11.5	12.75	22.75	110	654.75

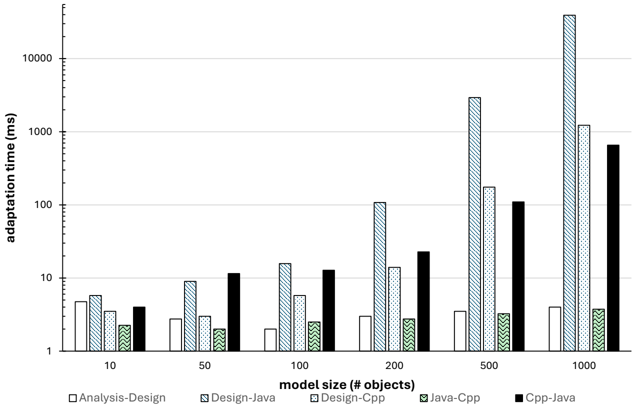


Fig. 23. Results of the experiment for RQ2 (vertical axis in logarithmic scale).

Overall, with the exception of two cases, the times are below 1.3 seconds. The Analysis-Design migration uses an empty transformation, making it one of the quickest. Both Design-Java and Design-C++ are non-totally covered migrations (cf. Example 6.8), and require converting full associations into references. However, Design-Java is significantly more costly as it needs to convert from multiple to single inheritance as well. For instance, Design-Java takes almost 3 seconds for models with 500 objects. Finally, both migrations between configurations Java and C++ are totally covered transformations (cf. Example 6.8). The Java-C++ migration is among the quickest ones, as it only involves a straightforward bridge between single and multiple inheritance. Instead, C++-Java is more time-consuming because it must convert from multiple to single inheritance.

8.2.3 Answering RQ2: How efficient is the adaptation process at runtime? In our experiment with models containing up to 1 000 objects, most adaptations are fluid, typically taking only a few milliseconds. The only exception is the Design-Java migration, where models of 500 objects have delays of almost 3 seconds, and those with 1 000 objects can take up to 39 seconds. This long adaptation time is due to the complex transformation required to emulate multiple inheritance with interfaces and delegation. This makes it highly sensitive to the number of inheritance relationships in the model. In our experiment, 25% of the classes were set to have inheritance. Reducing this to 15% yields median execution times of 24.5 seconds for models of size 1 000, while increasing it to 30% yields a median of 54.5 seconds. We argue that such large models are unlikely in this domain. Contrary to standard model-to-model transformations, designed to bridge likely very different languages, our migration transformations bridge variants of the same language, which typically results in fast adaptations.

8.3 Threats to Validity

Regarding internal validity, for RQ1, we created adapters between language features when this made sense. We cannot claim that it is not possible to define further adapters for some of the case studies, however, that would not change substantially the assessment on the feasibility of defining adaptive languages, or the gains to specify migrations w.r.t. a naive approach.

Regarding construct validity, RQ1.1 and RQ1.2 assess specification size reduction by measuring the decrease in the number of migration transformations and rules. However, these RQs do not evaluate effort reduction due to the use of adapters. For instance, our approach has the overhead of devising suitable adapters and their diffs, though this can be seen as a way to organise rules into migration transformations, which any naive approach should do manually in one way or another. Another possible overhead is related to testing the correctness of migrations. While we provide some analyses for adapters, we currently lack specific facilities for testing migrations within an adaptive language. Thus, while we argue that effort is correlated with specification size, only a user study can confirm this hypothesis. For RQ2, we used random models of increasing size, using probability distributions for the number of objects to emulate realistic models. Some migrations – notably Design-Java – are sensitive to model features like the number of inheritance relationships. We reported its effect, but perhaps other model characteristics may influence the execution time of other migrations. To reduce the effects of non-determinism in the execution times, we run each migration on each model 10 times, taking the median. Also for RQ2, our implementation pre-computes the migration transformations between the configurations of interest. It can be argued that other implementations may generate those migrations dynamically, in the adaptive editor. In any case, this generation time has very low impact in our experiment, with a median of 110 milliseconds.

With respect to external validity, the main threat for RQ1 is the limited number of case studies (six). To minimise this threat, we selected representative modelling languages targeting both structural system descriptions (class diagrams, relational schemas, multi-level modelling) and behaviour definition (Petri nets, process modelling, state machines). A related threat is the limited meta-model size of the case studies (between 7 and 16 classes). We argue that the main issue with specification scalability is not the size of the meta-model, but the size of the variability space (i.e., the number of language configurations) which in our evaluation ranges from 256 to 27 640. We reckon that larger meta-models may provide room for more variability, and new features may require additional adapters to bridge models of the new language variants. Still, in our case studies, the cost of building an adapter was relatively cheap, since each adapter required a low number of rules (between 1.3 and 2.2 in average). We hypothesise that the reason is that these transformations adapt models *within the same language*. Hence, they do not need to bridge wildly different languages, as might be the case for standard model-to-model transformations. While we expect that this is also the case for larger meta-models, stronger results would be obtained by more case studies, which we will tackle in future work. Similarly, the main external threat for RQ2 is the limited number of migrations measured (five). To mitigate this threat, we selected a variety of transformations (empty, covered, and totally covered).

9 RELATED WORK

Next, we revise related works on techniques to deal with families of modelling languages (Section 9.1), flexible modelling (Section 9.2), specification of adaptive systems (Section 9.3), mechanisms for model migration and transformation (Section 9.4), and configuration diffs (Section 9.5).

9.1 Families of Modelling Languages

Several researchers have recognised the usefulness of defining product lines of modelling languages to enable language reuse [14, 24, 30, 33, 55, 80]. They typically rely on feature models to represent the language variability, and use approaches either compositional (building the language out of components) [14, 24, 33] or annotative (building the language by removing elements) [30, 55, 80]. We opted for an annotative approach to facilitate defining adapters, since the rules are typed by the *150MM*. Adaptive languages go beyond LPLs because they consider adaptation triggers and model migration across language variants.

Transformational approaches to model variability, like delta-modelling [19], specify variants of a core model by a set of deltas that describe modifications on this core model [31]. Delta-modelling has been mainly applied to specify model variants [31]. Even though it can also be used to specify meta-model variants [56], to support a notion akin to adaptive languages, it should be complemented with corresponding migrations at the model level, and triggers for language reconfiguration.

Multi-level modelling [27] can also be used to define language families as specialisations of a generic language. In [23], we combined a product-line approach with multi-level modelling to enable the customisation of generic languages, which can be specialised via instantiation. However, that approach does not consider model migrations or adaptation triggers.

Close to our motivation, Hedy [32] is a Python-based gradual programming language for children education. It has five increasing levels of sophistication, to be used as programming expertise is gained. Similarly, adaptive languages may define several language configurations to be used in a learning process. All variants of Hedy are compiled into Python, but there is no transfer of programs between levels. Instead, adaptive languages support model migration across language variants.

Related to the previous work, van der Storm and Hermans [75] investigate the definition of textual gradual languages. Instead of building a parser for each language variant, they propose the gradual extension of grammars with (and deprecation of) syntactic constructs in consecutive levels, and syntax internationalisation. Our adaptive languages go beyond, since we consider migration between language versions (which do not need to be considered as a sequence of levels) and trigger mechanisms for language reconfigurations.

9.2 Flexible Modelling

Flexible modelling approaches [29] advocate the benefits of flexibility in modelling. They allow customising the *conformance* relationship, which enables the creation of modelling languages bottom-up [45, 85] or dealing with inconsistent models [29]. This makes modelling languages adaptable to different usages, from informal discussion to precise modelling aiming at code generation. This goal is in common with our notion of adaptive languages. However, flexible modelling approaches do not provide an explicit definition of language variants that offer users different primitive sets.

Kite [29] and Dandelion [49] are two flexible modelling tools that support the definition of process models governing the relaxations of the conformance checks to be made on a model w.r.t. its meta-model. While this can be seen as a light form of adaptation, these tools do not consider an explicit definition of language variants, or the migration of models between variants.

9.3 Modelling of Adaptive Languages and Systems

Self-adaptive systems [15, 42] modify their behaviour to achieve their goals. To do so, they exhibit a MAPE-K control loop to monitor their state and context, analyse whether an adaptation is required, choose the adaptation, and execute it. As described in Section 4.4, our triggered languages make this loop explicit to govern the modelling language adaptation. However, even if sharing similarities, many *self-** features of autonomic computing, like self-healing or self-protection, do not apply in our setting. Moreover, our setting involves one adaptive element (the language, with one control loop) and not distributed networks of adaptive elements.

There is extensive work on modelling for adaptive systems [15, 16, 84]. The modelled systems frequently perform their adaptations using a MAPE-K loop [42], which is explicit. From the modelling perspective, research lines include the proposal of requirement languages able to cope with the uncertainty of the adaptive systems [4, 81], or modelling languages to express adaptation strategies and utility functions and analyse their consequences [17, 53]. Instead, in adaptive languages, the system being adapted is the language itself. As Section 4.3 showed, our adaptation loop permits designers of adaptive languages to include adaptation triggers based on knowledge about, e.g., the modelling history, similar models, or language usage patterns.

Jouneaux et al. propose the notion of self-adaptive language [37] as a language that adapts its run-time semantics depending on contextual conditions, to obtain some trade-off. For example, a language that trades computation accuracy by execution time when the CPU load increases, or a robotics language that trades robot displacement time by energy saving. Adaptivity is achieved by incorporating feedback loops within the virtual machine [36], and prototype implementations are evaluated using Truffle. The authors propose a research roadmap, arguing that adaptations could also be supported at the language level by adding a language design feedback loop. They discuss that such a language adaptation could be based on a fixed set of features (as we do), or on an open set. The latter would allow adding new primitives to the language when discovering recurring patterns on how it is used. They propose a reference framework, called L-MODA, that considers both run-time and design-time feedback loops. Our notion of adaptive modelling language focuses on the design-time feedback loop, offering concrete mechanisms, architecture and tooling for its realisation. L-MODA envisions the utility of the design-time feedback loop for language evolution, such as adding features to a language by inspecting its actual usage. Instead, our motivation is flexibility of language usage. For this purpose, we provide a closed set of variants (with their adaptation and migration mechanisms) adaptable to the language context of use (user background, device, modelling aim, etc.). Adaptation at run-time is complementary to our design-time language/model adaptation, and uses entirely different techniques and technologies. We plan to explore semantic variability of modelling languages in future work, as well as open syntactic variability.

Metamorphic languages [1] are a proposal to support different shapes of a DSL, like internal, external, or using fluent APIs. Instead, our adaptive languages enable language variants and adaptation among these. We plan to study adapting the concrete syntax in future work.

9.4 Model Migration and Model Transformation

A key aspect of adaptive languages is the need to build migration transformations across variants. Some dedicated transformation languages exist to facilitate migration, e.g., exploiting implicit copying mechanisms [59, 60]. We emulate this by using the *150MM* to type the models. Moreover, while migration languages consider one migration between two meta-models, adaptive languages need to consider migrations between a large set of variants.

Modifying a meta-model can cause its associated artefacts (models, transformations, code generators, editors) become obsolete and stop working [61]. To alleviate this problem, techniques to semi-automatically co-evolve those artefacts have been proposed, mainly for the adaptation of models after meta-model changes [18]. For example, Cicchetti et al. [18] produce model migration transformations out of a meta-model and its evolved version. Our setting is more complex as it involves many language variants. Thus, we propose the manual definition of adapters, and the automated composition of migration transformations for specific source and target meta-models.

Model transformation product lines [25] equip a given LPL with a product line of in-place transformations, which are built out of transformation fragments depending on PCs. Our migration problem is more complex as it involves source and target meta-models, and hence the variability is not only in the transformation source but also in the target.

Transformation approaches have also been applied to manipulate models with variability [63]. In such setting, the meta-model is fixed, and the model contains variability. Here, we deal with the converse problem: the meta-model has variability, and we seek migrations between models. Variability rules [70] have been proposed as a compact way to model similar rules. Instead, our rules are standard, but transformations are composed by selecting appropriate rules from adapters.

Our mechanisms for selecting and composing adapters build suitable transformations between two configurations. Automated chaining of transformations has been studied in [9, 10] for model-to-model transformations. While they use meta-model coverage as criterion for composing transformations, we use diffs to select the compatible adapters included in the transformations.

9.5 Diffs of Feature Model Configurations

Related to our approach to describe changes between feature configurations (diffs), in [79], the problem of moving between two configurations is formalised as a SAT problem. Differencing of feature models has been widely studied as well [2, 73], including the definition of consistency-preserving configuration operators for efficient product line configuration [34]. However, we are not aware of works dealing with diffs of configurations and their composition.

10 CONCLUSIONS AND FUTURE WORK

This paper has introduced the concept of adaptive modelling language, which comprises a family of language variants and mechanisms for reconfiguring the language and its instance models across variants. Adaptive languages enable a better fit to the user expertise, modelling process, or IDE. We have presented tool support and an evaluation on six case studies, showing the feasibility of the approach and its advantages w.r.t. specifying the migrations between language variants explicitly.

This paper has focused on the abstract syntax of languages, but the concrete syntax could be adapted as well. Just like web pages adapt to the client – loading less content, special menus or smaller images in mobile devices – the concrete syntax of a language should be adaptable. This goes beyond to having graphical syntaxes with different levels of detail, but the adaptation of the concrete and abstract syntax should be coordinated. Moreover, adaptive languages may exhibit syntaxes of different nature, like graphical, textual, tabular or conversational [54].

An important ingredient of adaptive languages is the adaptation triggering mechanism. In this respect, we plan to contribute a library of useful reconfiguration triggers that consider, e.g., recurring modelling errors, language usage, or the detection of patterns. We would also like to experiment with the application of adaptive languages with implicit triggers in practice.

Our evaluation suggests that it is technically feasible to build adaptive languages with many configurations. However, we identify some opportunities for enhancement. First, regarding expressiveness, our adapter definition language could be extended to specify overriding relations between adapters diffs, in the style of [25], to indicate that a more general diff overrides a more specific diff, or vice versa. Second, regarding analysability, it would be interesting to identify the adaptations that lead to information loss (e.g., when moving to a class diagram language variant without cardinalities). Likewise, it is also worth exploring the combination of operational adapters (e.g., based on rules) and declarative adapters (e.g., based on OCL) which might be used in a bidirectional way. Finally, we would like to investigate testing techniques for adapters.

ACKNOWLEDGMENTS

This work has been funded by the Spanish Ministry of Science with projects TED2021-129381B-C21, PID2021-122270OB-I00, and RED2022-134647-T.

REFERENCES

[1] Mathieu Acher, Benoît Combemale, and Philippe Collet. 2014. Metamorphic domain-specific languages: A journey into the shapes of a language. In *Onward!@SPLASH*. ACM, 243–253.

[2] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. 2012. Feature model differences. In *CAiSE (LNCS, Vol. 7328)*. Springer, 629–645.

[3] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan de Lara. 2022. Recommender systems in model-driven engineering: A systematic mapping review. *Softw. Syst. Model.* 21, 1 (2022), 249–280.

[4] Aradea, Iping Supriana, and Kridanto Surendro. 2023. ARAS: Adaptation requirements for adaptive systems. *Autom. Softw. Eng.* 30, 1 (2023), 2.

[5] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. 2010. Henshin: Advanced concepts and tools for in-place EMF model transformations. In *MoDELS (LNCS, Vol. 6394)*. Springer, 121–135.

[6] Colin Atkinson and Ralph Gerbig. 2016. Flexible deep modeling with Melanee. In *Modellierung (LNI, Vol. 255)*. GI, 117–122.

[7] Colin Atkinson and Thomas Kühne. 2008. Reducing accidental complexity in domain models. *Softw. Syst. Model.* 7, 3 (2008), 345–359.

[8] Önder Babur, Eleni Constantinou, and Alexander Serebrenik. 2024. Language usage analysis for EMF metamodels on GitHub. *Empir. Softw. Eng.* 29, 1 (2024), 23.

[9] Francesco Basciani, Mattia D'Emidio, Davide Di Ruscio, Daniele Frigioni, Ludovico Iovino, and Alfonso Pierantonio. 2020. Automated selection of optimal model transformation chains via shortest-path algorithms. *IEEE Trans. Software Eng.* 46, 3 (2020), 251–279.

[10] Francesco Basciani, Daniele Di Pompeo, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2021. Integrating semantic reasoning in information loss-based transformation chain rankers. In *SAC*. ACM, 1494–1503.

[11] Narasimha Bolloju and Felix S. K. Leung. 2006. Assisting novice analysts in developing quality conceptual models with UML. *Commun. ACM* 49, 7 (2006), 108–112.

[12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-driven software engineering in practice, Second edition.* Morgan & Claypool Publishers, San Rafael, California (USA).

[13] Léa Brunschwig, Esther Guerra, and Juan de Lara. 2022. Modelling on mobile devices. *Softw. Syst. Model.* 21, 1 (2022), 179–205.

[14] Arvid Butting, Jerome Pfeiffer, Bernhard Rumpe, and Andreas Wortmann. 2020. A compositional framework for systematic modeling language reuse. In *MoDELS*. ACM, 35–46.

[15] Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. 2009. Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar] (LNCS, Vol. 5525)*. Springer, 1–26.

[16] Betty H. C. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. 2009. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *MoDELS (LNCS, Vol. 5795)*. Springer, 468–483.

[17] Shang-Wen Cheng and David Garlan. 2012. Stitch: A language for architecture-based self-adaptation. *J. Syst. Softw.* 85, 12 (2012), 2860–2875.

[18] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. 2008. Automating co-evolution in model-driven engineering. In *EDOC*. IEEE Computer Society, 222–231.

[19] Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. 2015. Abstract delta modelling. *Math. Struct. Comput. Sci.* 25, 3 (2015), 482–527.

[20] Loris D'Antoni and Margus Veanes. 2021. Automata modulo Theories. *Commun. ACM* 64, 5 (2021), 86–95.

[21] Juan de Lara, Roswitha Bardohl, Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2007. Attributed graph transformation with node type inheritance. *Theor. Comput. Sci.* 376, 3 (2007), 139–163.

[22] Juan de Lara and Esther Guerra. 2010. Deep meta-modelling with MetaDepth. In *TOOLS (LNCS, Vol. 6141)*. Springer, 1–20.

[23] Juan de Lara and Esther Guerra. 2021. Language family engineering with product lines of multi-level models. *Formal Aspects Comput.* 33, 6 (2021), 1173–1208.

[24] Juan de Lara, Esther Guerra, and Paolo Bottoni. 2022. Modular language product lines: a graph transformation approach. In *MoDELS*. ACM, 334–344.

[25] Juan de Lara, Esther Guerra, Marsha Chechik, and Rick Salay. 2018. Model transformation product lines. In *MoDELS*. ACM, 67–77.

[26] Juan de Lara, Esther Guerra, Ruth Cobos, and Jaime Moreno-Llorena. 2014. Extending deep meta-modelling for practical model-driven engineering. *Comput. J.* 57, 1 (2014), 36–58.

[27] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. 2014. When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 12:1–12:46.

[28] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. 2006. *Fundamentals of algebraic graph transformation*. Springer.

[29] Esther Guerra and Juan de Lara. 2018. On the quest for flexible modelling. In *MoDELS*. ACM, 23–33.

[30] Esther Guerra, Juan de Lara, Marsha Chechik, and Rick Salay. 2022. Property satisfiability analysis for product lines of modelling languages. *IEEE Trans. Software Eng.* 48, 2 (2022), 397–416.

[31] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Klaus Müller, Bernhard Rumpe, Ina Schaefer, and Christoph Schulze. 2015. Systematic synthesis of delta modeling languages. *Int. J. Softw. Tools Technol. Transf.* 17, 5 (2015), 601–626.

[32] Felienne Hermans. 2020. Hedy: A gradual language for programming education. In *ICER*. ACM, 259–270.

[33] Katrin Hölldobler, Bernhard Rumpe, and Andreas Wortmann. 2018. Software language engineering in the large: towards composing and deriving languages. *Comput. Lang. Syst. Struct.* 54 (2018), 386–405.

[34] José Miguel Horcas, Daniel Strüber, Alexandru Burdusel, Jabier Martinez, and Steffen Zschaler. 2023. We're not gonna break it! Consistency-preserving operators for efficient product line configuration. *IEEE Trans. Software Eng.* 49, 3 (2023), 1102–1117.

[35] Santiago P. Jácome-Guerrero and Juan de Lara. 2020. *TOTEM*: Reconciling multi-level modelling with standard two-level modelling. *Comput. Stand. Interfaces* 69 (2020), 103390.

[36] Gwendal Jouneaux, Olivier Barais, Benoît Combemale, and Gunter Mussbacher. 2021. SEALS: A framework for building self-adaptive virtual machines. In *SLE*. ACM, 150–163.

[37] Gwendal Jouneaux, Olivier Barais, Benoit Combemale, and Gunter Mussbacher. 2021. Towards self-adaptable languages. In *Onward!@SPLASH*. ACM, 1–16.

[38] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. 2005. Benchmark precision and random initial state. In *SPECTS*. SCS, 182–196.

[39] 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. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[40] 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.

[41] Steven Kelly and Juha-Pekka Tolvanen. 2008. *Domain-specific modeling - Enabling full code generation*. Wiley.

[42] Jeffrey O. Kephart and David M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.

[43] Hans-Jörg Kreowski, Sabine Kuske, and Grzegorz Rozenberg. 2008. Graph transformation units - An overview. In *Concurrency, Graphs and Models, Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday (LNCS, Vol. 5065)*. Springer, 57–75.

[44] Remo Lemma, Michele Lanza, and Andrea Mocci. 2015. CEL: Touching software modeling in essence. In *SANER*. IEEE Computer Society, 439–448.

[45] Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2015. Example-driven meta-model development. *Softw. Syst. Model.* 14, 4 (2015), 1323–1347.

[46] Jochen Ludewig. 2003. Models in software engineering. *Softw. Syst. Model.* 2, 1 (2003), 5–14.

[47] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. 2013. What industry needs from architectural languages: A survey. *IEEE Trans. Software Eng.* 39, 6 (2013), 869–891.

[48] Nicolas Mangano, Thomas D. LaToza, Marian Petre, and André van der Hoek. 2014. Supporting informal design with interactive whiteboards. In *CHI*. ACM, 331–340.

[49] Francisco Martínez-Lasaca, Pablo Díez, Esther Guerra, and Juan de Lara. 2023. Engineering low-code modelling environments with Dandelion. In *MoDELS Companion*. IEEE, 14–18.

[50] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering software variability with FeatureIDE*. Springer. See also https://featureide.github.io/.

[51] 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.

[52] T. Murata. 1989. Petri nets: Properties, analysis and applications. *Proc. IEEE* 77, 4 (1989), 541–580.

[53] Juliane Päßler, Maurice H. ter Beek, Ferruccio Damiani, Silvia Lizeth Tapia Tarifa, and Einar Broch Johnsen. 2023. Formal modelling and analysis of a self-adaptive robotic system. In *iFM (LNCS, Vol. 14300)*. Springer, 343–363.

[54] Sara Pérez-Soler, Mario González-Jiménez, Esther Guerra, and Juan de Lara. 2019. Towards conversational syntax for domain-specific languages using chatbots. *J. Object Technol.* 18, 2 (2019), 5:1–21.

[55] 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, 1–7.

[56] Christopher Pietsch, Timo Kehrer, Udo Kelter, Dennis Reuling, and Manuel Ohrndorf. 2015. SiPL - A delta-based modeling framework for software product line engineering. In *ASE*. IEEE Computer Society, 852–857.

[57] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software product line engineering: Foundations, principles and techniques*. Springer-Verlag, Berlin, Heidelberg.

[58] Alejandro Rodríguez, Fernando Macías, Francisco Durán, Adrian Rutle, and Uwe Wolter. 2023. Composition of multilevel domain-specific modelling languages. *J. Log. Algebraic Methods Program.* 130 (2023), 100831.

[59] Louis M. Rose, Markus Herrmannsdoerfer, Steffen Mazanek, Pieter Van Gorp, Sebastian Buchwald, Tassilo Horn, Elina Kalnina, Andreas Koch, Kevin Lano, Bernhard Schätz, and Manuel Wimmer. 2014. Graph and model transformation tools for model migration - Empirical results from the transformation tool contest. *Softw. Syst. Model.* 13, 1 (2014), 323–359.

[60] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, Fiona A. C. Polack, and Simon M. Poulding. 2014. Epsilon Flock: a model migration language. *Softw. Syst. Model.* 13, 2 (2014), 735–755.

[61] Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2012. Coupled evolution in model-driven engineering. *IEEE Softw.* 29, 6 (2012), 78–84.

[62] Davide Di Ruscio, Dimitrios S. Kolovos, Juan de Lara, Alfonso Pierantonio, Massimo Tisi, and Manuel Wimmer. 2022. Low-code development and model-driven engineering: Two sides of the same coin? *Softw. Syst. Model.* 21, 2 (2022), 437–446.

[63] Rick Salay, Michalis Famelis, Julia Rubin, Alessio Di Sandro, and Marsha Chechik. 2014. Lifting model transformations to product lines. In *ICSE*. ACM, 117–128.

[64] Donald Sannella and Andrzej Tarlecki. 2012. *Foundations of algebraic specification and formal software development.* Springer.

[65] Ina Schaefer. 2010. Variability modelling for model-driven development of software product lines. In *VaMoS (ICB-Research Report, Vol. 37)*. Universität Duisburg-Essen, 85–92.

[66] Sirius. (last accessed in May 2024). https://www.eclipse.org/sirius/.

[67] Larissa Rocha Soares, Pierre-Yves Schobbens, Ivan do Carmo Machado, and Eduardo Santana de Almeida. 2018. Feature interaction in software product line engineering: A systematic mapping study. *Inf. Softw. Technol.* 98 (2018), 44–58.

[68] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional, Upper Saddle River, NJ.

[69] Harald Störrle. 2019. Modeling moods. In *MoDELS*. IEEE, 468–477.

[70] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. 2018. Variability-based model transformation: Formal foundation and application. *Formal Asp. Comput.* 30, 1 (2018), 133–162.

[71] Xiaoyuan Su and Taghi M. Khoshgoftaar. 2009. A survey of collaborative filtering techniques. *Adv. Artif. Intell.* 2009 (2009), 421425:1–421425:19.

[72] Gabriele Taentzer and Arend Rensink. 2005. Ensuring structural constraints in graph-based models with type inheritance. In *FASE (LNCS, Vol. 3442)*. Springer, 64–79.

[73] Thomas Thüm, Don S. Batory, and Christian Kästner. 2009. Reasoning about edits to feature models. In *ICSE*. IEEE, 254–264.

[74] UML 2017. UML 2.5.1 OMG specification. http://www.omg.org/spec/UML/2.5.1/.

[75] Tijs van der Storm and Felienne Hermans. 2022. Gradual grammars: Syntax in levels and locales. In *SLE*. ACM, 134–147.

[76] Boban Vesin, Rodi Jolak, and Michel R. V. Chaudron. 2017. OctoUML: An environment for exploratory and collaborative software design. In *ICSE Companion Volume*. IEEE Computer Society, 7–10.

[77] Michael von der Beeck. 1994. A comparison of statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (LNCS, Vol. 863)*. Springer, 128–148.

[78] Guido Wachsmuth. 2007. Metamodel adaptation and model co-adaptation. In *ECOOP (LNCS, Vol. 4609)*. Springer, 600–624.

[79] Jules White, Brian Dougherty, Douglas C. Schmidt, and David Benavides. 2009. Automated reasoning for multi-step feature model configuration problems. In *SPLC*, Vol. 446. ACM, 11–20.

[80] Jules White, James H. Hill, Jeff Gray, Sumant Tambe, Aniruddha S. Gokhale, and Douglas C. Schmidt. 2009. Improving domain-specific language reuse with software product line techniques. *IEEE Softw.* 26, 4 (2009), 47–53.
[81] Jon Whittle, Peter Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean-Michel Bruel. 2009. RELAX: Incorporating uncertainty into the specification of self-adaptive systems. In *RE*. IEEE Computer Society, 79–88.
[82] Enes Yigitbas, Simon Gorissen, Nils Weidmann, and Gregor Engels. 2023. Design and evaluation of a collaborative UML modeling environment in virtual reality. *Softw. Syst. Model.* 22, 5 (2023), 1397–1425.
[83] Zoe Zarwin, Marija Bjekovic, Jean-Marie Favre, Jean-Sébastien Sottet, and Henderik Alex Proper. 2014. Natural modelling. *J. Object Technol.* 13, 3 (2014), 4: 1–36.
[84] Ji Zhang and Betty H. C. Cheng. 2006. Model-based development of dynamically adaptive software. In *ICSE*. ACM, 371–380.
[85] Athanasios Zolotas, Nicholas Matragkas, Sam Devlin, Dimitrios S. Kolovos, and Richard F. Paige. 2019. Type inference in flexible model-driven engineering using classification algorithms. *Softw. Syst. Model.* 18, 1 (2019), 345–366.

A THEORY OF DIFFS, AND PROOFS

This appendix contains a theory of diffs as transformers of configurations, of diff composition, and provides the proofs of the lemmas, propositions and theorems in the paper.

A.1 Diffs as transformers of configurations

Diffs can be used as transformers on configurations, as Definition A.1 shows.

Definition A.1 (Diff application). Let FM be a feature model, $\Delta = \langle \delta = \langle F^{+-}, F^{-+} \rangle, C = \langle F^{++}, F^{--} \rangle \rangle$ be a wff diff, and $\rho \in CFG(FM)$ be a configuration of FM with F^{+} and F^{-} its sets of selected and unselected features. Diff Δ is *applicable* on ρ, written $\rho \models \Delta$, if:

(1) the diff changes are applicable: $(F^{+-} \subseteq F^{+}) \wedge (F^{-+} \subseteq F^{-})$
(2) the diff context is satisfied: $(F^{++} \subseteq F^{+}) \wedge (F^{--} \subseteq F^{-})$
(3) the post-state is consistent: $\Psi[true/((F^{+} \setminus F^{+-}) \cup F^{-+}), false/((F^{-} \setminus F^{-+}) \cup F^{+-})] = true$

Given a wff diff Δ, and a configuration $\rho \in CFG(FM)$ s.t. $\rho \models \Delta$, applying Δ to ρ, written $\Delta(\rho)$, yields configuration $\rho' = \langle (F^{+} \setminus F^{+-}) \cup F^{-+}, (F^{-} \setminus F^{-+}) \cup F^{+-} \rangle$.

Condition (1) in Definition A.1 states that for a diff Δ to be applicable to a configuration ρ, the selected features of the configuration should contain the features changing to false, and the unselected features should contain those changing to true. Condition (2) requires the context of the diff to be satisfied: the configuration should select the features within the positive context (F^{++}), and unselect those within the negative context (F^{--}). Finally, condition (3) requires the result of swapping the features in F^{+-} from true to false, and those in F^{-+} from false to true, to be consistent with the feature model. This ensures that the result from applying Δ to the configuration is also a configuration, as the following lemma captures.

Lemma A.2 (Diff application correctness). *Given a configuration $\rho \in CFG(FM)$ and a wff diff Δ s.t. $\rho \models \Delta$, then $\Delta(\rho) \in CFG(FM)$.*

Proof. Trivially by condition (3) in Definition A.1, which states exactly the condition for $\Delta(\rho)$ to be a configuration of FM (cf. Definition 3.15). □

As an observation, the wff conditions for diffs in Definition 4.3 are no substitute for condition (3) in Definition A.1. Instead, a diff whose pre-state or post-state is not wff is never applicable. This is captured by the next proposition.

Proposition A.3 (non-wff diffs are not applicable). *Given a feature model FM and a non-wff diff Δ, then $\nexists \rho \in CFG(FM)$ s.t. $\rho \models \Delta$.*

Proof. Let us assume Δ's pre-state is not wff. Then, according to Definition 4.3, $\Psi[true/(F^{+-} \cup F^{++}), false/(F^{-+} \cup F^{--})] = false$. But this means that there cannot be a configuration $\rho = \langle F^{+}, F^{-} \rangle$

that satisfies conditions (1) and (2) in Definition A.1, since if $F^{+-} \cup F^{++} \subseteq F^{+}$ and $F^{-+} \cup F^{--} \subseteq F^{-}$, then $\Psi[true/F^{+}, false/F^{-}] = false$.

Now, let us assume Δ's post-state is not wff. Then, according to Definition 4.3, $\Psi[true/(F^{-+} \cup F^{++}), false/(F^{+-} \cup F^{--})] = false$. However, given any $\rho \in CFG(FM)$, the resulting configuration $\Delta(\rho) = \langle (F^{+} \setminus F^{+-}) \cup F^{-+}, (F^{-} \setminus F^{-+}) \cup F^{+-} \rangle$ cannot satisfy condition (3) in Definition A.1. This is so as $F^{-+} \cup F^{++} \subseteq (F^{+} \setminus F^{+-}) \cup F^{-+}$ (since according to condition (2) in Definition A.1, $F^{++} \subseteq F^{+}$; and by Definition 4.1, $F^{++} \cap F^{+-} = \emptyset$) and $F^{+-} \cup F^{--} \subseteq (F^{-} \setminus F^{-+}) \cup F^{+-}$ (since according to condition (2) in Definition A.1, $F^{--} \subseteq F^{-}$; and by Definition 4.1, $F^{--} \cap F^{-+} = \emptyset$). Therefore, $\Psi[true/((F^{+} \setminus F^{+-}) \cup F^{-+}), false/((F^{-} \setminus F^{-+}) \cup F^{+-})] = false$. □

Lemma A.4 (Configuration diffs are wff). *Given $\rho_i, \rho_j \in CFG(FM)$, their configuration diff Δ_{ij}, constructed as in Definition 4.5, is wff w.r.t. FM.*

Proof. The pre-state (cf. Definition 4.3) is wff since Ψ is evaluated substituting a subset of F_i^{+} (i.e., $(F_i^{+} \cap F_j^{-}) \cup (F_i^{+} \cap F_j^{+})$) by true, and a subset of F_i^{-} ($(F_i^{-} \cap F_j^{+}) \cup (F_i^{-} \cap F_j^{-})$) by false. This cannot yield false because Ψ yields true when substituting the complete sets F_i^{+} and F_i^{-} by true and false, respectively. Similarly, the post-state is wff since Ψ is evaluated substituting $(F_i^{-} \cap F_j^{+}) \cup (F_i^{+} \cap F_j^{+}) \subseteq F_j^{+}$ by true, and $(F_i^{+} \cap F_j^{-}) \cup (F_i^{-} \cap F_j^{-}) \subseteq F_j^{-}$ by false, which cannot yield false. □

Configuration diffs are not only required to be wff, but they must also agree with the semantics of diff application (cf. Definition A.1). This way, any configuration diff Δ_{ij} must be applicable to ρ_i, resulting in ρ_j, as the next lemma describes.

Lemma A.5 (Application of configuration diffs). *Given $\rho_i, \rho_j \in CFG(FM)$, then $\rho_i \models \Delta_{ij}$ and $\Delta_{ij}(\rho_i) = \rho_j$.*

Proof. We start checking that $\rho_i = \langle F_i^{+}, F_i^{-} \rangle \models \Delta_{ij} = \langle \delta_{ij} = \langle F_i^{+} \cap F_j^{-}, F_i^{-} \cap F_j^{+} \rangle, C_{ij} = \langle F_i^{+} \cap F_j^{+}, F_i^{-} \cap F_j^{-} \rangle \rangle$ (cf. Definition A.1).

Conditions (1) and (2) of Definition A.1 are immediate, since we just need to show that $(F_i^{+} \cap F_j^{-}) \subseteq F_i^{+}$ and $(F_i^{-} \cap F_j^{+}) \subseteq F_i^{-}$ (for condition 1), and $(F_i^{+} \cap F_j^{+}) \subseteq F_i^{+}$ and $(F_i^{-} \cap F_j^{-}) \subseteq F_i^{-}$ (for condition 2).

For condition (3) in Definition A.1, we use the fact that $\langle F_i^{+}, F_i^{-} \rangle$ and $\langle F_j^{+}, F_j^{-} \rangle$ are two partitions of the set F of features (cf. Figure 24). This means we can express F_j^{+} as $(F_i^{+} \setminus F_j^{-}) \cup (F_i^{-} \cap F_j^{+})$, which can be rewritten into $(F_i^{+} \setminus F_j^{-}) \cup F^{-+}$ and then into $(F_i^{+} \setminus (F_i^{+} \cap F_j^{-})) \cup F^{-+}$ and $(F_i^{+} \setminus F^{+-}) \cup F^{-+}$.

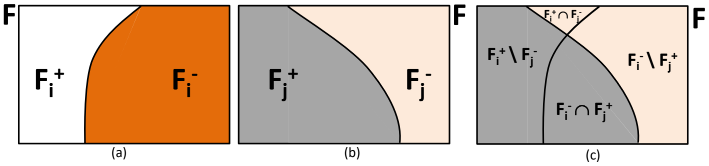


Fig. 24. Representation of (a) $\rho_i = \langle F_i^{+}, F_i^{-} \rangle$, and (b) $\rho_j = \langle F_j^{+}, F_j^{-} \rangle$, as partitions of set F. (c) Expressing $\rho_j = \langle F_j^{+}, F_j^{-} \rangle$ in terms of the intersections of partitions (a) and (b).

Similarly, we can express F_j^{-} as $(F_i^{-} \setminus F_j^{+}) \cup (F_i^{+} \cap F_j^{-})$, which then can be rewritten into $(F_i^{-} \setminus F_j^{+}) \cup F^{+-}$ and then into $(F_i^{-} \setminus (F_i^{-} \cap F_j^{+})) \cup F^{+-}$ and $(F_i^{-} \setminus F^{-+}) \cup F^{+-}$.

Since $\Psi[true/F_j^{+}, false/F_j^{-}] = true$, we have that $\Psi[true/((F_i^{+} \setminus F^{+-}) \cup F^{-+}), false/((F_i^{-} \setminus F^{-+}) \cup F^{+-})] = true$, and so $\rho_i \models \Delta_{ij}$. Moreover, we have already shown that $F_j^{+} = (F_i^{+} \setminus F^{+-}) \cup F^{-+}$ and $F_j^{-} = (F_i^{-} \setminus F^{-+}) \cup F^{+-}$, and therefore, $\Delta_{ij}(\rho_i) = \rho_j$, as desired. □

A.2 Lemma 5.4: *Wff diff composition*

Proof. To show that $\Delta_1;\Delta_2$ is a diff, according to Definition 4.1, we need to prove that $F_{12}^{+-} = (F_1^{+-} \setminus F_2^{-+}) \cup (F_2^{+-} \setminus F_1^{-+})$, $F_{12}^{-+} = (F_1^{-+} \setminus F_2^{+-}) \cup (F_2^{-+} \setminus F_1^{+-})$, $F_{12}^{++} = (F_1^{++} \setminus F_2^{+-}) \cup (F_2^{++} \setminus F_1^{-+}) \cup (F_1^{+-} \cap F_2^{-+})$ and $F_{12}^{--} = (F_1^{--} \setminus F_2^{-+}) \cup (F_2^{--} \setminus F_1^{+-}) \cup (F_1^{-+} \cap F_2^{+-})$ are disjoint. We proceed by parts.

Taking F_{12}^{+-}, we have that $(F_1^{+-} \setminus F_2^{-+})$ is disjoint with $(F_1^{-+} \setminus F_2^{+-})$, $(F_1^{++} \setminus F_2^{+-})$, $(F_1^{--} \setminus F_2^{-+})$ and $(F_1^{-+} \cap F_2^{+-})$ because Δ_1 is a diff, and its four sets are disjoint, and therefore subsets of these four sets are disjoint. Then, we need to prove that $(F_1^{+-} \setminus F_2^{-+})$ is disjoint with $(F_2^{-+} \setminus F_1^{+-})$, $(F_2^{++} \setminus F_1^{-+})$, $(F_2^{--} \setminus F_1^{+-})$ and $(F_1^{+-} \cap F_2^{-+})$. In the first case, it is disjoint since $(F_1^{+-} \setminus F_2^{-+}) \cap F_2^{-+} = \emptyset$, and therefore, $(F_1^{+-} \setminus F_2^{-+}) \cap (F_2^{-+} \setminus F_1^{+-}) = \emptyset$. In the second case, by Definition 5.1, we have that $(F_1^{--} \cup F_1^{+-}) \cap (F_2^{++} \cup F_2^{+-}) = \emptyset$, and therefore, $(F_1^{+-} \setminus F_2^{-+}) \cap (F_2^{++} \setminus F_1^{-+}) = \emptyset$ as requested. For the third case, we have that $F_1^{+-} \cap (F_2^{--} \setminus F_1^{+-}) = \emptyset$, and therefore, $(F_1^{+-} \setminus F_2^{-+}) \cap (F_2^{--} \setminus F_1^{+-}) = \emptyset$ as requested. Finally, $(F_1^{+-} \setminus F_2^{-+})$ is disjoint with $(F_1^{+-} \cap F_2^{-+})$ by the definition of set subtraction.

The disjointness of F_{12}^{-+}, F_{12}^{++} and F_{12}^{--} with the others can be proved similarly.

Proving that if equations (1) and (2) are satisfied, then $\Delta_1;\Delta_2$ is a wff diff, is immediate. This is so as equations (1) and (2) are exactly the requirements for $\Delta_1;\Delta_2$ to be wff. ☐

A.3 Diff composition correctness

Lemma A.6 states that applying a composite diff, and each diff in sequence, yield the same result.

Lemma A.6 (Diff composition correctness). *Given a feature model FM, a configuration $\rho \in$ CFG(FM), and two diffs Δ_1, Δ_2 s.t. wffComposable(Δ_1, Δ_2) and $\rho \models \Delta_1;\Delta_2$, then, $\Delta_1;\Delta_2(\rho) = \Delta_2(\Delta_1(\rho))$.*

Proof. On the one hand, we have that $\Delta_2(\Delta_1(\rho)) = ((F^+ \setminus F_1^{+-}) \cup F_1^{-+}) \setminus F_2^{+-} \cup F_2^{-+}$, which is equal to $(F^+ \setminus F_1^{+-}) \setminus F_2^{+-} \cup (F_1^{-+} \setminus F_2^{+-}) \cup F_2^{-+}$.

On the other hand, we have $\Delta_1;\Delta_2(\rho) = F^+ \setminus ((F_1^{+-} \setminus F_2^{-+}) \cup (F_2^{+-} \setminus F_1^{-+})) \cup (F_1^{-+} \setminus F_2^{+-}) \cup (F_2^{-+} \setminus F_1^{+-})$, which is equal to $(F^+ \setminus (F_1^{+-} \setminus F_2^{-+})) \setminus (F_2^{+-} \setminus F_1^{-+}) \cup (F_1^{-+} \setminus F_2^{+-}) \cup (F_2^{-+} \setminus F_1^{+-})$.

The term $(F_1^{-+} \setminus F_2^{+-})$ is in both expressions. In the second one, we can express $(F^+ \setminus (F_1^{+-} \setminus F_2^{-+})) \setminus (F_2^{+-} \setminus F_1^{-+})$ as $(F^+ \setminus F_1^{+-}) \setminus F_2^{+-} \cup (F^+ \cap F_2^{-+} \cap F_1^{+-}) \cup ((F^+ \setminus F_1^{+-}) \cap (F_2^{+-} \cap F_1^{-+}))$. The term $(F^+ \setminus F_1^{+-}) \setminus F_2^{+-}$ is now common in both expressions.

Now, we only need to show that F_2^{-+} (from the first expression) is equal to $(F^+ \cap F_2^{-+} \cap F_1^{+-}) \cup ((F^+ \setminus F_1^{+-}) \cap F_2^{+-} \cap F_1^{-+}) \cup (F_2^{-+} \setminus F_1^{+-})$. We have that $(F^+ \setminus F_1^{+-}) \cap (F_2^{+-} \cap F_1^{-+}) = \emptyset$, since F^+ and F_1^{-+} are disjoint. Since $F_1^{+-} \subseteq F^+$, we have that $F^+ \cap F_2^{-+} \cap F_1^{+-} = F_2^{-+} \cap F_1^{+-}$. Therefore, we have $F_2^{-+} = F_2^{-+} \cap F_1^{+-} \cup (F_2^{-+} \setminus F_1^{+-})$, as required. ☐

A.4 Lemma 5.10: *Composing completers*

Proof. We must show $\Delta_a;\Delta_b \subseteq \Delta_{st}$. For the delta, we have $\delta_{ab} = \langle (F_a^{+-} \setminus F_b^{-+}) \cup (F_b^{+-} \setminus F_a^{-+}), (F_a^{-+} \setminus F_b^{+-}) \cup (F_b^{-+} \setminus F_a^{+-}) \rangle$. By the definition of completer, $\delta_{ab} = \langle (F_a^{+-} \setminus F_b^{-+}) \cup (F_a^{-+} \setminus F_{st}^{-+}) \setminus F_a^{-+}, (F_a^{-+} \setminus (F_a^{-+} \setminus F_{st}^{-+})) \cup (F_b^{-+} \setminus F_a^{+-}) \rangle = \langle (F_a^{+-} \setminus F_b^{-+}), (F_a^{-+} \cap F_{st}^{-+}) \cup (F_b^{-+} \setminus F_a^{+-}) \rangle$. Since $F_a^{+-} \subseteq F_{st}^{+-}$ and $F_b^{-+} \subseteq F_{st}^{-+}$, we have $F_a^{+-} \setminus F_b^{-+} \subseteq F_{st}^{-+}$ and $(F_a^{-+} \cap F_{st}^{-+}) \cup (F_b^{-+} \setminus F_a^{+-}) \subseteq F_{st}^{-+}$, as required.

For the context, we have $C_{ab} = \langle (F_a^{++} \setminus F_b^{+-}) \cup (F_b^{++} \setminus F_a^{-+}) \cup (F_a^{+-} \cap F_b^{-+}), (F_a^{--} \setminus F_b^{-+}) \cup (F_b^{--} \setminus F_a^{+-}) \cup (F_a^{-+} \cap F_b^{+-}) \rangle$. We have $F_{ab}^{++} \subseteq F_{st}^{++}$ since $(F_a^{++} \setminus F_b^{+-}) \subseteq F_{st}^{++}$, $(F_b^{++} \setminus F_a^{-+}) \subseteq F_{st}^{++}$, and $(F_a^{+-} \cap F_b^{-+}) = \emptyset$. Similarly, we have $(F_a^{--} \setminus F_b^{-+}) \subseteq F_{st}^{--}$ and $(F_b^{--} \setminus F_a^{+-}) \subseteq F_{st}^{--}$. Since $F_a^{-+} \subseteq F_s^{-}$, and $F_b^{+-} \subseteq F_t^{-}$, then $F_a^{-+} \cap F_b^{+-} \subseteq F_{st}^{--}$, and so, $F_{ab}^{++} \subseteq F_{st}^{--}$ as required. ☐

A.5 Lemma 5.13: *Composing context fixers*

Proof. Since $\Delta_{\vec{a}}$ and Δ_b are inverse of each other, the diff $\Delta_{\vec{a}}; \Delta_a; \Delta_b$ has the changes of Δ_a, while Δ_b fixes Δ_a's unsatisfied context. We next prove the case of *PositiveFixer*, since the proof for *NegativeFixer* is analogous.

First, we check that $\Delta_{\vec{a}}$ and Δ_a are composable according to Definition 5.1. For this, we need to check that $(F_{\vec{a}}^{--} \cup F_{\vec{a}}^{+-}) \cap (F_a^{++} \cup F_a^{+-}) = \emptyset$. This holds since $F_{\vec{a}}^{--} = \emptyset$, $F_{\vec{a}}^{+-} = F_b^{-+}$, $F_b^{-+} \cap F_a^{++} = \emptyset$ (since *composable*(Δ_a, Δ_b)), and $F_b^{-+} \cap F_a^{+-} = \emptyset$ (since predicate *FixerApplicable* requires the actions of Δ_a and Δ_b to be disjoint). The proof of $(F_{\vec{a}}^{++} \cup F_{\vec{a}}^{-+}) \cap (F_a^{--} \cup F_a^{-+}) = \emptyset$ (the second part of Definition 5.1) is analogous.

Then, $\Delta_{\vec{a}}; \Delta_a = \langle\langle (F_b^{-+} \setminus F_a^{-+}) \cup (F_a^{+-} \setminus F_b^{+-}), (F_b^{+-} \setminus F_a^{+-}) \cup (F_a^{-+} \setminus F_b^{-+})\rangle, \langle (\emptyset \setminus F_a^{+-}) \cup (F_a^{++} \setminus F_b^{+-}) \cup (F_b^{-+} \cap F_a^{-+}), (\emptyset \setminus F_a^{-+}) \cup (F_a^{--} \setminus F_b^{-+}) \cup (F_b^{+-} \cap F_a^{+-})\rangle\rangle$. Simplifying, we have $\Delta_{\vec{a}}; \Delta_a = \langle\langle F_b^{-+} \cup F_a^{+-}, F_b^{+-} \cup F_a^{-+}\rangle, \langle F_a^{++} \setminus F_b^{+-}, F_a^{--} \setminus F_b^{-+}\rangle\rangle$.

Then, $\Delta_{\vec{a}}; \Delta_a$ and Δ_b are composable by Definition 5.1, which requires showing $((F_a^{--} \setminus F_b^{-+}) \cup (F_b^{-+} \cup F_a^{+-})) \cap (F_b^{++} \cup F_b^{+-}) = \emptyset$. By cases, we have that: (1) $(F_a^{--} \setminus F_b^{-+}) \cap F_b^{++} = \emptyset$, since *composable*$(\Delta_a, \Delta_b)$; (2) $(F_a^{--} \setminus F_b^{-+}) \cap F_b^{+-} = \emptyset$ for the same reason; (3) $(F_b^{-+} \cup F_a^{+-}) \cap F_b^{++} = \emptyset$ since F_b^{-+} and F_b^{++} are disjoint by Definition 4.1, and $F_a^{+-} \cap F_b^{++} = \emptyset$ since *composable*(Δ_a, Δ_b); and (4) $(F_b^{-+} \cup F_a^{+-}) \cap F_b^{+-} = \emptyset$ for the same reason. The proof for the 2^{nd} part of Definition 5.1 is analogous.

Then, the composed diff $\Delta_{\vec{a}}; \Delta_a; \Delta_b$ is $\langle\langle ((F_b^{-+} \cup F_a^{+-}) \setminus F_b^{-+}) \cup (F_b^{+-} \setminus (F_b^{+-} \cup F_a^{-+})), ((F_b^{+-} \cup F_a^{-+}) \setminus F_b^{+-}) \cup (F_b^{-+} \setminus (F_b^{-+} \cup F_a^{+-}))\rangle, \langle (F_a^{++} \setminus F_b^{+-}) \setminus F_b^{+-} \cup (F_b^{++} \setminus (F_b^{+-} \cup F_a^{-+})) \cup ((F_b^{-+} \cup F_a^{+-}) \cap F_b^{-+}), (F_a^{--} \setminus F_b^{-+}) \setminus F_b^{-+} \cup (F_b^{--} \setminus (F_b^{-+} \cup F_a^{+-})) \cup ((F_b^{+-} \cup F_a^{-+}) \cap F_b^{+-})\rangle\rangle$. Simplifying, we have $\Delta_{\vec{a}}; \Delta_a; \Delta_b = \langle\langle F_a^{+-}, F_a^{-+}\rangle, \langle (F_a^{++} \setminus F_b^{+-}) \cup (F_b^{++} \setminus F_a^{-+}) \cup F_b^{-+}, (F_a^{--} \setminus F_b^{-+}) \cup (F_b^{--} \setminus F_a^{+-}) \cup F_b^{+-}\rangle\rangle$.

It remains to show that $\Delta_{\vec{a}}; \Delta_a; \Delta_b \subseteq \Delta_{st}$. This is the case since, on the one hand, $F_a^{+-} \subseteq F_{st}^{+-}$ and $F_a^{-+} \subseteq F_{st}^{-+}$ because *FixerApplicable*$(\Delta_a, \Delta_b, \Delta_{st})$. On the other hand, the context is also satisfied. First, $F_a^{++} \setminus F_b^{+-} \subseteq F_{st}^{++}$. Since *PositiveFixer*$(\Delta_a, \Delta_b, \Delta_{st})$, we have $(F_a^{++} \setminus F_{st}^{++}) \subseteq F_b^{+-} \subseteq F_{st}^{--}$. This means that $F_b^{+-} \cap F_{st}^{++} = \emptyset$, and so $(F_a^{++} \setminus F_b^{+-}) = F_a^{++} \cap F_{st}^{++} \subseteq F_{st}^{++}$. For the positive context, we also need to show $F_b^{++} \setminus F_a^{-+} \subseteq F_{st}^{++}$ (which holds by predicate *FixerApplicable*), and $F_b^{-+} \subseteq F_{st}^{++}$ (which holds by predicate *PositiveFixer*). Regarding the negative context, we have $(F_a^{--} \setminus F_b^{-+}) \subseteq F_{st}^{--}$ (which holds by predicate *PositiveFixer*, which requires $F_a^{--} \subseteq F_{st}^{--}$), $(F_b^{--} \setminus F_a^{+-}) \subseteq F_{st}^{--}$ (since $F_b^{--} \subseteq F_{st}^{--}$ by predicate *FixerApplicable*), and $F_b^{+-} \subseteq F_{st}^{--}$ (by predicate *PositiveFixer*). □

A.6 Lemma 6.5: *Migration compatibility*

Proof. We deal with each of the three cases:

(1) $\Delta \subseteq \Delta_{st} \implies compat_{\Delta_{st}}(a)$
Given a rule tr of adapter a, by Definition 6.3 of $compat_{\Delta}(tr)$, we have $src\text{-}compat_{\Delta}(L \setminus l(K))$, and so $\forall x \in (L \setminus l(K)) \cdot \Phi(type(x)) = false \vee \Phi(type(x))[true/(F^{+-} \cup F^{++}), false/\ (F^{-+} \cup F^{--})] = true$. Since $\Delta \subseteq \Delta_{st}$ we have $F^X \subseteq F_{st}^X$ for $X \in \{+-, -+, ++, --\}$. This means that $(F^{+-} \cup F^{++}) \subseteq (F_{st}^{+-} \cup F_{st}^{++})$ and $(F^{-+} \cup F^{--}) \subseteq (F_{st}^{-+} \cup F_{st}^{--})$. Hence, given $x \in (L \setminus l(K))$, either $\Phi(type(x)) = false$, or else, substituting a larger set of features cannot change the valuation of $\Phi(type(x))[true/(F_{st}^{+-} \cup F_{st}^{++}), false/(F_{st}^{-+} \cup F_{st}^{--})]$ from *true* to *false*, and hence, $src\text{-}compat_{\Delta_{st}}(L \setminus l(K))$. A similar reasoning follows for $tar\text{-}compat_{\Delta}(R \setminus r(K))$, $compat_{\Delta}(K)$, and $compat_{\Delta}(N_i)$.

(2) $\Delta \sqsubseteq_{pre} \Delta_{st} \wedge \neg create(F^{++} \setminus F_{st}^{++}, F^{--} \setminus F_{st}^{--}, a) \implies compat_{\Delta_{st}}(a)$
Since $\Delta \sqsubseteq_{pre} \Delta_{st}$, we have $F^X \subseteq F_{st}^X$ for $X \in \{+-, -+\}$, $F^{++} \subseteq F_{st}^{++} \cup F_{st}^{+-}$, and $F^{--} \subseteq F_{st}^{--} \cup F_{st}^{-+}$. Like in the previous case, this means $(F^{+-} \cup F^{++}) \subseteq (F_{st}^{+-} \cup F_{st}^{++})$ and $(F^{-+} \cup F^{--}) \subseteq (F_{st}^{-+} \cup F_{st}^{--})$. Hence, for any rule tr of a, we have $src\text{-}compat_{\Delta_{st}}(L \setminus l(K))$, $src\text{-}compat_{\Delta_{st}}(K)$ and $src\text{-}compat_{\Delta_{st}}(N_i)$ (for each NAC N_i). But this means that $compat_{\Delta_{st}}(K)$ and $compat_{\Delta_{st}}(N_i)$ (for each NAC N_i). Since $\neg create(F^{++} \setminus F_{st}^{++}, F^{--} \setminus F_{st}^{--}, a)$, then each element in $R \setminus r(K)$ is not typed by $F^{++} \setminus F_{st}^{++}$ or $F^{--} \setminus F_{st}^{--}$, hence $tar\text{-}compat_{\Delta_{st}}(R \setminus r(K))$, and so $compat_{\Delta_{st}}(a)$ as required.

(3) $\Delta \sqsubseteq_{post} \Delta_{st} \wedge \neg delete(F^{++} \setminus F^{++}_{st}, F^{--} \setminus F^{--}_{st}, a) \implies compat_{\Delta_{st}}(a)$
Since $\Delta \sqsubseteq_{post} \Delta_{st}$, we have $F^X \subseteq F^X_{st}$ for $X \in \{+-, -+\}$, $F^{++} \subseteq F^{++}_{st} \cup F^{-+}_{st}$ and $F^{--} \subseteq F^{--}_{st} \cup F^{+-}_{st}$. This means that $(F^{-+} \cup F^{++}) \subseteq (F^{-+}_{st} \cup F^{++}_{st})$ and $(F^{+-} \cup F^{--}) \subseteq (F^{+-}_{st} \cup F^{--}_{st})$. Hence, for any rule tr of a, we have $tar\text{-}compat_{\Delta_{st}}(R \setminus r(K))$, $tar\text{-}compat_{\Delta_{st}}(K)$ and $tar\text{-}compat_{\Delta_{st}}(N_i)$ (for each NAC N_i). But this also means that $compat_{\Delta}(K)$ and $compat_{\Delta}(N_i)$ (for each NAC N_i). Since $\neg delete(F^{++} \setminus F^{++}_{st}, F^{--} \setminus F^{--}_{st}, a)$, then each element in $L \setminus l(K)$ is not typed by $F^{++} \setminus F^{++}_{st}$ or $F^{--} \setminus F^{--}_{st}$, and so, $src\text{-}compat_{\Delta_{st}}(L \setminus l(K))$, and therefore, $compat_{\Delta_{st}}(a)$ as required. □

A.7 Theorem 6.6: *Extended migration compatibility*

Proof. Given two configurations ρ_s and ρ_t, in a first step, Algorithm 2 selects adapters just like in Definition 4.17. Therefore, by Lemma 6.5, those adapters are compatible with Δ_{st}.

Then, the algorithm selects context fixers, which by Lemma 5.13, have a diff included in Δ_{st}. By Lemma 6.5, those adapters are compatible with Δ_{st}. Similarly, the algorithm selects completers, which by Lemma 5.10, have a diff included in Δ_{st}, and therefore, they are compatible with Δ_{st}. Finally, the algorithm also selects soft completers, which are then concatenated with a context fixer. By the properties of context fixers (Lemma 5.13), this yields adapters compatible with Δ_{st}. □