Pattern-based Rewriting through Abstraction

Paolo Bottoni[1]**, Esther Guerra**[2]**, Juan de Lara**[2]

[1]*Computer Science Department, "Sapienza" Università di Roma (Italy)*

[2]*Computer Science Department, Universidad Autónoma de Madrid (Spain)*

bottoni@di.uniroma1.it, Esther.Guerra@uam.es, Juan.deLara@uam.es

---

Abstract. Model-based development relies on models in different phases for different purposes, with modelling patterns being used to document and gather knowledge about good practices in specific domains, to analyse the quality of existing designs, and to guide the construction and refactoring of models. Providing a formal basis for the use of patterns would also support their integration with existing approaches to model transformation. To this end, we turn to the commonly used, in this context, machinery of graph transformations and provide an algebraic-categorical formalization of modelling patterns, which can express variability and required/forbidden application contexts. This allows the definition of transformation rules having patterns in left and right-hand sides, which can be used to express refactorings towards patterns, change the use of one pattern by a different one, or switch between pattern variants. A key element in our proposal is the use of operations to abstract models into patterns, so that they can be manipulated by pattern rules, thus leading to a rewriting mechanism for classes of graphs described by patterns and not just individual graphs. The proposal is illustrated with examples in object-oriented software design patterns and enterprise architecture patterns, but can be applied to any other domain where patterns are used for modelling.

Keywords Model-based development, Modelling patterns, Graphs, Trees, Category theory, Graph transformation, Refactorings, Pattern variants.

1. Introduction

Model-based approaches have become commonplace for software development [31, 35, 49], where models are actively used to specify, verify, test, and generate code for the applications to be built. In this setting, modelling patterns [1, 25] become essential assets, as they describe proven solutions to recurring problems within a domain. Therefore, means are needed to formally describe modelling patterns for specific domains, and to make them operational. Such a formal theory would enable their use for pattern-assisted modelling, reasoning about the consequences of their combination or possible conflicts.

Pattern-based modelling [9] enables modelling at a higher-level of abstraction, as patterns become higher-level modelling primitives. However, making patterns "first-class citizens" requires appropriate means for their manipulation, including ways to express refactoring toward patterns, formally describe how to change the use of one pattern by another one in a certain context, and generate pattern variants.

In previous works [8, 9] we proposed a formal notion of *pattern*, including the satisfaction of a pattern by a model, and the composition of patterns. In this paper we formalize and improve such notion, distinguishing between pattern trees and variable patterns. The latter are a natural improvement of the former with equations governing the allowed variability of the pattern variable regions. We define several kinds of pattern morphisms and define a category that admits pushouts [32], on which we base a pattern rewriting technique [21]. We characterize satisfiability, and provide a classification of patterns according to this property. Then, we define pattern rules, and an abstraction operation permitting the abstraction of a graph with respect to a pattern, and hence its rewriting via a pattern rule. This form of abstract rewriting opens the door to analysis of rewriting transformations for classes of initial graphs (as abstracted in a pattern) and not just for individual initial graphs, as usual in standard graph transformation [21].

We also formally study the recurring notion of pattern *variant*, where a pattern may be realized in several ways. For example, in the `Adapter` pattern, one can use inheritance or delegation [25], and operations of the adapter can be provided in the same class, or in multiple classes. The rewriting technique can be used to obtain such pattern variants, and to substitute one variant for another on concrete models.

Patterns can be derived for domain-specific modelling languages, and we illustrate our approach at work in the domains of enterprise application architecture [23] and object oriented software design patterns [25]. Adopting patterns as a way of specifying architectural styles and pattern rules as a way to describe architectural reconfigurations capitalises on a vast literature employing type graphs, graph constraints and transformation rules for specifying the construction of architectural configurations conforming to the constraints (see e.g. [4, 6]). In [48] a similar formalism is used to verify conformance of a realised architecture to a style, while graph transformations allow the architecture to evolve. Similar techniques can be used for model smell (or anti-pattern) detection and repair [33, 3].

Altogether, this paper makes the following contributions: (a) a formal notion of pattern, and the definition of pattern rules, (b) an abstraction operation for graphs into patterns, and compatibility conditions for their manipulation using graph and pattern rules, (c) some operations to construct variants of a given base pattern, and (d) application of the theory to enterprise application architecture and object oriented design. Due to the length of the underlying theory, in this paper we restrict to pattern trees, while the developments for variable patterns will be the subject of a subsequent contribution.

The rest of the paper is organized as follows. Section 2 provides a motivation on the different choices of our approach, and gives an overview of it. Section 3 introduces pattern trees admitting nested (positive and negative) variable regions. Section 4 discusses abstraction operations to represent graphs as patterns. Section 5 defines pushouts and pattern rules. Section 6 describes some mechanisms to obtain variants of a given pattern. Section 7 shows some examples in the area of enterprise application architecture and software design patterns. Section 8 compares with related research and Section 9 concludes the paper. An appendix provides the details of the proofs of the different results. The reader is assumed to have basic knowledge of category theory, as can be found for example in [32].

2.   Motivation and overview of the approach

We provide an overview of the paper, starting from a collection of requirements for a formalism for pattern modelling.

1. It should be easily integrated with existing ones for Model Driven Development (MDD).

2. It should allow the assessment of the relation between a model and a pattern.

3. It should support model refactoring to enforce conformance of a model to a pattern.

4. It should support model transformation so that a model conformant to a pattern is transformed to a model conformant to a different pattern.

Patterns are *restrictions* on the structure that models may assume and as such they can be expressed either as logical constraints using some textual notation based on First Order Logic (FOL), or through some graph-based formalism, or even through second order logic over graphs [16]. Considering Requirement 1, we adopt a graph-based formalism, since (attributed typed) graphs are a widespread formalism for the representation of models and meta-models (especially when the Unified Modeling Language (UML) is adopted) in MDD. In this approach, graph transformations are used as a declarative specification according to which one can construct models progressively closer to the level where concrete implementation is possible, or different views of semantically equivalent models. A number of options are then available to express that only graphs with specific "shapes" are admissible. Among them we mention Hierarchical Graphs [18, 12] and nested graph constraints [27, 20]. Both these approaches allow the expression of an overall structure as well as of local configurations within the structure. We leave to Section 8 a more detailed discussion on the specific choices which led us to adopt a particular restriction of the formalism of nested graph constraints. The requirement could have also been met by using the Object Constraint Language (OCL), which is an integral part of UML and which provides *navigation expressions* to refer to the graph-based representation of the model. However OCL expressions tend to be verbose and the definition of complex patterns becomes rapidly unwieldy, while a graphical representation can express them in a compact form. Moreover, OCL specifications do not lend themselves well to the kind of formal transformation needed to meet Requirements 3 and 4.

Considering Requirement 2, the fundamental relation that one wants to express is whether a model is *conformant* to a pattern. Conformance is more complex than direct pattern matching, as some suitably denoted parts of a pattern can occur repeatedly in the model. A typical example is that of the *Observer* pattern, where several instances of different types of concrete observer can be associated with the same *Subject*. In this case we want to see the complex formed by the subject, all of its observers, and all of the links between them as forming a single instantiation of the pattern, rather than several occurrences of it. Besides this, another typical activity one wants to perform is the discovery of underlying similar structures within different models. A notion of *abstraction* is therefore needed, by which the relevant part of a model can be seen as an instance of a general pattern. With the mentioned techniques based on graph transformation theory, specific algebraic or categorical constructions come to that effect.

Requirements 3 and 4 point to the need for a notion of *pattern-based model transformation*, whereby one can transform a model into another with reference to some pattern. By adopting the graph transformation approach, the notion of a pattern rule arises naturally, through which one can indicate modifications both in the overall (hierarchical or nested) structure of the pattern and in its local configurations.

Pattern rules can then be used to realise a form of *refactoring* - one transforms a model, abstracted as a source pattern, so as to make it conformant to a target pattern, then concretised back to a model – or to specify how a pattern can be transformed to a different one. In this second case, one can be interested in the transformation per se, for example to express pattern *variants*, or can apply the transformation so that all instances of the source pattern in a model are *migrated* to become instances of the target pattern.

In order to meet the requirements above, we proceed as follows.

We first introduce patterns as a collection of typed graphs structured by graph morphisms in a tree shape, with annotations indicating whether subtrees must be interpreted positively or negatively. This is equivalent to a form of nested graph constraints with a standard formula associated with the tree structure, rather than arbitrary operators along each branch, and it allows a subsequent simple definition of morphisms between patterns, leading to the identification of a category of patterns equipped with pushouts, and providing the setting for a number of original contributions.

We provide a formal notion of pattern satisfaction and discuss the conditions under which the semantics of a pattern is preserved when the pattern structure or the local configurations are extended.

We use pushout constructions as a basis for a number of pattern operators extending the classical operations of tree concatenation and graph glueing to patterns, as well as introducing an original notion of merging of a graph with a pattern. These operations are the building blocks for defining the abstraction of a graph into a pattern, preserving the graph as part of the semantics of the derived pattern.

The existence of pushouts allows the introduction of a notion of pattern rule, extending standard Double Pushout rules from graph transformation, so that a number of theoretical results developed under that approach can be used. This provides a uniform view of pattern-related transformations, including the refactoring of concrete models to conform to patterns, the migration of instances of a source pattern to instances of a target pattern, or the derivation of variants from a source pattern.

3. Trees and Pattern Trees

We revise the definition of *pattern* in [9] to make it more amenable to the definition of morphisms, but without losing generality. Patterns are modelled as trees, where each node of such a tree is associated with a graph and there is a corresponding graph morphism for each edge of the tree. The tree structure is used to express variability regions in graphs. Roughly speaking, such variability is given by the difference between the source and target graphs in each morphism.

A *tree* is defined by a structure $T = (V, E, root)$, where V is a finite set of nodes, $E \subset V \times V$ is a set of edges s.t. $(v_i, v_j) \in E \implies ((v_j \neq v_i) \land (v_j, v_i) \notin E \land \nexists v_k [v_k \neq v_i \land (v_k, v_j) \in E])$, and $\exists! root \in V$ s.t. $(v, root) \notin E$ for any $v \in V$. Moreover, if we define the sets of edges in the path from $root$ to a node v as the fixpoint of the function $p(v)$ defined by Equation (1), we have $v \in V \land v \neq root \implies \exists! v_l[(root, v_l) \in p(v)]$. As a consequence of the above properties, for each node $v \in V$ there is one and only one path (with no cycle in it) from the root to v.

$$p(v) = \begin{cases} \emptyset & \text{if } v = root, \\ p(v') \cup \{(v', v)\} & \text{if } (v', v) \in E. \end{cases} \tag{1}$$

Given a node $n \in V$, its children are defined by the set $children(n) = \{m \mid (n, m) \in E\}$.

Given two trees T_1 and T_2, a *tree morphism* $m \colon T_1 \to T_2$ is defined by a pair of total *injective* set functions $m = (m_V \colon V_1 \to V_2, m_E \colon E_1 \to E_2)$, mapping the roots $(m_V(root_1) = m_V(root_2))$ and

preserving the tree structure: $e = (n_1, n_2) \in E_1 \implies m_E(e) = (m_V(n_1), m_V(n_2))$.

As usual for trees, a tree T induces two functions, *Subtree* and *Sons*, where for each $n \in V$, *Subtree*$(n) = (V' \subseteq V, E' \subseteq E, n \in V')$ is the subtree of T rooted in n and *Sons*$(n) = \{Subtree(m) \mid m \in children(n)\}$ is the set of subtrees of each child of n. Moreover, T induces a collection of paths, as described above, from which we derive $Path(T) = \{path(v) \mid v \in V\}$ where each set $path(v) = \{v\} \cup \{v_i \in V \mid (v_i, v_j) \in p(v)\}$ contains all and only the nodes appearing in the sequence of edges from the *root* to $v \in V$, including v. $Cod(f)$ denotes the codomain of function f.

3.1. Pattern trees

Given the basic concepts regarding trees above, we are ready to introduce our notion of pattern trees.

Definition 1. (Pattern Tree)
A *pattern tree* is a construct $PT = (V, E, root, G, Mor, \sigma, \gamma_V, \gamma_E)$ where:

- $T = (V, E, root)$ is a tree rooted in $root$.

- G is a finite set of graphs and $\gamma_V \colon V \to G$ is a surjective function associating each node $v_i \in V$ with a graph in G.

- $Mor = \{m_{ij} \colon \gamma_V(v_i) \to \gamma_V(v_j) \mid (v_i, v_j) \in E\}$ is a finite set of total injective graph morphisms. We associate each $(v_i, v_j) \in E$ with a morphism m_{ij} through the surjective function $\gamma_E \colon E \to Mor$.

- $\sigma \colon Cod(Subtree) \to \{\oplus, \ominus\}$, with $\sigma(Subtree(root)) = \oplus$, assigns positive ($\oplus$) or negative ($\ominus$) *sign* to each subtree of T.

We call **PT** the set of all pattern trees.

Remark 3.1. While definition 1 is agnostic with respect to the kind of graphs used, in this paper we use attributed, typed graphs [21]. In particular, we assume the presence of a set, $TYPES$, of types that nodes and edges (collectively referred to as *elements*) may take through a function *type*. In practice, such types are given by a meta-model. If untyped graphs are used instead, one still has two types, one for nodes and another one for edges. We use the terms "graph" and "model" (in the sense of software engineering model) interchangeably.

Remark 3.2. The graphs in G correspond to the *variable regions* in [9], indicating the possibility of multiple instances of additional elements. Roughly, such variability is given by the difference between the source and target graphs in each morphism in Mor. According to the sign assigned by σ, we talk of *positive* or *negative* regions (or graphs). The fact that we allow nested negative regions of arbitrary length presents an improvement with respect to [9] (where no nesting of negative regions was allowed). Abusing of notation, we sometimes use the function σ on nodes of V instead of on subtrees.

Example 3.1. Figure 1(a) shows a pattern tree (named C/S) representing a simple client/server architecture in the form of a tree of graph morphisms, with a direct correspondence to the formal definition. The root graph is made of a Server node. A nested positive variable region clients models the fact that

there should be at least one client node, where the nested positive region `channels` requires that each client has at least one connection to the server. No connection is allowed between clients, as specified by the negative regions `inc` and `out`. The sign of the subtree rooted in a node is shown in the left-top corner of each box. To represent the graphs, we use the UML notation for object diagrams. Hence, nodes are mandatorily decorated with a type from the set {Server, Client} and optionally with an identifier (e.g., s, c). Graph morphisms are implicitly shown by equality of object identifiers. Figure 1(b) depicts the same pattern using a more compact notation visualizing nested regions as inclusions of graph differences. The area representing the `root` region is not explicitly marked, but would include all elements. Negative regions are decorated using the "forbidden" symbol.
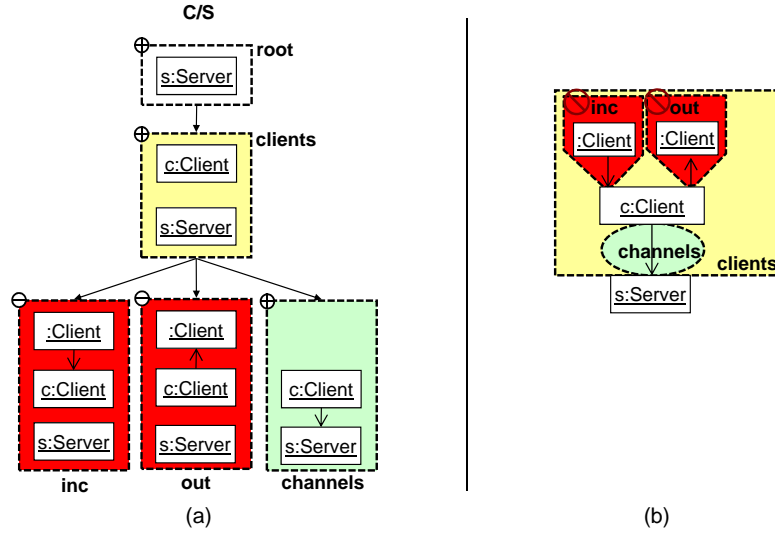

Figure 1. The C/S pattern tree. (a) in theoretical notation, (b) in compact notation.

Next, we introduce some operations that will be useful later. The first one (`mpath`) returns the chain of morphisms from the graph $\gamma_V(root)$ associated with the *root* of the tree to the graph $\gamma_V(v)$ associated with a given tree node v. The second one (`terminal`) returns the set of nodes of positive sign that do not have any children with positive sign.

Definition 2. (Morphism chain path)
Let PT be a pattern tree and $v \in V$ a node. The operation $mpath(PT, v) = \gamma_V(root) \rightarrow \ldots \rightarrow \gamma_V(v)$ returns the chain of graph morphisms from $\gamma_V(root)$ to $\gamma_V(v)$.

Definition 3. (Positive terminal nodes)
Given a pattern tree PT, we define its set of positive terminal nodes as $terminal(PT) = \{v \in V \mid \nexists\, v' \in children(v)\ with\ \sigma(v') = \oplus\ \wedge\ \forall\, v_i \in path(v)[\sigma(v_i) = \oplus]\}$.

Next, we define the satisfaction of a pattern tree by a graph. Following [9], we consider an existential semantics. Intuitively, a graph satisfies a pattern tree PT if it presents an occurrence of the *root*, at least one occurrence of a configuration of graphs in the subtrees associated with positive (up to terminal) nodes, and no occurrence of a configuration of the graphs in the subtrees associated with negative nodes,

where configurations are defined according to the morphisms in Mor. More precisely, Definition 4 gives a notion of satisfaction for pattern trees, where all graph morphisms are required to be injective.

Definition 4. (Pattern Tree Satisfaction)
Given a pattern tree $PT \in \mathbf{PT}$ and a graph M, M *satisfies* PT (or M is a *model* of PT), written $M \models PT$, if

$$\exists m \colon \gamma_V(root) \to M \, [\bigwedge_{v_i \in children(root)} \mathtt{SAT_{PT}}(PT, root, v_i, m, \gamma_E((root, v_i)))]$$

where the predicate $\mathtt{SAT_{PT}}$ is defined as follows:

$\mathtt{SAT_{PT}}(PT, r, v, m \colon \gamma_V(r) \to M, n \colon \gamma_V(r) \to \gamma_V(v))$

1. Let $T_v = \{p \colon \gamma_V(v) \to M \mid m = p \circ n\}$ be the set of all injective graph morphisms from $\gamma_V(v) \in V$ into M, commuting with the given morphisms m and n (see Figure 2).

2. if $children(v) = \emptyset$:

   - Let $S_v = T_v$.

3. else:

   - Let $S_v = \{p \in T_v \mid \bigwedge_{v_i \in children(v)} \mathtt{SAT_{PT}}(PT, v, v_i, p \colon \gamma_V(v) \to M, \gamma_E((v, v_i)))\} \subseteq T_v$.

4. if $\sigma(v) = \oplus$ return $|S_v| \geq 1$; else return $|S_v| = 0$.

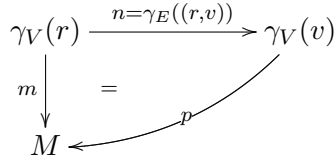$$\gamma_V(r) \xrightarrow{\ n = \gamma_E((r,v))\ } \gamma_V(v)$$

Figure 2.　Morphism compatibility for set S.

Remark 3.3. The codomains of the morphisms in T_v need not be disjoint (i.e., non-overlapping), but each morphism in T_v should be different.

We use the set of morphisms:

$$S_{PT,M} = \bigcup_{v_i \in V \wedge \sigma(v_i) = \oplus} S_{v_i} \cup \bigcup_{v_i \in V \wedge \sigma(v_i) = \ominus} T_{v_i} \tag{2}$$

to denote the union of all sets S_v (for nodes of positive sign) and T_v (for nodes of negative sign) used by $\mathtt{SAT_{PT}}$ in checking the satisfaction of $M \models PT$. The set $S_{PT,M}$ is useful because, for a graph M satisfying a pattern PT, $S_{PT,M}$ identifies a *maximal occurrence* of the pattern tree PT in M. Such a set contains all the morphisms from the positive graphs $\gamma_V(v) \in V$ to M that satisfy $\mathtt{SAT_{PT}}$ at each stage

of the recursion. For a node v with negative sign, we use T_v instead of S_v, as S_v would be empty if M satisfies the pattern, even if v has a nested negative region. In such case, we are interested in gathering all morphisms, as nested negative regions have universal semantics.

Example 3.2. Figure 3(a) shows an example of pattern tree satisfaction, where the graph M satisfies the pattern C/S in Figure 1. There is only one occurrence of the graph $\gamma_V(\texttt{root})$, given by morphism m_1 (morphism subindices denote the step in the recursion call to $\text{SAT}_{\textbf{PT}}$). Hence, the function $\text{SAT}_{\textbf{PT}}$ is invoked only once, with $\texttt{root}$, $\texttt{clients}$, m_1 and n_1 since the $\texttt{root}$ node has only one child. At this stage, the set $T_{\texttt{clients}}$ contains three morphisms: $m_{1,1}$, $m_{1,2}$ and $m_{1,3}$. The first identifies clients c and $c1$, the second identifies c and $c2$, while the third identifies c and $c3$. A new recursion call is performed with $\texttt{clients}$, $\texttt{channels}$, $m_{1,1}$ and $n_{2,1}$ as parameters. At this stage, the set $T_{channels}$ contains one morphism $m_{1,1,1}$ identifying the unique edge between the server s and $c1$. As node $\texttt{channels}$ does not have further children, the function returns $true$ in line 4 of the previous definition. Then, the function is evaluated with the other two children of $\texttt{clients}$ ($\texttt{inc}$ and $\texttt{out}$), which have negative sign. In both cases, sets $T_{\texttt{inc}}$ and $T_{\texttt{out}}$ contain no morphisms, since the client $c1$ is not connected to other clients, hence both invocations return $true$. The function also returns *true* when evaluated with morphism $m_{1,2}$. In this case there are two occurrences of $\gamma_V(\texttt{channels})$ (morphisms $m_{1,2,1}$ and $m_{1,2,2}$), but the function returns *false* when invoked with $m_{1,3}$. Altogether, at the second stage, the size of the set $|S_{\texttt{clients}}|$ is 2 (as two morphisms in $T_{\texttt{clients}}$ satisfy the pattern). Hence, the overall result is $true$. $S_{PT,M} = \{m_1, m_{1,1}, m_{1,2}, m_{1,1,1}, m_{1,2,1}, m_{1,2,2}\}$ identifies the maximal occurrence of the pattern in M, and is shown enclosed in a dotted area in the graph M.
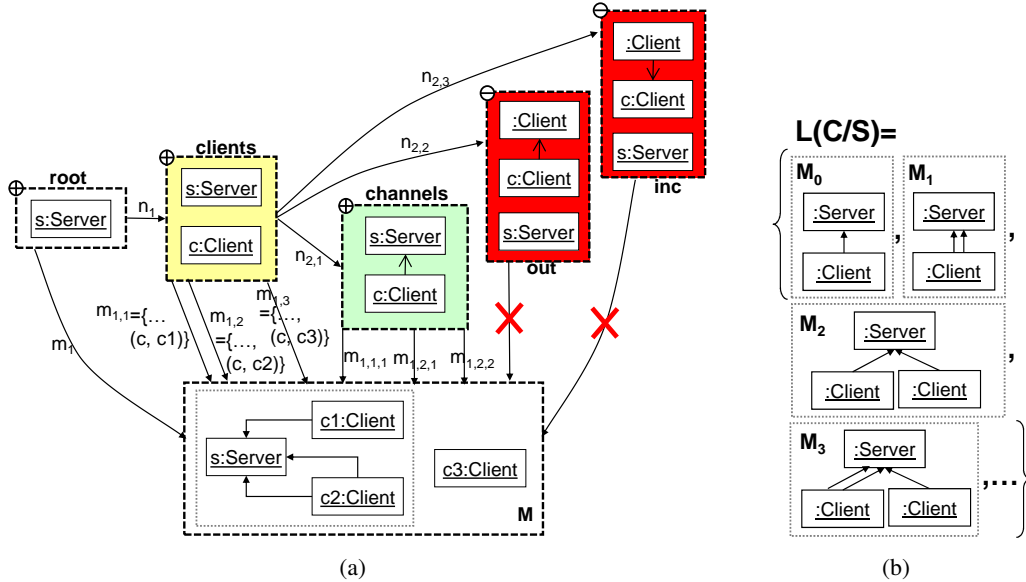

Figure 3.   Pattern tree satisfaction example (a), and (b) some elements in $L($C/S$)$.

We define the semantics of a pattern tree as the set of all graphs that are satisfied by it. We are also interested in characterizing the set of those models that do not contain "extra" elements but only

those necessary for satisfaction, as defined by the notion of *joint surjectivity*: A set of graph functions $\{f_i \colon X_i \to M\}$ with same codomain M is jointly surjective if each node and edge of M is the image of a node or edge in some X_i according to f_i.

Definition 5. (Pattern Tree Semantics)

Given a pattern tree $PT \in \mathbf{PT}$, its *semantics* $\llbracket PT \rrbracket = \{M \mid M \models PT\}$ is given by the (possibly infinite) set of graphs that satisfy it. The language of models spawned by PT is $L(PT) = \{M \in \llbracket PT \rrbracket \mid S_{PT,M}$ is jointly surjective$\}$. PT is called *unsatisfiable* if $L(PT) = \emptyset$, otherwise it is called *satisfiable*.

The main point of the pattern tree semantics is that, in each graph $M \in \llbracket PT \rrbracket$ there is an "occurrence" $M' \hookrightarrow M$ of the pattern. Such occurrence is one graph $M' \in L(PT)$.

Theorem 3.1. (Occurrence embedding)

Given a pattern tree PT, for each finite graph M, we have: $M \in \llbracket PT \rrbracket \implies \exists M' \in L(PT)[M' \hookrightarrow M]$.

Proof:

(Sketch) The set $L(PT)$ contains all minimal graphs satisfying PT. Assume there is some $M \in \llbracket PT \rrbracket$ for which $\nexists M' \in L(PT)$ such that $M' \hookrightarrow M$; this means that M does not contain an occurrence of the pattern PT, and hence does not satisfy it. Therefore such M cannot belong to $\llbracket PT \rrbracket$. □

Example 3.3. Figure 3(b) shows some elements in the set $L(\texttt{C/S})$, of graphs presenting at least one server and one connected client (as required by regions $\texttt{root}$, $\texttt{clients}$ and $\texttt{channels}$). The variability region named $\texttt{clients}$ permits more than one client, each one of them connected with at least one arrow to the server. Note that for the graph M in Figure 3(a), $M \notin L(\texttt{C/S})$, but there is an inclusion $M_3 \hookrightarrow M$.

A pattern may be unsatisfiable if the structure required by positive regions is forbidden by negative ones. In Section 3.2, we will define a certain class of morphisms between satisfiable patterns. In general, satisfiability of pattern trees is undecidable, but it is decidable for a certain class of pattern trees, with limited nesting of negative regions. Definition 6 introduces a class of trees with limited negative nesting.

Definition 6. (n-negation pattern trees)

Given a natural number $n \in \mathbb{N}_0$, a pattern tree PT is called an n-*negation pattern tree iff* $p \in Path(T) \implies (|\{v \mid v \in p \land \sigma(v) = \ominus\}| \le n)$.

Example 3.4. The $\texttt{C/S}$ pattern in Figure 1 is a 1-negation pattern tree, because it has at most one graph with negative sign in each branch.

Intuitively, if PT has at most one negative region in each path, i.e., it is a 1-negation pattern tree, satisfiability can be decided by building a graph made of the glueing of all positive regions, and then checking whether some negative region occurs.

Theorem 3.2. (Satisfaction of 1-negation pattern trees)

Satisfaction is decidable for 1-negation pattern trees.

Proof:

See Appendix 9. □

Example 3.5. To check satisfiability of pattern `C/S` in Figure 1, we build the colimit of the maximal subtree of `C/S` rooted in `root` and containing positive regions only (more details of this construction in Appendix 9), leading to graph M_0 in Figure 3(b). As $M_0 \models$ `C/S`, the pattern is satisfiable. One can also easily note that M_0 is the minimal graph satisfying `C/S`.

The source of difficulty in checking satisfaction of n-negation pattern trees with $n > 1$ lies with the presence of nested negative regions (which we also call *productive regions*), which due to negation, have a universal semantics. In particular, the problem arises from those nested regions demanding a certain extra structure to form a *match* (and occurrence) for another nested negative region. The morphisms defining nested regions are similar to those defining (non-deleting) graph transformation rules [21] and satisfiability becomes similar to termination of graph transformations. We characterize nested negative regions and their dependencies in Definition 7. A dependency arises if a region (v_i, v_j) contains in $\gamma_V(v_j)$ some element x not mapped from $\gamma_V(v_i)$, and there is some element m in the graph $\gamma_V(v_k)$ of some other nested region (v_k, v_l) with the same type as x.

Definition 7. (Negative nested region and dependency)
Given a pattern tree PT, $(v_i, v_j) \in E$ is called a *nested negative region*, if $\sigma(v_i) = \sigma(v_j) = \ominus$. Given two nested negative regions $(v_i, v_j), (v_k, v_l) \in E$ we say that (v_k, v_l) *depends on* (v_i, v_j), written $(v_k, v_l) \rightsquigarrow (v_i, v_j)$ if $\exists t \in TYPES[\exists x \in \gamma_V(v_j)[type(x) = t \wedge \nexists x' \in \gamma_V(v_i)[\gamma_E((v_i, v_j))(x') = x]] \wedge \exists m \in \gamma_V(v_k)[type(m) = t]]$.

Remark 3.4. Regions $(v_i, v_j), (v_k, v_l) \in E$ need not be different. A region (v_i, v_j) depends on itself if it presents an element of type t in $\gamma_V(v_i)$ and an additional element of the same type in $\gamma_V(v_j)$.

Theorem 3.3. (Satisfaction of 2-negation pattern trees without dependencies)
Satisfaction is decidable for 2-negation pattern trees without negative region dependencies.

Proof:
See Appendix 9. □

Example 3.6. In Figure 4(a) the `C/S` pattern of Figure 1 is modified to contain two productive regions. For the nested region (`connS`,`connCli`) each client connected to a server must be connected to another client. The nested region (`cli`,`conn`) demands that each client be connected to the server. Hence, the first region demands some structure (another client), which could be part of a match for the second region, which demands additional structure in its turn (a connection to the server). Therefore, there is a dependency (`cli`,`conn`) $\rightsquigarrow$ (`connS`,`connCli`), and another dependency (`connS`,`connCli`) $\rightsquigarrow$ (`cli`,`conn`) because (`cli`,`conn`) demands an arrow between each client and server, which could be part of a match for γ_V(`connS`). Finally, there is a dependency of (`connS`, `connCli`) with itself, because the nested region demands an extra client, which can be part of a match for `connS`. This situation is analogous to those considered in termination criteria based on critical pair analysis for non-deleting graph transformation rules with NACs [19]. In this case, the pattern is finitely satisfiable (e.g. by the graph shown in Figure 4(b)), but we show in Appendix 9 that there are patterns (with negative regions dependencies) that are only satisfied by infinite graphs.
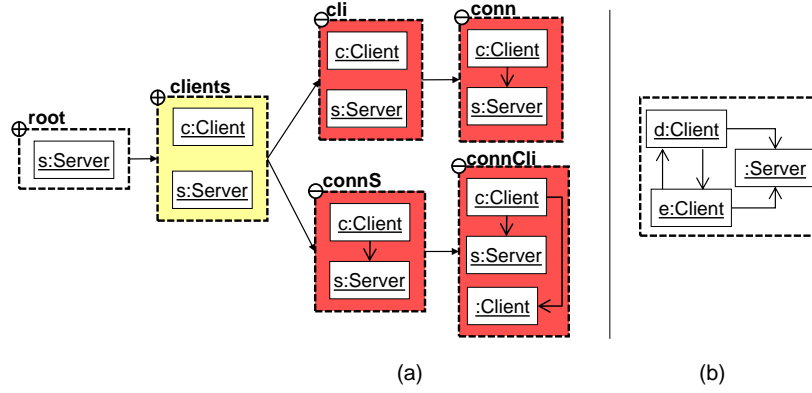
Figure 4. A pattern with nested negative region dependencies (a); a graph that satisfies it (b).

3.2. Pattern Tree Morphisms

We next define two kinds of morphism between pattern trees. Morphisms of the first kind are defined structurally only, identifying regions of source and target pattern trees, while those of the second type are more restrictive, as they imply a relation between the semantics of their source and target patterns.

Definition 8. (Pattern Tree Morphism)
Given two pattern trees PT_1 and PT_2, a *pattern tree morphism* (or *PT-morphism* for short) $m \colon PT_1 \to PT_2$ is given by the tuple $m = (m_V \colon V_1 \to V_2, m_E \colon E_1 \to E_2, m_F)$, where:

- (m_V, m_E) is a tree morphism that preserves the signs ($\sigma^2 \circ m_V = \sigma^1$)

- m_F is a family of graph morphisms containing a morphism $m_G^i \colon H_i^1 \to H_i^2$ from each graph $H_i^1 = \gamma_V^1(v_i^1) \in G_1$ into some graph $H_i^2 = \gamma_V^2(m_V(v_i^1)) \in G_2$ and preserving the morphism structure, as the left of Figure 5 shows.

Remark 3.5. Making abuse of notation, but for simplicity, in Figure 5 we have depicted $e_1 = (v_i^1, v_j^1) \in E_1$ as an arrow $E_1 \colon v_i^1 \to v_j^1$, and similarly for $e_2 = (v_i^2, v_j^2) \in E_2$. Note also that $e_2 = m_E(e_1)$ (not indicated in the diagram) and $m_{ij}^1 = \gamma_E^1(e_1)$, $m_{ij}^2 = \gamma_E^2(e_2)$.
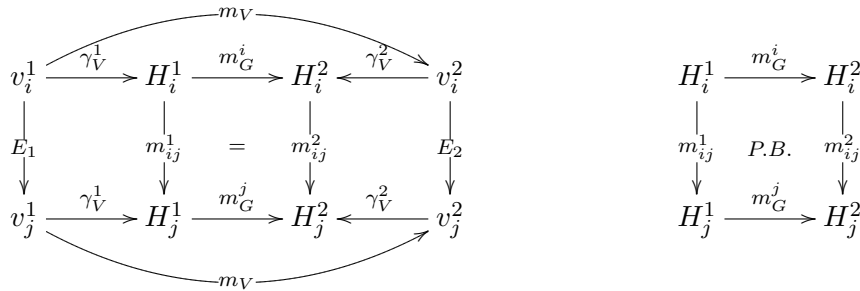

Figure 5. Compatibility conditions for PT-morphisms (left) and for SPT-morphism semantics (right)

Definition 9. (Semantic PT-morphism)
Given two satisfiable pattern trees PT_1 and PT_2, a *semantic pattern tree morphism* (or *SPT-morphism* for short) is a PT-morphism $m = (m_V, m_E, m_F = \{m_G^i\})$ satisfying the following two conditions:

1. All negative regions of PT_2 are the image of some negative region of PT_1, i.e. $\forall v_2 \in V_2[\sigma(v_2) = \ominus \implies \exists v_1 \in V_1[m_V(v_1) = v_2]]$.

2. The variability in regions is preserved: all commuting squares $m_G^j \circ m_{ij}^1 = m_{ij}^2 \circ m_G^i$ induced by the underlying PT-morphism are pullbacks (see the right of Figure 5).

Remark 3.6. Note that the direction of the graph morphisms in m_F is from PT_1 to PT_2, even in case of regions with negative sign. As the semantics of patterns and morphisms in Theorem 3.4 will clarify, this is needed to ensure that for each graph satisfying the source pattern there is a morphism to a graph satisfying the second pattern. For the same reason we require that PT_2 does not contain negative regions that are not mapped from any region of PT_1.

Example 3.7. Figure 6(a) shows an example of SPT-morphism. Indeed, the 3T pattern does not add unmapped negative regions, and each variability region in C/S is preserved in 3T. That is, the client c in C/S and the edge to s are variable in C/S (because these are not mapped from the morphism from the *root*), and are also variable in 3T. This is so as square (1) is not only commuting, but also a pullback.
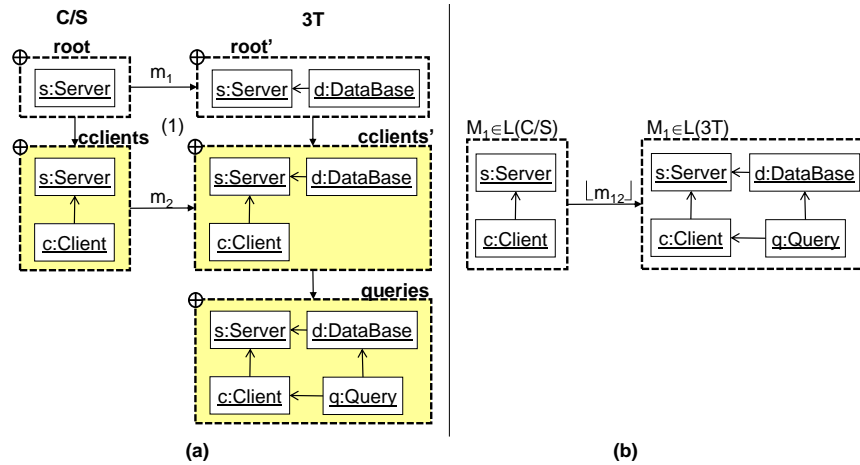

Figure 6. An SPT-morphism (a), and an element of its semantics (b).

Figure 7(a) shows a PT-morphism, which fails to be an SPT morphism due to the second condition in Definition 9: some variable elements in C/S (the client c and the edge) are not variable in 3T, and hence square (1), although commuting, is not a pullback (the root in C/S would need to be equal to cclients for (1) to be a pullback).

Figure 8(a) shows a PT-morphism, which fails to be an SPT morphism due to the first condition in Definition 9: the target pattern tree 3T has a negative region (nodup) which is not the image of any region in the source pattern.
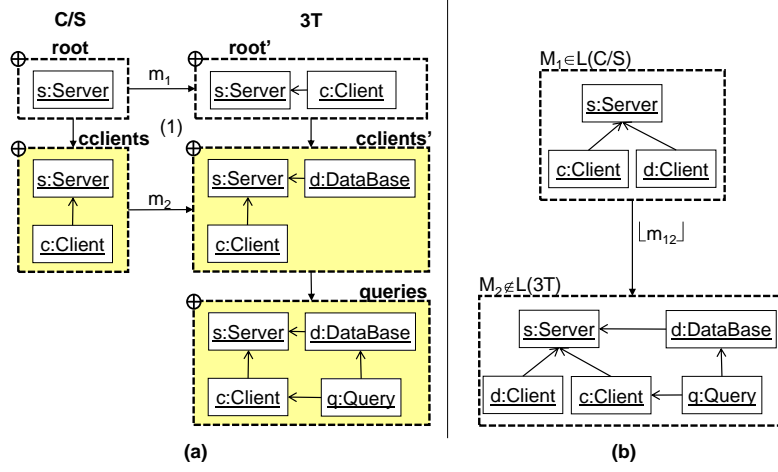
Figure 7.   A PT-morphism that is not an SPT-morphism (a), failure of SPT semantics (b).

The semantic properties of SPT-morphisms are summarized by Theorem 3.4. The main idea is that an SPT morphism m between two pattern trees PT_1 and PT_2 is "equivalent" to a set of morphisms, containing morphisms $\lfloor m_{ij} \rfloor$ between each graph $M_1^i \in L(PT_1)$ and some graph $M_2^j \in L(PT_2)$. Moreover, such $\lfloor m_{ij} \rfloor$ morphisms can be constructed from the S_{PT_1, M_1^i} and S_{PT_2, M_2^i} sets used in the satisfaction checking of $M_1^i \models PT_1$ and $M_2^i \models PT_2$.

Theorem 3.4. (SPT-Morphism Semantics)
Given an SPT-morphism $m \colon PT_1 \to PT_2$, we have that for each $M_1 \in L(PT_1)$ there exist $M_2 \in L(PT_2)$ and $\lfloor m_{12} \rfloor \colon M_1 \to M_2$ such that the diagram of Figure 9 commutes, where D_i (for $i = 1, 2$) is the diagram made of the graphs in G_i and the morphisms in Mor_i; S_{PT_i, M_i} is the set of morphisms used in the satisfaction checking of M_i and m_F is the set of graph morphisms in m (a set of morphisms is indicated with a dotted double arrow).

Proof:
(Sketch) See Appendix 9.   □

The semantics of m is given by all such morphisms: $[\![m]\!] = \{ \lfloor m_{12} \rfloor \colon M_1 \to M_2 \mid M_1 \in L(PT_1) \land M_2 \in L(PT_2) \}$.

Example 3.8. Figure 6(b) shows an example of how an SPT-morphism $m \colon \texttt{C/S} \to \texttt{3T}$ induces a graph morphism $\lfloor m_{12} \rfloor \colon M_1 \to M_2$ between $M_1 \in L(\texttt{C/S})$ and $M_2 \in L(\texttt{3T})$. Actually, there are morphisms from M_1 into every $M' \in L(\texttt{3T})$ with at least one occurrence of $\gamma_V(\texttt{root}')$ and $\gamma_V(\texttt{cclients}')$ and an arbitrary number of occurrences of $\gamma_V(\texttt{query})$.

The PT-morphism $m \colon \texttt{C/S} \to \texttt{3T}$ of Figure 7(a) is not an SPT-morphism, and in this case there exists an $M_1 \in L(\texttt{C/S})$ for which there is no $M_2 \in L(\texttt{3T})$ with a corresponding graph morphism $\lfloor m_{12} \rfloor \colon M_1 \to M_2$, as illustrated in Figure 7(b). This is due to the fact that the variability of $\texttt{C/S}$ is not preserved in $\texttt{3T}$, as any graph with some client not receiving a query does not belong to $L(\texttt{3T})$ (even though it might belong to $[\![\texttt{3T}]\!]$).
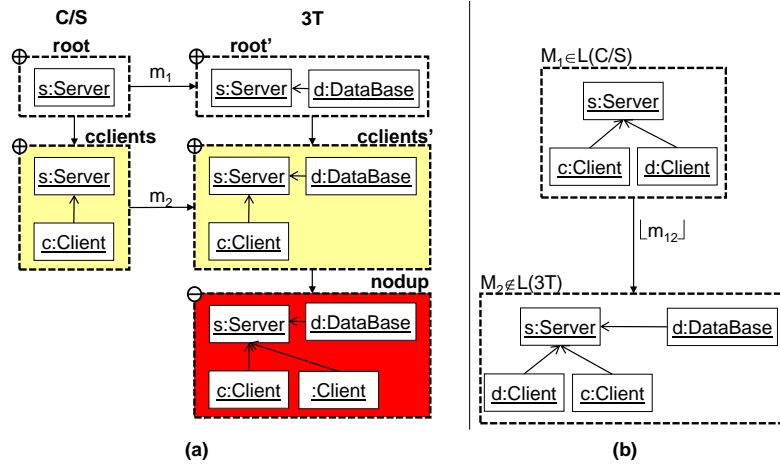
Figure 8.   A PT-morphism that is not an SPT-morphism (a), failure of SPT semantics (b).

$$
\begin{array}{ccc}
D_1 & \overset{m_F}{\dashrightarrow} & D_2 \\
\vdots & & \vdots \\
S_{PT_1, M_1} & = & S_{PT_2, M_2} \\
\vee & & \vee \\
M_1 & \xrightarrow[\lfloor m_{12} \rfloor]{} & M_2
\end{array}
$$

Figure 9.   Compatibility conditions for SPT-morphism semantics.

The PT-morphism m of Figure 8(a) is not an SPT-morphism (as in this case 3T adds an unmapped negative region). Figure 8(b) shows that there is some $M_1 \in L(\text{C/S})$ (in particular those having two or more clients connected to a server) for which there is no $M_2 \in L(\text{3T})$ with a corresponding graph morphism $\lfloor m_{12} \rfloor \colon M_1 \to M_2$, as such a M_2 would not satisfy the unmapped negative region.

Example 3.9. Figure 10 shows an example of the construction of $\lfloor m_{12} \rfloor$ from m, $S_{\text{C/S}, M_1}$ and S_{3T, M_2} (see Appendix 9 for details on the construction). The left shows an SPT-morphism, together with two graphs $M_1 \in L(\text{C/S})$ and $M_2 \in L(\text{3T})$. The right shows a pattern making it explicit that M_1 contains two occurrences of cclient, so at least two are needed in M_2. Morphism $\lfloor m_{12} \rfloor$ is then constructed using the morphism m and the replicated morphisms m_2 and m_2' by a colimit construction.

Note that the converse of Theorem 3.4 does not hold in general. Indeed, given an SPT-morphism $m \colon PT_1 \to PT_2$ and a graph $M_2 \in L(PT_2)$, there might not exist a graph $M_1 \in L(PT_1)$ and a morphism $\lfloor m_{12} \rfloor \colon M_1 \to M_2$. The reason is that PT_2 may have weaker negative regions (i.e., "bigger" graphs), and hence $L(PT_2)$ may contain graphs that are not admissible by $L(PT_1)$. However, a weaker version of this result holds, if we demand $M_2 \in [\![PT_1]\!]$, because then it needs to satisfy all negative regions in PT_1. We will use this result in Section 5, when proving the correctness of pattern-based rewriting with respect to graph abstraction.
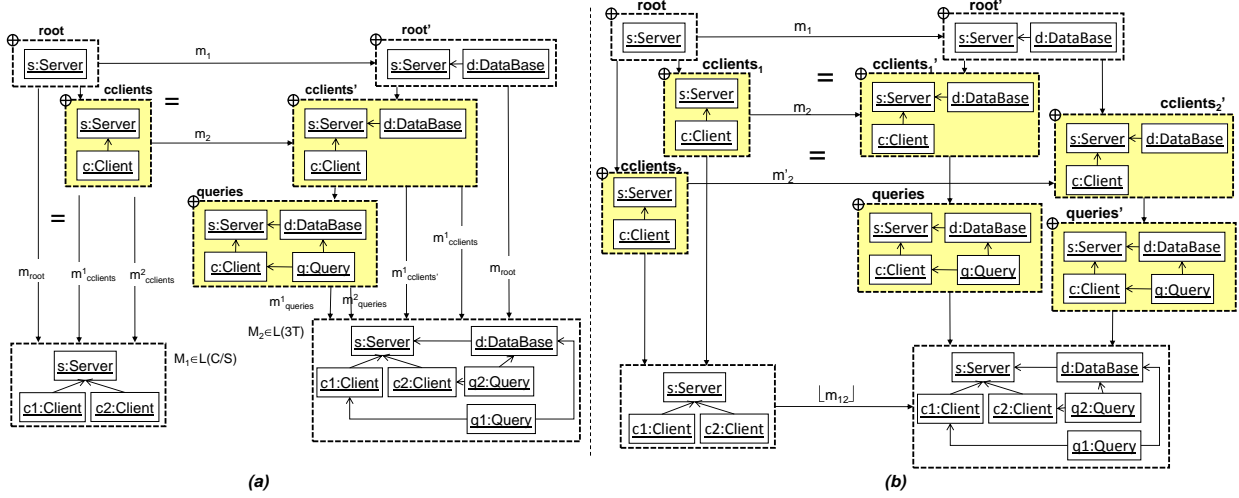
Figure 10.   Semantics of SPT-morphisms: Constructing $\lfloor m_{12} \rfloor$ from m, $S_{\text{C/S},M_1}$ and S_{3T,M_2}.

Lemma 3.1. (Source induced morphism)

Given an SPT-morphism $m\colon PT_1 \to PT_2$ and a graph M_2 s.t. $M_2 \in L(PT_2)$ and $M_2 \in \llbracket PT_1 \rrbracket$, there exist a graph $M_1 \in L(PT_1)$ and a morphism $\lfloor m_{12} \rfloor \colon M_1 \to M_2$.

Proof:

(Sketch) We take M_1 to be the subgraph of M_2 for which the set of morphisms S_{PT_1,M_2} is jointly surjective, and $\lfloor m_{12} \rfloor \colon M_1 \to M_2$ to be the inclusion. M_1 exists due to Theorem 3.1. Note that, given a set S_{PT_1,M_2}, the graph M_1 is unique, but if M_2 satisfies the pattern PT_1 at several occurrences (i.e., more occurrences of the root exist), then we have different sets $S^i_{PT_1,M_2}$ making the pattern satisfied.   □

Definition 10. (Isomorphic Pattern (sub)Trees)

Two pattern (sub)trees PT_1 and PT_2 are isomorphic, written $PT_1 \sim PT_2$ if there exist two injective PT-morphisms $m\colon PT_1 \to PT_2$ and $m'\colon PT_2 \to PT_1$ with $m'(m(PT_1)) = PT_1$ and $m(m'(PT_2)) = PT_2$.

Note that if two inverse morphisms $m\colon PT_1 \to PT_2$ and $m'\colon PT_2 \to PT_1$ exist, then they should be SPT-morphisms. This is so because every region (positive and negative) of both patterns should be mapped, and moreover, variability has to be preserved to have the graph isomorphisms. Therefore, as a corollary from Theorem 3.4, if $PT_1 \sim PT_2$, then $L(PT_1) = L(PT_2)$.

Theorem 3.5. (Pattern Tree Categories)

The collection of pattern trees and their PT-morphisms form the category $\mathbf{PattTree} = (\mathbf{PT}, Hom(\mathbf{PT}))$. The collection of satisfiable pattern trees and their SPT-morphisms form the category $\mathbf{SPattTree}$, which is a subcategory of $\mathbf{PattTree}$.

Proof:

See Appendix 9.   □

4. Abstracting Graphs into Patterns

We introduce an abstraction operation for graphs allowing their manipulation by pattern rules (made of pattern trees in left and right-hand sides). By abstracting, one takes a graph M satisfying a pattern PT ($M \in [\![PT]\!]$) and creates a pattern PT' with similar structure to PT, in the sense that it should allow an SPT-morphism from PT to the abstracted pattern PT', but ensuring that M becomes a member of $L(PT')$. Once such an abstraction is performed, the graph M can be manipulated using a pattern rule.

   While there are many strategies to abstract a graph into a pattern, here we provide a simple one, restricted to pattern trees without nested negative regions. We leave a general abstraction procedure to future work, as the current abstraction strategy is enough for many practical cases, as Section 7 shows.

   The main idea of the strategy is to first choose a path p of the pattern tree, whose graphs will be "enlarged" (in bottom-up order) to account for the graph M, and the graphs in other paths are adjusted to reflect the modification of parent graphs (i.e., a top-down traversal is done).

   Before defining abstraction, we first provide some auxiliary operations. First, we define some constructors of pattern trees given a set of graph morphisms with tree structure and a single graph. We say that a set of morphisms $Mo = \{m_j\}_{j \in J}$ has *tree structure* if the graph formed by taking the morphisms in Mo as edges, and the set of graphs in $Dom(m_j) \cup Cod(m_j)$ (with $m_j \in Mo$) as nodes is a tree.

Definition 11. (Pattern tree constructors)
Given a set of graph morphisms $Mo = \{m_j\}_{j \in J}$ with tree structure, the operation $tree(Mo)$ produces a pattern tree as follows: $tree(Mo) = \langle V, E, root, V, Mo, \sigma, \gamma_V, \gamma_E \rangle$ with:

- $V = \{Dom(m_j) \cup Cod(m_j) \mid m_j \in Mo\}$,

- $E = \{(G_i, G_j) \mid m_{ij} \colon G_i \to G_j \in Mo\}$,

- $root = G_i$ s.t. $\nexists m_{ji} \colon G_j \to G_i \in Mo$,

- σ yields always $\oplus$,

- $\gamma_V = id_V$,

- $\gamma_E = \{((G_i, G_j), m_{ij}) \mid (G_i, G_j) \in E\}$.

   Given a graph M, $tree(M, sign)$, with $sign \in \{\oplus, \ominus\}$, produces the pattern tree $\langle V = \{M\}, E = \emptyset, root = M, G = V, Mor = \emptyset, \sigma = \{(M, sign)\}, \gamma_V = id_V, \gamma_E = \emptyset \rangle$.

   Next, we define some operations to construct a pattern tree out of existing ones. The first operation (*concat*) appends the second tree to the first one, at a given place. The second (*fuse*) works on two patterns with isomorphic root nodes, and yields a tree with such root containing all subtrees of both trees. The *merge* operation glues a given graph and the graph corresponding to some tree node, adjusting the graphs in all the subtrees of said node. Finally *remove* deletes a given subtree of the pattern.

Definition 12. (Concatenation)
Let PT and PT' be two pattern trees, and $m \colon \gamma_V(v) \to \gamma'_V(root')$ a morphism between a graph $\gamma_V(v) \in G$ of the first pattern and the root graph $\gamma_V(root') \in G'$ of the second pattern. We define the operation *concat* as follows:

$$
\begin{aligned}
concat(PT, PT', m\colon \gamma_V(v) \to \gamma_V'(root')) \;=\; &\langle V \cup V', \\
&E \cup E' \cup \{(v, root')\}, \\
&root, \\
&G \cup G', \\
&Mor \cup Mor' \cup \{m\}, \\
&\sigma \cup \sigma', \\
&\gamma_V \cup \gamma_V', \\
&\gamma_E \cup \gamma_E' \cup \{((v, root'), m)\}\rangle
\end{aligned}
$$

The operation can be extended in a straightforward way to sets of tuples for the second and third parameters, $concat(PT, \{\langle PT_i, m\colon \gamma_V(v_{i_n}) \to \gamma_V'(root_i)\rangle\})$, by just iterating the concatenation on all tuples in the set.

Example 4.1. Figure 11 shows an example of concatenation. The operation simply puts the second tree as subtree of the first one, through the m morphism.
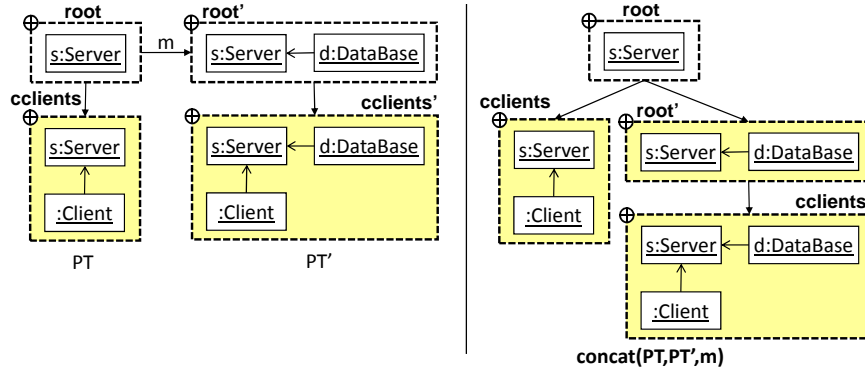

Figure 11.   Pattern tree concatenation example.

The *fuse* operation is defined in terms of *concat* by adding all subtrees of the root of the second pattern to the first one.

Definition 13. (Fuse)
Given PT and PT' with $root \cong root'$, $fuse(PT, PT')$ is the pattern tree defined as follows:

$$
fuse(PT, PT') \;=\; concat(PT, \{\langle Subtree(v_j'), \gamma_E((root', v_j'))\rangle \mid v_j' \in children(root')\})
$$

The operation can be easily generalized for an arbitrary number of arguments.

The next operation merges a graph M along a morphism $m\colon \gamma_V(v) \to M$ with all graphs in the subtree $Subtree(v)$ of v, resulting in a pattern tree, with same structure as $Subtree(v)$, but with those graphs "enlarged" according to M.

Definition 14. (Merge)
Given PT, a node $v \in V$, a graph M, and a morphism $m \colon \gamma_V(v) \to M$, we define the pattern tree graph merge as follows:

$$
\begin{aligned}
merge(PT, v, M, m \colon \gamma_V(v) \to M) \;=\; concat(\quad & tree(M, \sigma(v)), \\
& \{\langle merge(Subtree(v_i), v_i, M_i, m_i \colon \gamma_V(v_i) \to M_i), \\
& \quad m'_i \colon M \to M_i \rangle \mid v_i \in children(v)\})
\end{aligned}
$$

where M_i and $m_i \colon \gamma_V(v_i) \to M_i$ are calculated by a pushout, as shown in Figure 12.

$$
\begin{array}{ccc}
\gamma_V(v) & \xrightarrow{\;m\;} & M \\
\Big| & & \Big| \\
{\scriptstyle \gamma_E((v,v_i))}\;P.O. & & {\scriptstyle m'_i} \\
\downarrow & & \downarrow \\
\gamma_V(v_i) & \xrightarrow{\;m_i\;} & M_i
\end{array}
$$

Figure 12.   Step in the construction of the pattern tree graph merge.

We also use a variation of merge that accepts as extra argument a set $EX \subseteq V$ of excluded nodes, with $v \notin EX$, as follows:

$$
\begin{aligned}
merge(PT, v, M, m \colon \gamma_V(v) \to M, EX) \;=\; concat(\quad & tree(M, \sigma(v)), \\
& \{\langle merge(Subtree(v_i), v_i, M_i, \\
& \quad m_i \colon \gamma_V(v_i) \to M_i, EX), m'_i \colon M \to M_i \rangle \\
& \mid v_i \in children(v) \wedge v_i \notin EX\})
\end{aligned}
$$

Example 4.2. Figure 13 shows an example of merge. The operation results in a pattern tree with root M and same structure as $Subtree(root)$, containing enlarged graphs of PT with the extra elements given by M, each graph being obtained by iterated pushouts along the tree structure.
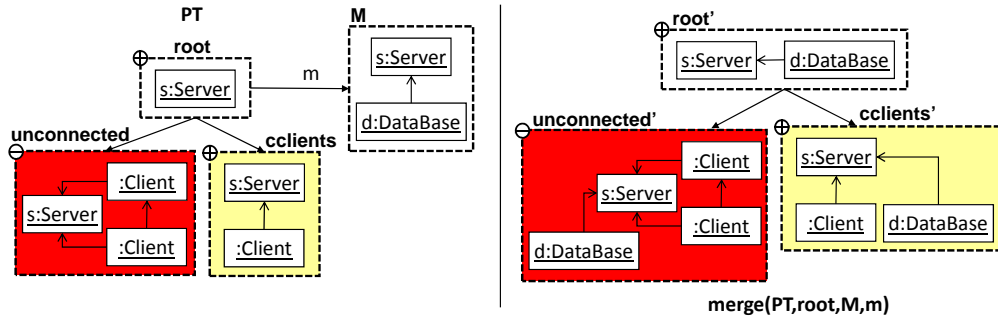

Figure 13.   Pattern tree merge example.

The next operation simply removes a subtree from a pattern tree.

Definition 15. (Remove)
Given a pattern tree $PT = \langle V, E, root, G, Mor, \sigma, \gamma_V, \gamma_E \rangle$, and $n \neq root \in V$ such that $Subtree(n) = \langle V_n, E_n, n, G_n, Mor_n, \sigma|_{V_n}, \gamma_V|_{V_n}, \gamma_E|_{E_n} \rangle$, we define the operation *remove* as follows:

$$remove(PT, n) = \langle V', E', root, G', Mor', \sigma|_{V'}, \gamma_V|_{V'}, \gamma_E|_{E'} \rangle$$

with $V' = V \setminus V_n$, $E' = E \setminus E_n \setminus \{(n', n) \in E \mid n \in children(n')\}$, $G' = G \setminus G_n$, $Mor' = Mor \setminus Mor_n \setminus \{\gamma_E((n', n)) \mid (n', n) \in E'\}$.

Next, we define a path abstraction operation, which abstracts a pattern with respect to a graph, along a given path of the tree.

Definition 16. (Path abstraction)
Let PT be a pattern tree, $M_j \in [\![PT]\!]$, $v_j \in terminal(PT)$ and $m_j \colon \gamma_V(v_j) \to M_j$. We define the *path abstraction* of M_j with respect to v_j in PT, written $pabs(PT, M_j, v_j, m_j \colon \gamma_V(v_j) \to M_j)$, as $ppabs(PT, M_j, v_j, m_j \colon \gamma_V(v_j) \to M_j, path(v_j))$, where $ppabs(PT, M_j, v_j, m_j \colon \gamma_V(v_j) \to M_j, EX)$ is defined as follows:

1. if $v_j = root$, then
$$merge(PT, v_j, M_j, m_j, EX)$$

2. if $\exists M_i$ s.t. (1) in Figure 14 is a pushout, then
$$concat(ppabs(PT, M_i, v_i, m_i, EX \cup Subtree(v_j)_V), merge(Subtree(v_j), v_j, M_j, m_j, EX), n_{ij})$$

3. else
$$remove(concat(PT, merge(PT, v_j, m_j, EX), m_j), v_j).$$

$$\gamma_V(v_i) \xrightarrow{\;m_i\;} M_i$$
$$\gamma_E((v_i, v_j))\;(1) \qquad n_{ij}$$
$$\gamma_V(v_j) \xrightarrow{\;m_j\;} M_j$$

Figure 14. Step in the construction of the path abstraction.

Remark 4.1. The abstraction traverses, bottom-up, a given path, calculating the pushout complements M_i according to Figure 14. At the same time, a top-down traversal with the *merge* operation is performed, enlarging all nodes in sibling paths. The nodes in the original path, and every subtree of these nodes, are excluded in this merge, as the enlargement of these graphs is already performed by the pushout complement in Figure 14, and the merge operation in step 2. If no such pushout complement exists, the graph $\gamma_V(v_j)$ is simply substituted by the graph M_j (see step 3), for which purpose the subtree of v_j is enlarged (via a *merge*), its result concatenated with PT, and the original subtree of v_j removed.

Example 4.3. Figure 15 shows a path abstraction example. In particular, the right part of the figure shows the result of evaluating $pabs(\mathtt{PT}, \mathtt{M}, \mathtt{cclients}, m)$. In a first step, the pushout complement $\mathtt{root'}$ of $\mathtt{root} \rightarrow \mathtt{cclients} \rightarrow \mathtt{M}$ is calculated, and a recursive call $ppabs(PT, \mathtt{root'}, \mathtt{root}, \mathtt{root} \rightarrow \mathtt{root'}, \{\mathtt{root}, \mathtt{cclients}\})$ is made. At this stage, we reach the top of the pattern tree, so that step 1 in the previous definition returns the result of merging $\mathtt{root'}$ with the tree made of $\mathtt{root}$ and $\mathtt{unconnected}$ (because $\mathtt{cclients}$ is excluded). This amounts to performing the pushout of $\mathtt{unconnected}$ obtaining $\mathtt{unconnected'}$. Finally, the trees are concatenated, obtaining the pattern tree shown in the right of the figure. Hence, it can be seen that a path abstraction involves a bottom-up traversal of a given path (computing pushout complements), and then a top-down traversal of each subtree (computing pushouts).
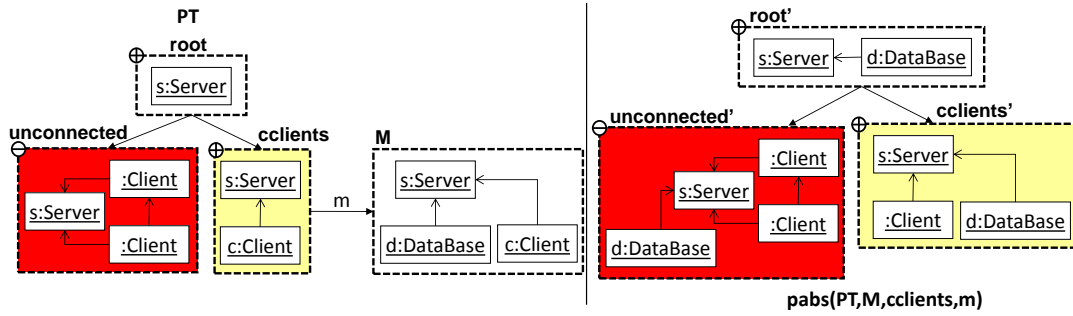

Figure 15.   Path abstraction example.

Now we are ready to define our abstraction operation. It makes use of the path abstraction operation, computing the patterns resulting from all possible path abstractions.

Definition 17. (Abstraction)
Let PT be a pattern tree, and $M_j \in [\![PT]\!]$. We define the *abstraction* of M_j with respect to PT, written $abs(PT, M_j)$ as follows:

$$abs(PT, M_j) = \{pabs(PT, v_j, M_j, m_j^i \colon \gamma_V(v_j) \rightarrow M_j) \mid v_j \in terminal(PT) \wedge m_j^i \in S_{PT, M_j}\}$$

Remark 4.2. The abstraction operation yields a set of pattern trees, for each node in the terminal set $terminal(PT)$ and each morphism used in the satisfaction checking.

Example 4.4. Figure 16 shows an abstraction example. Abstracting M with respect to PT yields two patterns, $\mathtt{PT}_1$ and $\mathtt{PT}_2$ shown to the right of the figure. $\mathtt{PT}_1$ is obtained by path abstraction using morphism m_1. This pattern reflects the fact that at least one database with a cache should exist, and then allows the same variability as PT. $\mathtt{PT}_2$ is obtained by path abstraction using morphism m_2. In this case, the pushout complement of $\mathtt{root} \rightarrow \mathtt{dbs} \rightarrow \mathtt{M}$ does not exist, and so M replaces $\mathtt{dbs}$ in PT.

Many other procedures and strategies for abstraction are possible. In particular, in the previous example, one possible abstraction would yield a pattern like PT, but with graph dbs enlarged with a cache object connected to the database object. While our abstraction uses information on one branch of
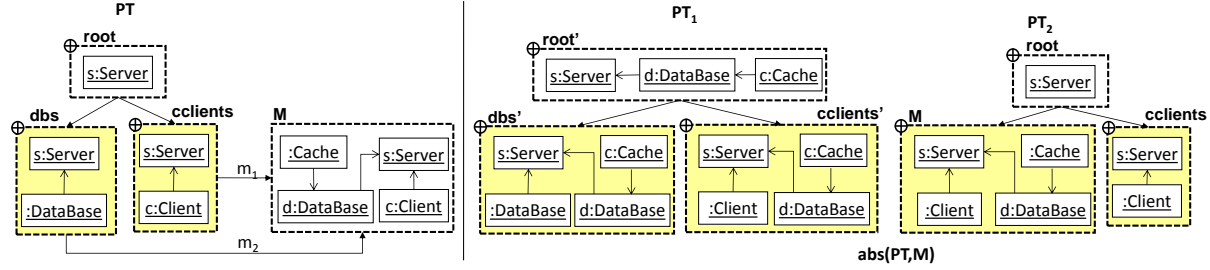
Figure 16.   Abstraction example.

the pattern tree at a time, this abstraction would require a more complex analysis of several branches of the pattern tree at the same time, which we leave for future work.

We next show that our abstraction strategy actually fulfils two desired properties for any abstraction strategy. The first one states that the abstraction should be "exact". That is, the graph should belong to the language of the resulting abstracted pattern(s). The second property states that there should be an SPT-morphism from the original pattern to the abstracted pattern(s). The existence of such a morphism is naturally expected, as the original model M belongs to the semantics of PT by the existence of a family of (graph) morphisms. Intuitively, such a family is included in the semantics of the SPT-morphism.

Theorem 4.1. (Abstraction implies language inclusion)
Let PT be a 1-negation pattern tree and $G \in [\![PT]\!]$, then $G \in L(PT')$ with $PT' \in abs(G, PT)$.

Proof:
(Sketch) Let $PT' \in abs(G, PT)$. If, in the construction of Definition 16, step 2 is taken, M_j is incorporated into PT', so it is clear that $S_{PT',G}$ is surjective, hence $G \in L(PT')$. Similarly, if step 3 is taken, then M_j is also incorporated into PT', and again $S_{PT',G}$ is surjective, so that $G \in L(PT')$.   □

Theorem 4.2. (Induced abstraction SPT-morphism)
Let PT be a 1-negation pattern tree and $G \in [\![PT]\!]$, then $\exists m \colon PT \to PT'$ with $PT' \in abs(G, PT)$, where m is an SPT-morphism.

Proof:
(Sketch) Each $PT' \in abs(G, PT)$ is constructed by *ppabs*. Patterns PT and PT' are structurally equal, but the graphs in PT' are those of PT enlarged, and a morphism between them exists as shown in Figure 14 (where a PO-complement is built), and Figure 12 (where a PO is done, in the *merge* operations in steps 1 and 3). The induced morphism is SPT because, assuming that the underlying graph category used is adhesive, pushouts along monomorphisms are pullbacks, as required by the second condition of SPT-morphism. In any case, PT' does not add new negative regions, and signs are preserved.   □

5.   Pattern-based Rewriting

This section introduces a notion of rewriting on patterns, based on rules having patterns in their pre- and post-conditions leveraging the theory of algebraic graph transformation [21], in which pushout complements are used to model deletion of elements, and pushout objects are used for the creation of elements. Hence, we next show how pushouts for pattern trees are built.

Definition 18. (Construction of Pushouts for Pattern Trees)
Given the span of $PT - morphisms$ $PT_1 \overset{m_1}{\Leftarrow} PT_0 \overset{m_2}{\Rightarrow} PT_2$, the pushout $PT_1 \overset{m'_1}{\Rightarrow} PT \overset{m'_2}{\Leftarrow} PT_2$ is calculated by the function $glue(PT_1 \overset{m_1}{\Leftarrow} PT_0 \overset{m_2}{\Rightarrow} PT_2, \gamma_V^1(root^1) \overset{m_1^{root}}{\Leftarrow} \gamma_V^0(root^0) \overset{m_2^{root}}{\Rightarrow} \gamma_V(root^2))$, defined as follows:

$$glue(PT_1 \overset{m_1}{\Leftarrow} PT_0 \overset{m_2}{\Rightarrow} PT_2, \gamma_V^1(v_i^1) \overset{m_1^i}{\Leftarrow} \gamma_V^0(v_i^0) \overset{m_2^i}{\Rightarrow} \gamma_V(v_i^2)) = fuse(concat(tree(v_i, \sigma^0(v_i^0)),$$
$$\{\langle glue(PT_1 \overset{m_1}{\Leftarrow} PT_0 \overset{m_2}{\Rightarrow} PT_2, \gamma_V^1(v_j^1) \overset{m_1^j}{\Leftarrow} \gamma_V^0(v_j^0) \overset{m_2^j}{\Rightarrow} \gamma_V(v_j^2)), u_j \rangle |$$
$$v_j^0 \in children(v_i^0) \wedge v_j^1 \in children(v_i^1) \wedge v_j^2 \in children(v_i^2)\}),$$
$$merge(PT_1, v_i^1, v_i, n_1^i, Cod_V(m^1)),$$
$$merge(PT_2, v_i^2, v_i, n_2^i, Cod_V(m^2)))$$

where each v_i is the pushout object of $\gamma_V^1(v_i^1) \overset{m_1}{\Leftarrow} \gamma_V^0(v_i^0) \overset{m_2}{\Rightarrow} \gamma_V(v_i^2)$, $u_j \colon v_i \to v_j$ uniquely exists due to the PO universal property, (see Figure 17), and $Cod_V(m^k)$ is the set of graphs in the co-domain of morphism m^k, for $k = 1, 2$.
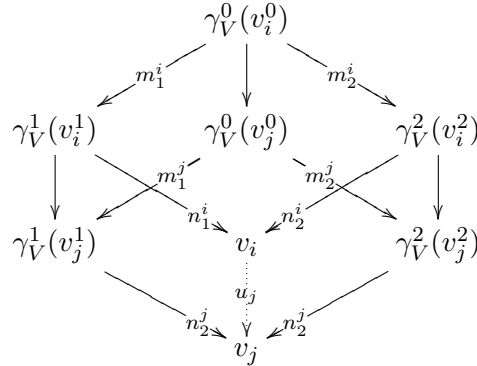

Figure 17.   Calculating a pushout on patterns

The *glue* function builds pushouts along the tree structure of PT_0, by recursive calls. The subtrees obtained in the recursive calls are added through the *concat* operation. In addition, we need to add the non-mapped regions of PT_1 and PT_2, which is made by calling *merge* and then fusing the obtained trees with the one originating from the *concat*.

Example 5.1. Figure 18(a) shows a pushout example and Figure 18(b) shows the details of the calculation. In particular, the extra region `nodirect` is added to PT via the *merge* operation.

Theorem 5.1. (Pushouts for Pattern Trees)
The category **PattTree** admits pushouts, built according to Definition 18.
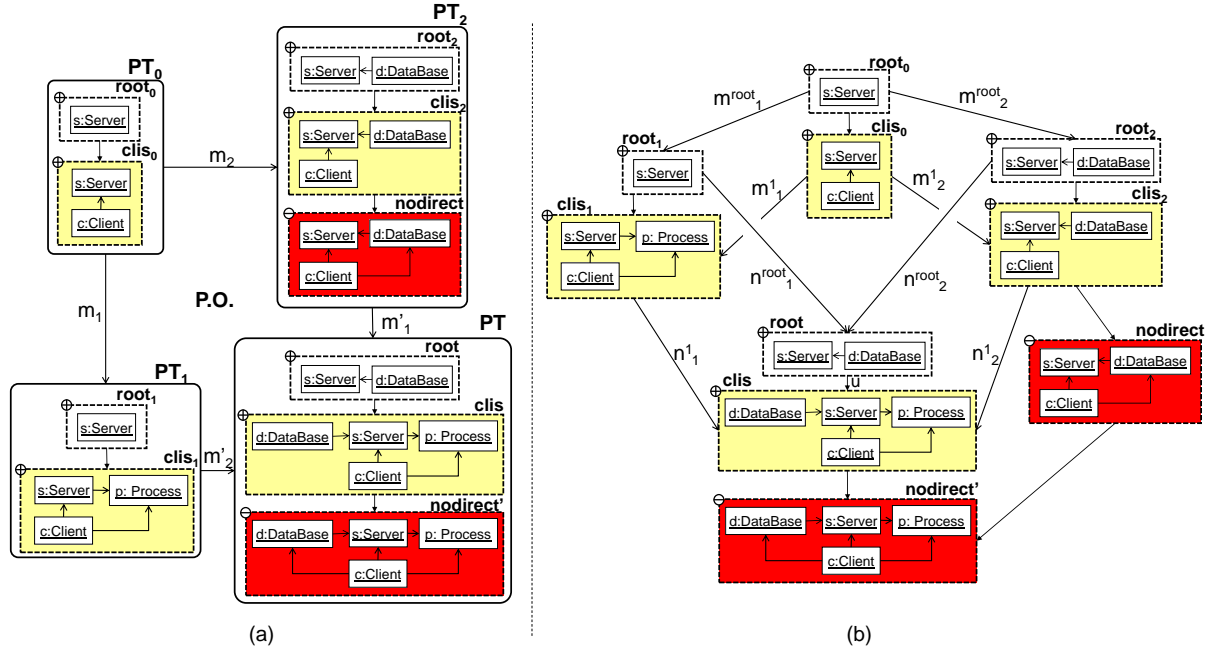
Proof:
See Appendix 9. □

Figure 18. (a) Pushout example. (b) Details of the pushout construction.

If the two PT-morphisms in the span $PT_2 \xleftarrow{g} PT_0 \xrightarrow{f} PT_1$ are SPT-morphisms, then the complement morphisms in the pushout are also SPT-morphisms if the resulting pattern is satisfiable.

Theorem 5.2. (Pushouts along SPT-Morphisms)
Given the pushout in Figure 19, f and g are SPT-morphisms *iff* $f*$ and $g*$ are SPT-morphisms.

$$
\begin{array}{ccc}
PT_0 & \xrightarrow{\ f\ } & PT_1 \\
\downarrow{g} & \text{P.O.} & \downarrow{g*} \\
PT_2 & \xrightarrow{\ f*\ } & PT
\end{array}
$$

Figure 19. Pushouts along SPT-morphisms

Proof:
(Sketch) If both f and g are SPT-morphisms, this means that every cube of the pushout is a van Kampen square [40]. Hence, in Figure 17, the top and bottom faces are pushouts and the back faces are pullbacks, hence the front faces are pullbacks. Moreover, from their being SPT-morphisms follows that all negative regions of PT_1 and PT_2 are mapped from PT_0, and so no new negative region appears in PT that is not in both PT_1 and PT_2. Hence, we can conclude that $f*$ and $g*$ are SPT-morphisms.

Conversely, if $f*$ and $g*$ are SPT-morphisms, the front faces in Figure 17 are pullbacks, and therefore by the van Kampen property so are the back faces. Moreover, all negative regions of PT are mapped from both PT_1 and PT_2. Because of the pushout properties, this means that those negative regions

should be mapped from PT_0 as well (otherwise they would not be mapped in PT from both PT_1 and PT_2, but they would be replicated in PT) and so f and g are SPT-morphisms. $\qquad\square$

Example 5.2. Figure 20 shows an example of pushout merging two patterns: PT_1, indicating that several caches can be associated with the same database, and PT_2, indicating that access to the database occurs through a process, but not directly. Note that as it is not the case that both m_1 and m_2 are SPT-morphisms (in particular m_2 is not), then it is not the case that both m_1' and m_2' are SPT-morphisms (in particular, m_2' is not).
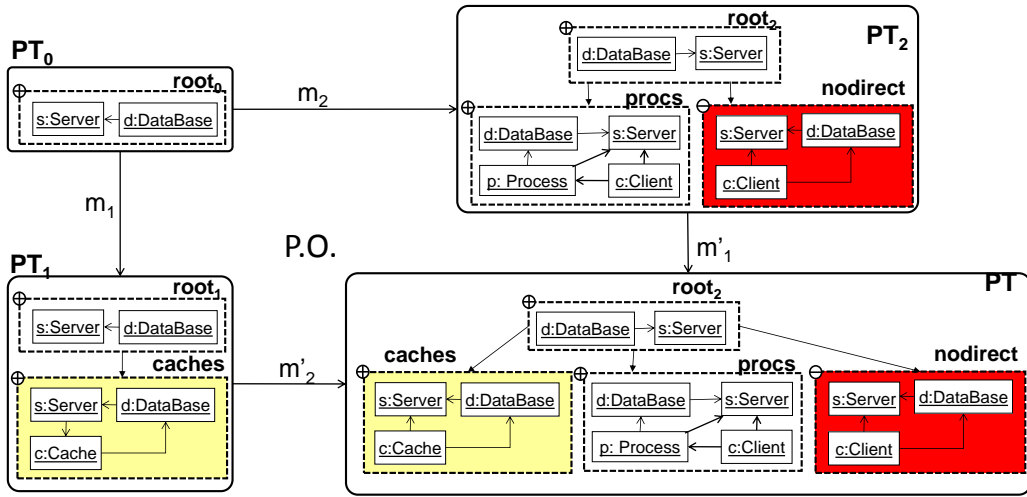

Figure 20.   Pushout example.

Next, we investigate a compatibility condition between pushouts for pattern trees (along SPT-morphisms) and pushouts for graphs, which will be useful to ensure that the rewriting of abstracted graphs is correct.

Theorem 5.3. (Pushout compatibility)
Given a span of SPT-morphisms $PT_1 \overset{m_1}{\leftarrow} PT_0 \overset{m_2}{\to} PT_2$ and a span of injective graph morphisms $G_1 \overset{\lfloor m_1 \rfloor}{\leftarrow} G_0 \overset{\lfloor m_2 \rfloor}{\to} G_2$ with $G_i \in L(PT_i)$ (for i=0, 1, 2), and $\lfloor m_j \rfloor \in [\![m_j]\!]$ (for j=1, 2), then $G \in L(PT)$ with G the pushout object graph of $G_1 \overset{\lfloor m_1 \rfloor}{\leftarrow} G_0 \overset{\lfloor m_2 \rfloor}{\to} G_2$ and PT the pushout pattern tree of $PT_1 \overset{m_1}{\leftarrow} PT_0 \overset{m_2}{\to} PT_2$ (see Figure 21).

Proof:
See Appendix 9. $\qquad\square$

Example 5.3. Figure 22 shows an example of pushout compatibility. Figure (a) shows a pushout on pattern trees, where each morphism is an SPT-morphism. Figure (b) shows a pushout on graphs, where $G_i \in L(PT_i)$, for $i = 0, 1, 2$, and $\lfloor m_i \rfloor \in [\![m_i]\!]$, for $i = 1, 2$. The resulting pushout object $G \in L(PT)$. This is so as SPT-morphisms do not add new negative regions to PT, and negative regions are only weakened. Moreover, as $\lfloor m_i \rfloor \in [\![m_i]\!]$, the morphisms ensure that there is no occurrence of a variable region mapped by PT_0 that is not mapped in G_1 or G_2. Should we make pushouts in
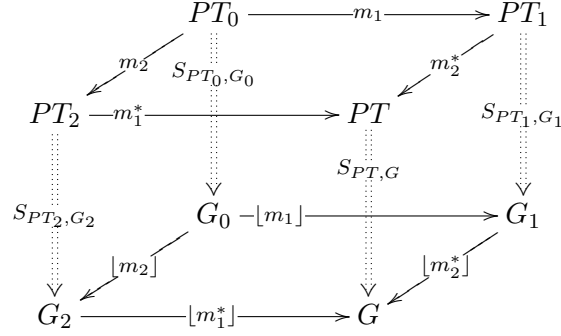
Figure 21.   Pushout compatibility.

graphs through arbitrary morphisms, we would end up with a graph $G \notin L(PT)$, but in $[\![PT]\!]$ only, as Figure 21 (c) shows. In that case, the resulting pushout pattern tree PT demands merging regions $clis_2$ and $clis_1$. However, at the graph level, the *Client* object in G_2 was not identified by G_0, and hence a full synchronization of subgraphs satisfying regions $clis_2$ and $clis_1$ has not occurred.
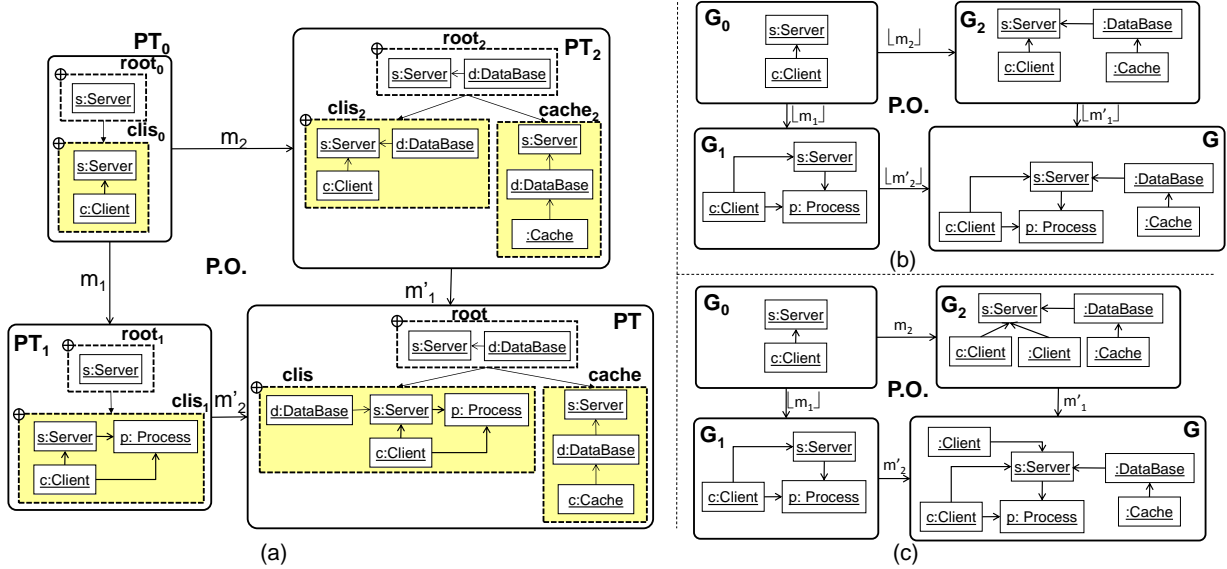

Figure 22.   Pushout compatibility example.

Next, we identify when pushout complements exist, as we need them to express deletion. To this end, we first review the *dangling edge* condition for pushout complements on graphs, taken from [21] (more precisely, see Definition 3.9 and Fact 3.11 there, where we do not need the identification condition, as we work with injective morphisms). This condition basically states that if a node is "deleted" all its stemming edges should be deleted as well.

Theorem 5.4. (Pushout complements in graph)
Let $K \xrightarrow{l} L \xrightarrow{m} M$ be a chain of two graph morphisms, and let $GN = \{n \in L_V \setminus l(K_V)\}$ be the set of glueing nodes for l. The *dangling edge* condition is satisfied if $\forall n \in GN[\forall edge\ e \in M[src(e) = m(n) \vee tar(e) = m(n) \implies \exists edge\ e' \in L[m(e') = e]]]$. If the dangling edge condition is satisfied, a graph D and morphisms $K \xrightarrow{d} D \xrightarrow{l*} M$ exist, making the square a pushout.

Proof:
In [21]. □

Now, we enumerate the conditions needed for the existence of pushout complements in pattern trees.

Theorem 5.5. (Pushout complements in pattern trees)
Given the chain of *PT-morphisms* $K \xrightarrow{l} L \xrightarrow{m} M$, the pattern D and the morphisms $K \xrightarrow{d} D \xrightarrow{l*} M$ exist, making the square a pushout if:

1. The dangling edge condition is satisfied in every graph morphism $m_G^i \circ l_G^i$ with $l_G^i \in l_F$ and $m_G^i \in m_F$.

2. The dangling region condition is satisfied: Let the set of glueing regions be $GR = \{v_i \in V_L \setminus l(V_K)\}$. This condition is satisfied if $\forall v_i \in GR[\forall v \in children(m(v_i))[\exists v' \in GR[m(v') = v \wedge v' \in children(v_i)]]]$.

3. The graphs in $m_V(GR)$ should be constructible by pushouts: $\forall v_i^L \in GR$, the diagram in Figure 23 is a pushout, where $(v_j^L, v_i^L) \in E_L$, and $(v_j^M, v_i^M) \in E_M$.

4. The graphs in $V_M \setminus m_V(V_L)$ should be constructible by pushouts. For this purpose, let $GR' = \{v_i^M \in V_M \setminus m(V_L)\}$. Let $v_j \in V_M$ with $v_i \in GR'$, and $(v_j, v_i) \in E_M$. Let $v_j^L \in V_L$ with $m(v_j^L) = v_j$ and $v_j^K \in V_K$ with $l(v_j^K) = v_j^L$, and v^D the pushout complement of $\gamma_V^M(v_j) \leftarrow \gamma_V^L(v_j^L) \leftarrow \gamma_V^K(v_j^K)$. We require the dangling edge condition to be satisfied for $\gamma_V^M(v_i) \leftarrow \gamma_V^M(v_j) \leftarrow v^D$.

$$
\begin{array}{ccc}
\gamma_V(v_j^L) & \longrightarrow & \gamma_V(v_i^L) \\
\downarrow l_G^j & P.O. & \downarrow l_G^i \\
\gamma_V(v_j^M) & \longrightarrow & \gamma_V(v_i^M)
\end{array}
$$

Figure 23. Condition for pushout complements

Proof:
See Appendix 9. □

Remark 5.1. In condition 4, the pushout complement v^D exists, as according to property in item 1, every graph satisfies the dangling edge condition.

The *dangling edge* condition is the one for pushout complements in graphs. In pattern trees, it should be satisfied in all regions of the pattern trees. The *dangling region* condition states that if some region is deleted, all its nested regions should be explicitly deleted as well. The third and fourth conditions state that the graphs in M that are not mapped from K should be constructible by pushouts. In particular, the third condition states that the graphs "coming from L only" should be constructible by pushouts, while the fourth condition states that the graphs "coming from D only" should be constructible by pushouts as well. These last three conditions are specific to pattern trees and do not occur in pushouts for graphs.

Example 5.4. Figure 24 shows two examples where each of the two first conditions for existence of pushout complements is violated. In the first case, the pushout complement has the effect of deleting the database node d. However, this is not possible in the nested region, as a dangling edge would occur (the edge connecting the client and the database). In the second case, the pushout complement deletes region $clis_L$. However, it is not possible to achieve a pushout, as region $querys_M$ would not be mapped from the pushout complement D.
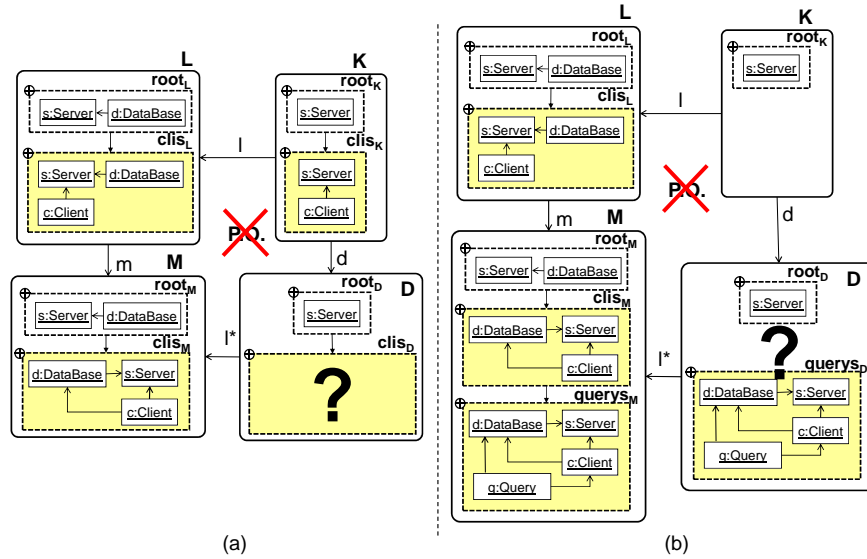

Figure 24.   Conditions for existence of pushout complements (a) Violation of dangling edges condition. (b) Violation of dangling regions condition.

Figure 25 shows other two examples violating conditions 3 and 4. Figure 25(a) shows that by deleting region $clis_M$ it is not possible to obtain a pattern tree forming a pushout. This is so, as the pushout construction requires graph $clis_M$ to be the pushout object of $root_M \overset{m}{\leftarrow} root_L \overset{e}{\rightarrow} clis_L$, but it is not so, since it adds an edge between the client and the database. This means that we cannot remove one region from D if such region is not matched with an identity morphism. Figure 25 (b) shows a violation of condition 4, which can be seen as a *deep deletion* condition. The problem here is that, we are deleting the *DataBase* from the root, but it produces a dangling edge in region $clis_M$.

Next, we enunciate a similar compatibility property to the one for pushouts. However, in this case, we need one more condition ensuring that if the pushout complement exists in patterns, it exists in graphs. Again, this is needed to ensure correctness of rewriting of abstracted graphs.
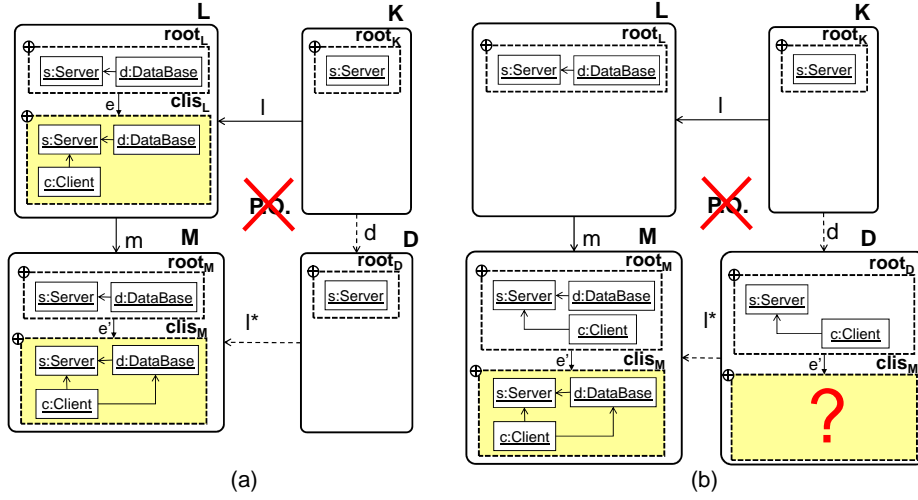
Figure 25.   Conditions for existence of pushout complements (a) Violation of condition 3. (b) Violation of condition 4.

Theorem 5.6. (Pushout complement compatibility)

Given a chain of SPT-morphisms $PT_0 \overset{m_1}{\to} PT_1 \overset{m_2}{\to} PT_2$ and a chain of injective graph morphisms $G_0 \overset{\lfloor m_1 \rfloor}{\to} G_1 \overset{\lfloor m_2 \rfloor}{\to} G_2$ with $G_i \in L(PT_i)$ (for i=0, 1, 2), and $\lfloor m_j \rfloor \in [\![m_j]\!]$ (for j=1, 2), then if PT exists s.t. the upper square in Figure 26 is a pushout, then G exists s.t. the bottom square in the figure is a pushout, and $G \in L(PT)$.

Figure 26.   Pushout complement compatibility.

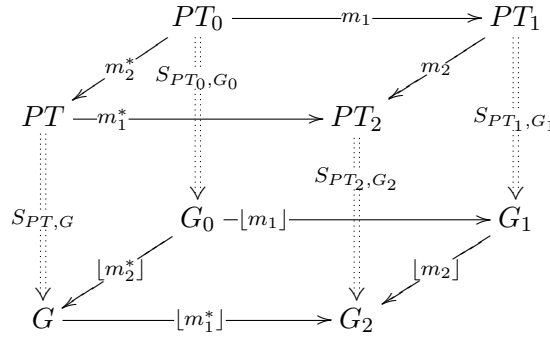### Proof:
See Appendix 9.   □

Example 5.5. Figure 27 shows an example of pushout complement compatibility. The pushout complement PT exists, because $PT_0 \to PT_1 \to PT_2$ satisfies all conditions of Theorem 5.5. This means in particular, that every square formed by the graph morphisms between the patterns is a pushout, and hence satisfies the dangling edge conditions. For this reason, the pushout complement of $G_0 \to G_1 \to G_2$
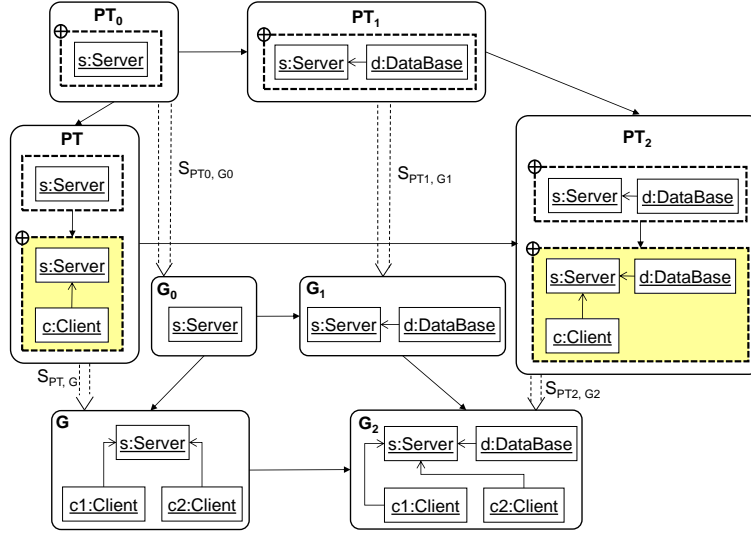
Figure 27. Compatibility of pushout complements.

(graph G) exists. Intuitively, at the pattern level, building the pushout complement requires deleting the database node. This action is ensured to be admissible in graphs belonging to the pattern language, because they do not have extra elements which could cause dangling edges, and therefore it is admissible in G_1, and so G can be constructed.

5.1. Pattern Rules

We use Double Pushout rules [21] made of a span of PT-morphisms: $L \xleftarrow{l} K \xrightarrow{r} R$. While rules are sometimes equipped with NACs, expressing forbidden execution context, we omit them here for simplicity. In any case, the fact that the left-hand side pattern L may contain negative regions allows expressing forbidden context. We distinguish a special kind of rule, called SPT rule, in which the l and r PT-morphisms are SPT-morphisms.

Definition 19. (Pattern rule)
A pattern rule $p = \langle L \xleftarrow{l} K \xrightarrow{r} R \rangle$ is made of a span of PT-morphisms. The rule p is an SPT rule, if l and k are SPT-morphisms.

A pattern rule can be applied to a pattern PT if a PT-morphism $m \colon L \to PT$ is found, and the pushout squares (1) and (2) in the diagram of Figure 28 can be built. In a first step, the pushout complement PT' is calculated. Such complement exists if the conditions of Theorem 5.5 are satisfied. Hence, this first step (roughly) deletes the elements in $L \setminus K$ from PT, yielding PT'. The second pushout adds the elements in $R \setminus K$ to PT', yielding PT''.

Definition 20. (Rule applicability)
Given a pattern rule $p = \langle L \xleftarrow{l} K \xrightarrow{r} R \rangle$, and a pattern PT, we say that p can be applied at a PT-morphism $m \colon L \to PT$, written $m \models p$, if m satisfies the dangling conditions of Theorem 5.5.

$$L \xleftarrow{\quad l \quad} K \xrightarrow{\quad r \quad} R$$

$$m \downarrow \quad (1) \quad d \downarrow \quad (2) \quad h \downarrow$$

$$PT \xleftarrow{\ l* \ } PT' \xrightarrow{\ r* \ } PT''$$

Figure 28.   Rule application.

Definition 21. (Rule application)

Given a pattern rule $p = \langle L \xleftarrow{l} K \xrightarrow{r} R \rangle$, a pattern PT, and a PT-morphism $m \colon L \to PT$, s.t. $m \models p$, p can be applied, written $PT \overset{p,m}{\Rightarrow} PT''$, by constructing the pushouts shown in the diagram of Figure 28.

Example 5.6. Figure 29(a) shows an example pattern rule application. The rule reorganizes a database server into a 2-tier architecture, by adding a backend server. The rule also adds a negative region forbidding direct accesses to the added backend server. Figure 29(b) shows a compact notation for rules, which omits the intermediate pattern K, and shows the patterns themselves in a compact way. This is the notation that we will use in Section 7.
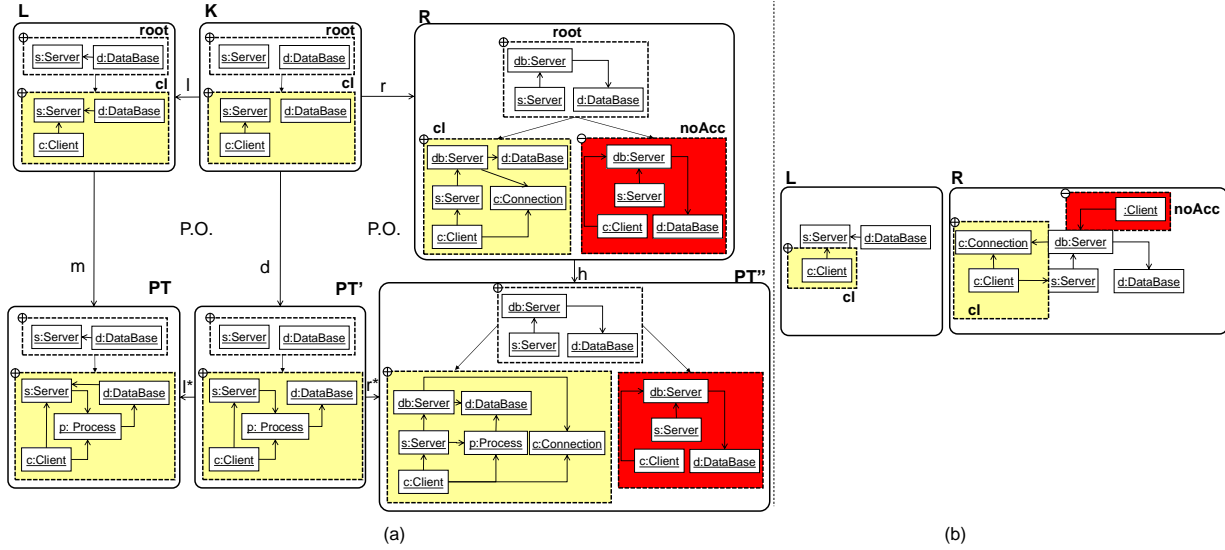

Figure 29.   (a) Pattern rule application example. (b) Pattern rule in compact notation.

Next, we show that rewriting using SPT rules is compatible with the abstraction. Assume that, given an SPT rule p, and a graph $G \in [\![L]\!]$, we abstract G according to the left hand side L of the rule p, obtaining a set of patterns $abs(G, L)$. According to theorem 4.2, for every pattern $PT_G \in abs(G, L)$ there is an induced SPT-morphism $m \colon L \to PT_G$. We rewrite PT_G through that morphism, $PT_G \overset{p,m}{\Rightarrow}$ PT_G'' yielding PT_G''. The compatibility condition must ensure that rewriting G using a graph rule $\widehat{p_G}$ (derived from p) yields a graph G'' s.t. $PT_G'' \in abs(R, G'')$. In order to show this result, we proceed in two steps. First, we describe how to obtain a concrete rule $\widehat{p_G}$, derived from a pattern rule p and a graph G. Second, we show that the result of the rules on G and an abstraction of it are equivalent.

Definition 22. (Concretized rule)
Given an SPT rule $p = \langle L \xleftarrow{l} K \xrightarrow{r} R \rangle$ and a graph $G \in [\![L]\!]$, the set of concretized rules of p with respect to G, written $\widehat{P_G}$ is given by the (possible infinite) set $\{\widehat{p_j^i} = \langle L_j \xleftarrow{\lfloor l_j \rfloor} K_j \xrightarrow{\lfloor r_j^i \rfloor} R_j^i \rangle\}$, where the rules are calculated as shown in Figure 30, and:

1. $PT_j \in abs(G, L)$,

2. $m_j \colon L \to PT_j$ is the induced abstraction SPT-morphism, according to Theorem 4.2,

3. L_j and $\lfloor m_j \rfloor \colon L_j \to G$ are calculated according to Lemma 3.1,

4. K_j and $\lfloor l_j \rfloor \colon K_j \to L_j$ are calculated according to Lemma 3.1,

5. R_j^i and $\lfloor r_j^i \rfloor \colon K_j \to R_j^i$ are calculated according to Theorem 3.4 (see Appendix).
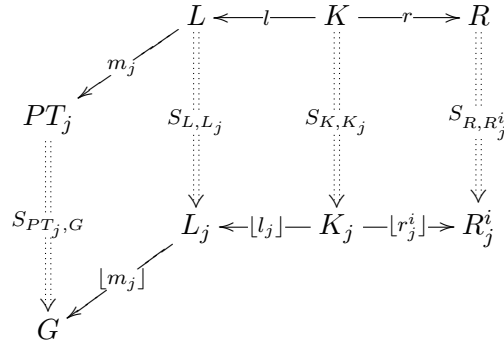

Figure 30.   Calculating a concretized rule.

Remark 5.2. The set in $\widehat{P_G}$ contains rules generated due to two factors. The first is the set $abs(G, L)$, as each $PT_j \in abs(G, L)$ induces a different $m_j \colon L \to PT_j$. This reflects the fact that there may be different occurrences of the rule in G, which means different concretized rules. The second is R_j^i, which reflects the fact that the extra variable regions in $R \setminus r(K)$ can be instantiated an arbitrary number of times, leading to different rules.

Example 5.7. Figure 31 shows a rule concretization example. The rule is an SPT-rule, and is concretized with respect to graph G. The abstraction yields two patterns, PT_1 and PT_2. The figure shows the details of the calculation of the first rule concretized rule, while the second rule is the same as the first one.

Theorem 5.7. (Concretized and pattern rule execution)
Given an SPT rule $p = \langle L \xleftarrow{l} K \xrightarrow{r} R \rangle$ and a graph $G \in [\![L]\!]$, we have that $PT_j \xRightarrow{p, m_j} PT_j''$, with $PT_j \in abs(G, L)$, then $\exists \widehat{p_j^i} \in \widehat{P_G}$ s.t. $G \xRightarrow{\widehat{p_j^i}, \lfloor m_j \rfloor} G''$ with $G'' \in [\![PT_j'']\!]$, as Figure 32 shows.

Proof:
(Sketch) Follows from the compatibility of pushouts (Theorem 5.3) and pushout complements (Theorem 5.6). $\qquad\square$
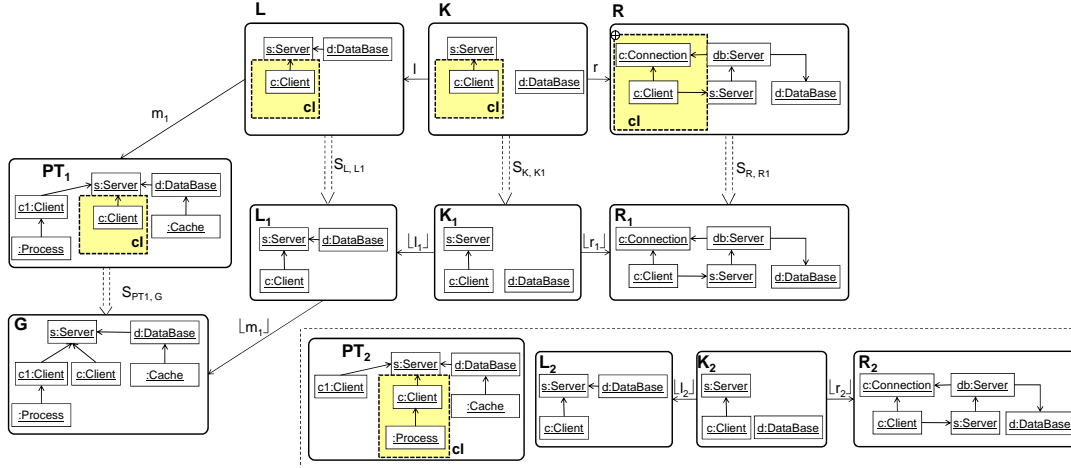
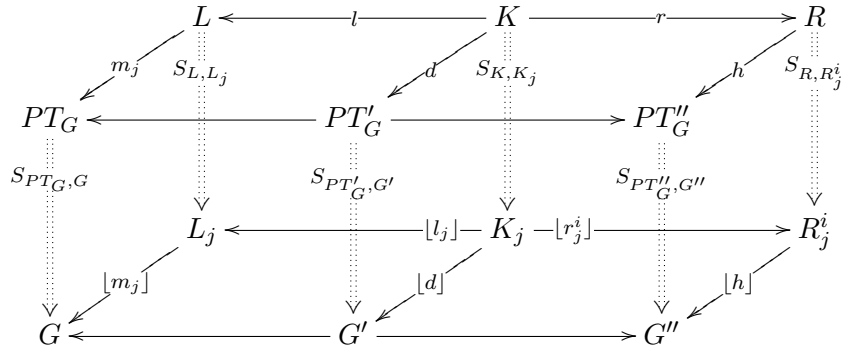Figure 31.   Rule concretization example.

Figure 32.   Concretized and pattern rule execution.

Example 5.8. Figure 33 shows an example of execution of a pattern rule and a concretized rule, showing their compatibility.

6.   Pattern Variants

A number of design patterns [25] occur in *variants*, the most notable being those based on distinguishing between *inheritance* and *delegation*. A typical example is the `Adapter` pattern, where a class, the *adapter*, is created to compound the gap between the interface of an offered operation and the signature of an available method implementing the semantics of the operation.

In other occasions, the difference between two pattern variants lies in the degree of variability offered by each pattern version. For example, in the `Adapter` pattern we may require all operations to be implemented by one class, or require a different class to implement each method.

Hence, we can build pattern variants in two ways. The first is by applying a pattern rule to an existing base pattern. The second is by changing the variability offered by the pattern. While the techniques for

Figure 33. Pattern rule and concretized rule execution example.

applying pattern rules have been studied in the previous section, this section introduces an operation to derive a set of pattern variants, which differ in the allowed variability. The main idea of the operation is shown in the following example.

Example 6.1. Figure 34 shows some variants of the base pattern PT_0, resulting from choosing different variability from the root. PT_1 allows several DBServers connected to the s Server, each with one DataBase. PT_2 is derived from PT_1 by allowing several DataBases in each DBServer. Finally, PT_3 is derived from PT_0, enabling one DBServer only, but each with one or more DataBases.

Figure 34. Pattern variants example.

Now, we define the operation to increase the variability of a pattern. The operation takes as parameters a pattern tree PT and two morphisms $a\colon \gamma_V(v_i) \to M$, $b\colon M \to \gamma_V(v_j)$, where M is the extra variability to be inserted.

Definition 23. (Insert variability)
Let PT be a pattern tree. The insert variability operation is defined as:

$$insert(PT, a\colon \gamma_V(v_i) \to M, b\colon M \to \gamma_V(v_j)) \quad = remove(\quad concat(PT,$$
$$concat(Tree(M, \oplus), Subtree(v_j), b), a),$$
$$v_j)$$

with $a\colon \gamma_V(v_i) \to M$, $b\colon M \to \gamma_V(v_j)$ injective, $(v_i, v_j) \in E$, $v_i, v_j \neq root$ and $\sigma(v_i) = \sigma(v_j) = \oplus$.

For the case of insertion of variability from the root, we define another version of *insert* as follows:

$$insert(PT, b\colon M \to \gamma_V(root)) \quad = concat(Tree(M, \oplus), PT, b)$$

Insertion simply occurs by substituting $Subtree(v_j)$ in PT by a subtree rooted in M and with $Subtree(v_j)$ as child. The operation distinguishes when the variability is to be inserted between two elements v_i, v_j with $(v_i, v_j) \in E$, and when it is to be inserted atop the root.

Example 6.2. In Figure 34, $\mathtt{PT_1} = insert(\mathtt{PT_0}, b\colon \mathtt{Server} \to \mathtt{root})$, with $\mathtt{Server}$ the graph made of just the server node. Similarly, $\mathtt{PT_2} = insert(\mathtt{PT_1}, \mathtt{DBServs}, a\colon \mathtt{root} \to \mathtt{DBServs}, b\colon \mathtt{DBServs} \to \mathtt{DBs})$, with $\mathtt{DBs}$ the root graph of $\mathtt{PT_0}$ and $\mathtt{DBServs}$ the root graph of $\mathtt{PT_3}$, and $\mathtt{PT_3} = insert(\mathtt{PT_0}, b\colon \mathtt{DBServs} \to \mathtt{root})$.

The set of pattern variants of a pattern PT, $var(PT)$, is the set of all patterns that can be constructed in this way. Next, we investigate the effects of the variant pattern with respect to satisfaction of the original pattern.

Theorem 6.1. (Variant Language)
If PT does not have negative regions, $L(PT) \subset L(PT')$, for each $PT' \in var(PT)$.

Proof:
(Sketch) Let $G \in L(PT)$ and take $PT' \in var(PT)$ assuming just one inserted graph M between v_i and v_j. For all occurrences of $\gamma_V(v_i)$ in G there is at least one occurrence of the whole $\gamma_V(v_j)$. Now, in any $G' \in L(PT')$, for each occurrence of $\gamma_V(v_i)$ there is at least one occurrence of M and for each occurrence of M there is at least one occurrence of the whole $\gamma_V(v_j)$. As in PT we can identify a chain of subgraphs $\gamma_V(v_i) \to M \to \gamma_V(v_j)$ for each occurrence of $\gamma_V(v_i)$ (or no occurrence of $\gamma_V(v_i)$ at all), we have that $G \in L(PT')$, where $\gamma_V(v_j)$ occurs once for each occurrence of M. Note that we do not have equality of languages, because $L(PT')$ admits graphs with several occurrences of $\gamma_V(v_j)$ for each occurrence of M. The reasoning can be completed by induction on the inserted graphs. $\qquad\square$

7. Case Studies

In this section, we show the applicability of our approach by defining some pattern-based rules enabling: (a) the description of refactorings of models towards patterns, (b) modelling how to reconfigure a system by changing one pattern by an alternative one, and (c) explaining how to define pattern variants and to alternate between them. The examples used are in the domain of enterprise application architecture [23], and object-oriented design [25].

7.1. Refactoring towards patterns and pattern-based reconfiguration

Different patterns have emerged over the years to help in architecting enterprise applications [23]. They consider general principles, like layering, and must address all aspects of the enterprise application, like the domain organization, persistence, presentation, concurrency, efficiency and distribution. In many cases, the architect needs to decide between several options. For example when organizing the domain logic, it is possible to follow the *transaction script*, *domain model* or *table module* patterns [23], each one with different trade-offs depending on how complex the domain logic is. For this purpose, a way to understand, and automate, how to change a design for the usage of different patterns, or refactor an existing design towards a certain pattern would be helpful in this situation. This subsection shows how our pattern rules can describe and automate these two activities.

Distribution patterns provide strategies aimed at improving performance when there is a need to access remote objects. One of such patterns is *Remote Facade*, which is recommended whenever one needs to access fine-grained remote objects [23]. In this case, the pattern proposes the use of a coarse-grained facade for the remote objects to improve efficiency over the network, since methods in the facade can encapsulate a number of invocations to the fine-grained objects. Figure 35 (a) shows this distribution pattern expressed as a pattern tree. For the examples in this section we use stereotypes (like *local* or *facade*) as a way to annotate the roles of the different elements in the pattern. Figure 35 (b) shows another distribution pattern called *Data Transfer Object*, which encapsulates the data interchanged between classes into a single object to reduce the number of method calls [23], as well as the number of parameters in the calls. The pattern includes a data transfer object (DTO) that holds all data for the method call, and an assembler class in the server in charge of transferring data between the data transfer object and the domain objects. In the pattern trees of this section, the underlying graph category that we use here is attributed typed graphs [21] and hence we use variables as values of attributes, like *n* and *m*.
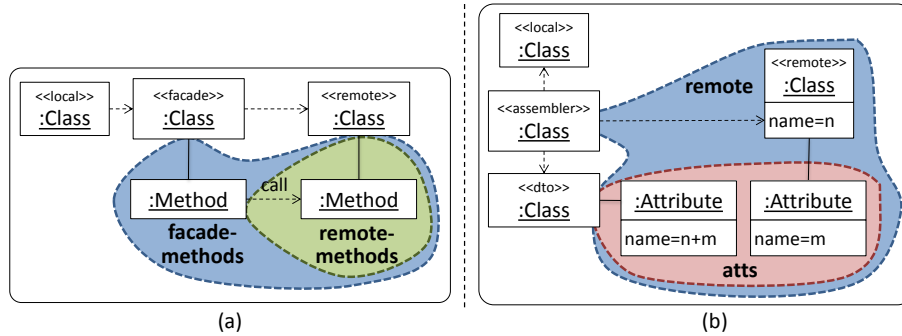

Figure 35. (a) *Remote Facade* pattern. (b) *Data Transfer Object* pattern.

It is interesting to have a means to structure an existing design towards these patterns. As an example, the rule in Figure 36 refactors a model towards the *Data Transfer Object* pattern. In this way, it applies the pattern to a particular class annotated as "remote", and a non-empty set of its attributes annotated as "accessed". The rule creates an assembler class for the remote class, and data transfer object with attributes for all accessed ones.

In addition to refactoring an architectural model towards a pattern, sometimes it may be necessary to reconfigure the actual architecture by changing one architectural pattern by another one. For this
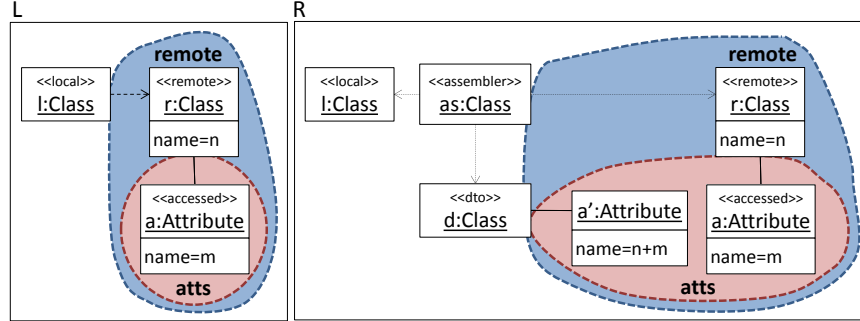
Figure 36.   Refactoring towards the *Data Transfer Object* pattern.

purpose, we can use pattern rules describing a possible migration strategy for reconfiguring a system that uses the first pattern, into a system that uses the second one. This kind of rules are useful for analysing the impact of every design decision, and are especially valuable in software evolution, migration and modernization, to reduce effort and errors [50].

For instance, Figure 37 shows two rules to switch between the remote facade and data transfer object architectural options. In particular, rule (a) identifies all remote classes that are accessed by a local class through remote facades (variable region `remote`). This rule removes the facade as well as its methods (variable region `methods`, appearing in L but not in R, so that the matched methods are deleted all together). Note that by deleting these methods we do not lose any functionality, as the domain logic remains in the methods of the remote classes which are preserved. In place of the facade, the rule creates a data transfer object class (labelled dto), with copies of a subset of the attributes in the remote classes (variable region `atts`). The name of the created attributes is built by concatenating the names of the remote class and the remote attribute, to avoid duplicate attribute names in the data transfer object class.
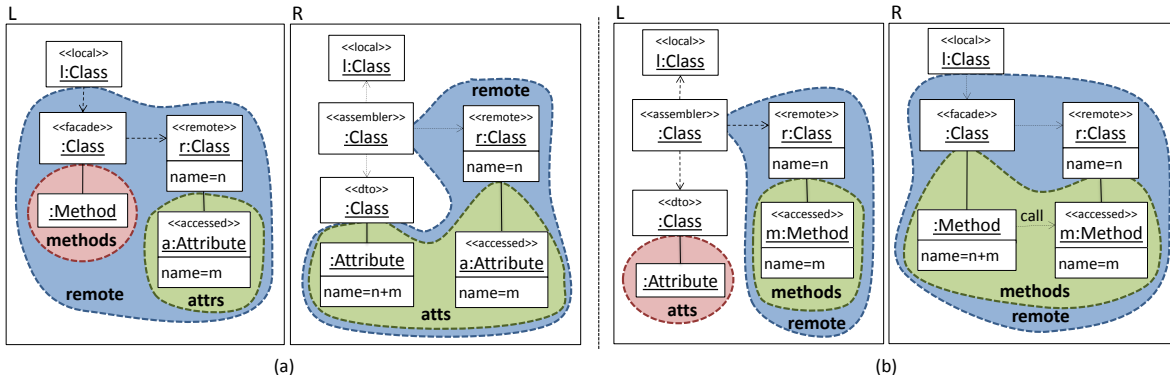

Figure 37.   Some rules for architectural reconfigurations: (a) from *Remote Facade* to *Data Transfer Object*, (b) from *Data Transfer Object* to *Remote Facade*.

Figure 37(b) shows a rule for the reverse situation: given a local class that is using a dto, the rule deletes the dto and its attributes, and substitutes it by a remote facade. In this case, the *facade* object is created within the *remote* region, so that every *remote* class is assigned a facade.

Note that these two rules do not simply have the source pattern (e.g., *Remote Facade*) as left hand side and the target pattern (e.g., *Data Transfer Object*) as right hand side, but describe a migration strategy. Hence, they need to relate elements that are preserved in both patterns, consider just relevant fragments of the patterns that need to be refactored, and possibly use pattern variants to describe the changes. For example, the rule (a) in Figure 37 uses a variant of the pattern in Figure 35(a) with an extra nested region.

Domain logic patterns express strategies for organizing the domain logic, according to different styles. For example, while the *Transaction Script* pattern advocates the use of procedures handling each request from the presentation, the *Domain Model* pattern proposes the use of objects modelling the domain, which incorporate both behaviour and data. In a sense, the *Domain pattern* is opposed to the *Transaction Script*, because the former really promotes an object-oriented conceptualization of the domain. However, a problem with the domain logic pattern is that it complicates the definition of the interface with a database. Hence, the *Table Module* pattern can be used, which organises the domain logic with one class per table in the database: a single instance of such classes contains the various procedures manipulating the data [23]. Figure 38 shows a description of both patterns with our formalism.
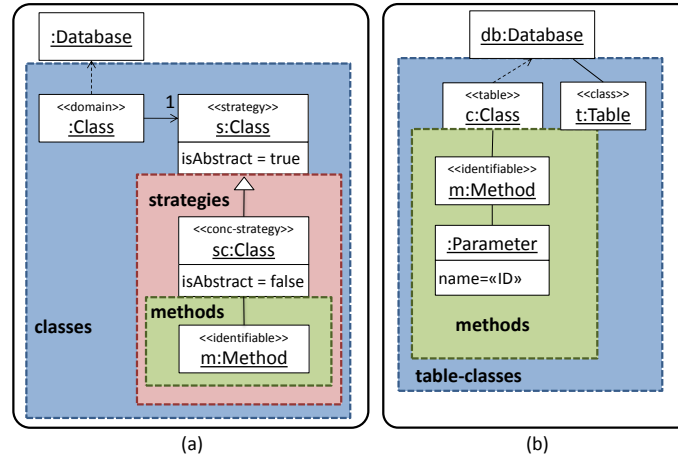

Figure 38.   Some domain logic patterns: (a) Domain model pattern, (b) Table module pattern.

Figure 39(a) shows a pattern rule to migrate from the architectural style *Domain Model* into *Table Module*. Its left hand side looks for *all* classes that conform to the domain model pattern (i.e., they use a strategy class and are annotated as "domain"). For those classes, the rule deletes the strategies they use (both abstract and concrete classes), and transforms the methods in the concrete strategies into methods of the class, thus preserving the logic in the methods. Moreover, the methods are added a new parameter, ID, to be consistent with the *Table Module* pattern. Note that the variable region `strategies` in R is empty because the rule deletes the concrete strategies, but we maintain the region to preserve the tree morphism from K (not shown in the figure) to R. Also, the rule renames the methods to avoid duplication, by adding the name of the concrete strategy initially defining the method as prefix to the method name. This is possible even if the concrete strategy is not present in R, as m is a variable which gets initialised in L with the name of the concrete strategy, and is available in R.

Figure 39(b) shows a rule implementing the reverse reconfiguration: from table module to domain model. The rule matches all table-classes (variable region `classes`), and removes the ID parameter from
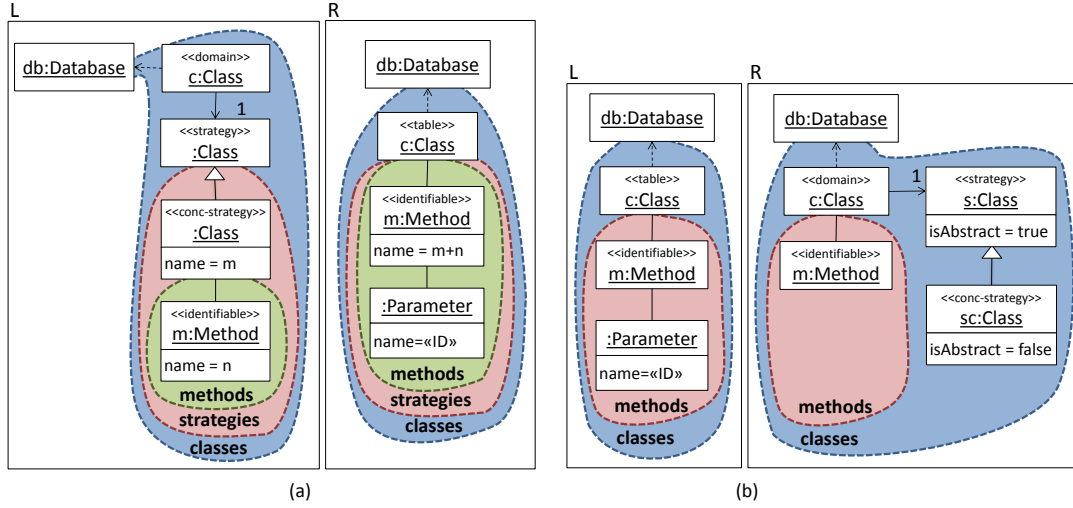
Figure 39.   Some rules for architectural reconfigurations: (a) from *Domain Model* to *Table Module*, (b) from *Table Module* to *Domain Model*.

their methods (variable region `methods`) as they were required by the table module pattern, but not by the domain model pattern that is being applied. Instead, the class is associated with a new strategy.

7.2.   Alternating between variants for design patterns

In this section we capitalize on the notion of pattern variant, applied to object oriented design patterns. As previously stated, some design patterns can occur in variants, the most notable being those based on distinguishing between *inheritance* and *delegation*. Figure 40 illustrates the two meta-model representations of the inheritance- and delegation-based variants of the `Adapter` pattern. In both cases, the pattern is defined only by its root part.
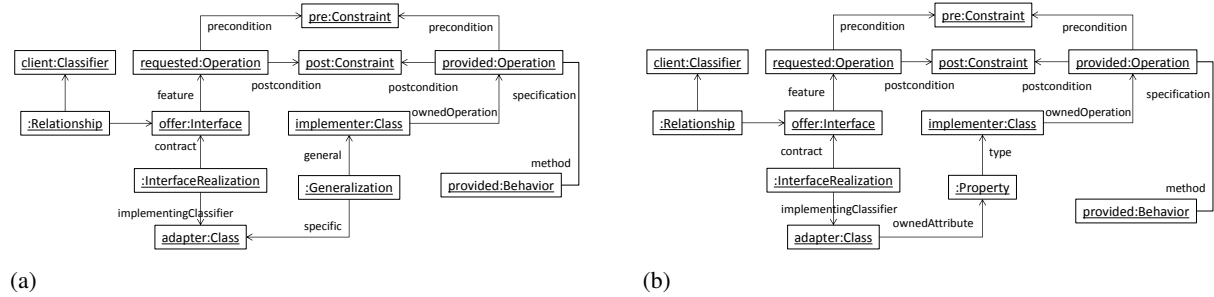

Figure 40.   The two variants for the `Adapter` pattern: inheritance (a) and delegation (b).

As stated in Section 6, variants can be created either by applying pattern rules to a given pattern, or by increasing its variability. Figure 41 represents the two rules (one inverse of the other) to transform one variant into the other. In this case the rules can be automatically obtained: each rule is the minimal rule derived from the span $G \leftarrow D \rightarrow H$, following the construction in [6], where the two whole patterns

play the role of G and H, exchanging them depending on the direction of the transformation. In this case, since the semantics of the left and right hand sides is correct within the language of patterns, we do not need any additional context. Because of the localisation of the transformation, the two rules have the same interface graph K, while the other two graphs play the role of L or R, depending on the direction of the transformation. Note that when applied to the patterns of Figure 40, the class identified by 2 will play the role of the *implementer*, while the one identified by 1 the role of the *adapter*.
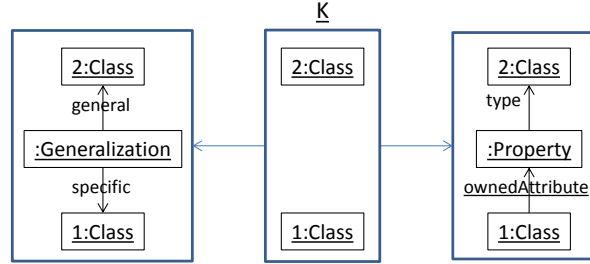

Figure 41.   A representation of the two rules for going from one variant to the other.

Consider now the two variants of the `Adapter` pattern, in its inheritance version, shown in Figure 42, where the possibility of adapting the interface of a class to multiple operations, rather than just one is modelled. The variant (a) requires that, for each operation that is offered, a distinct class will provide that operation, while in the variant (b) a single class provides all the needed operations. In both cases, a single adapter class is needed, using multiple inheritance for the first variant. In this case, the variants are constructed by increasing the variability of the pattern in Figure 40(a), using the operation *insert* in Definition 23. Similar variants for the delegation version of the `Adapter` could be constructed.
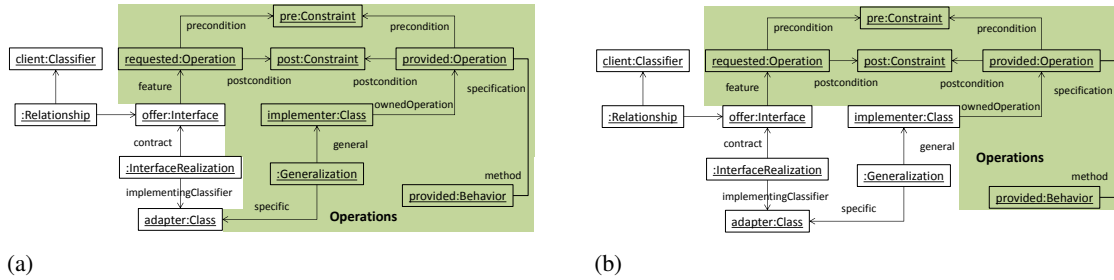

Figure 42.   Variants of the `Adapter` pattern (inheritance version) for multiple operations.

The rules in Figure 41 can be adapted to rules performing the same function, i.e. transforming inheritance into delegation, or vice versa, for the case of the variants for multiple operations. In particular, one obtains a new rule by identifying the highest node in the tree structure where subgraphs of the original pattern are isomorphic to subgraphs in the variant, and placing elements of the rule according to the resulting structure. For the case of a single class implementing all the methods (pattern (b) on Figure 42) the original rules remain unaltered, since it matches to a subgraph which remains in the root of the transformed pattern. For the case of multiple classes (pattern (a) in Figure 42), we obtain the rules in Figure 43, since the node matched by the class identified by 2 is found in the root in all cases, while

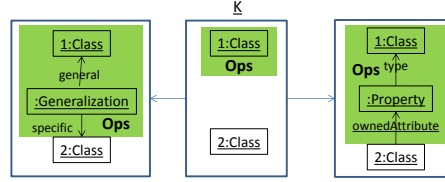all the other elements appear in the `Operations` region in the two variants.

Figure 43.   The variants of the rules of Figure 41 for multi-class adapters.

8.   Related Work

Various meta-models for patterns have been proposed, usually relative to some specific modelling language, typically UML [24, 22]. In [9] we provide a language-independent meta-model for patterns, which can be specialised to any language, through a suitable vocabulary of roles.

Rensink *et al.* have developed a line of research based on expressing constraints on the shape of graphs, thus identifying abstract representations of them [41]. This is aimed at the verification of graph transformation systems, by abstracting them into systems operating on abstract representations. In [5, 42] the authors present some abstractions that can be applied on graphs, while labelling nodes and edge ends with upper bounds, where two graphs are abstracted into the same shape if they have equal local neighbourhoods. A logic is also presented which is preserved and reflected by the abstractions. Similar to our approach, in [7] the authors show that if a graph G can be transformed by a rule, yielding graph H, then G's abstraction can also be transformed by the same rule, yielding H's abstraction.

The use of pattern rules to substitute occurrences of a source pattern with occurrences of a target pattern is analogous to the use of nested rules in [43], where all the occurrences of single rules are applied in an amalgamated way. In the case of [43], however, a single structure accommodates the different rules, whereas pattern rules, having patterns in both their left- and right-hand sides (which may contain nested positive and negative regions), can indicate variations of the overall pattern structure. Since our rules are defined as pattern morphisms, they are more expressive than rules defined by graph morphisms. Moreover, rules can be applied both to concrete models, after abstracting them, and directly to patterns, for example to produce variants.

A notion of pattern and of abstraction close to ours is given in [44], where a pattern is defined as a directed acyclic graph (DAG), with each node in the DAG associated with a graph and each edge in the DAG with a morphisms between the corresponding graphs. The authors adopt a more limited form of pattern morphism than ours, mapping nodes to nodes with isomorphic graphs. They derive a canonical representation for patterns, which is however not preserved by pattern transformations, so that the patterns resulting from a transformation must be normalised back.

In the version without variability equations presented in this paper, pattern trees are a simple form of nested graph constraints, extensively discussed in [28], where their relation with general application conditions on rule application is also discussed. More precisely, pattern trees represent a form of nested constraint where only $\exists$ (for positive regions) and $\nexists$ (for negative regions) operators are used in the formulae associated with each constraint. In this way, one does not have to deal with logical formulae,

as only graph morphisms are used in the definition of the pattern; morphisms can be directly defined between patterns, thus giving rise to a category, which is not the case for general nested constraints. In addition [28] does not present rules made of constraints in left and right hand sides, or abstraction of graphs into constraints. Moreover, the full version of patterns presented in [9] further allows additional positive and negative constraints to be added to individual graphs in the tree.

A mechanism based on graph transformations is that of shapely graph grammars [29], where rewriting has to maintain conformance to some shape, e.g. a chain or a star configuration. Similarly, adaptive star grammars enrich rule schemes with the possibility of cloning, thus allowing the generation of graphs in which some relations between parts are maintained [17]. The relation between pattern trees and these generative mechanisms has to be explored.

Finally, parallel graph transformations [46] enable the definition of complex rules made of an interaction scheme, rules, and subrules. Complex rules can be applied to a standard graph, enabling the identification of all occurrences of some subrule. While this is somewhat similar to our pattern rules, patterns may include negative regions. Moreover, we define the semantics of a pattern by means of a language, and define compatibility conditions for the rewriting of patterns and standard graphs.

Formalisations of patterns based on textual, logical representations have also been proposed. In this line, Zhu and Bailey have recently proposed a number of operators on specification of design patterns given in algebraic form. These operators typically restrict or extend the variability of parts of the pattern, add further constraints or compose different patterns [51]. They thus define a set of algebraic laws for these operators and use them to reduce patterns to canonical forms, which can be checked for equivalence through logical inference. While the possibility of reducing to a canonical form patterns expressed in our language is still under investigation, our definition of pattern morphism allows us to reason about the preservation of some properties of the transformed patterns without resorting to external inference mechanisms. As hinted at in Section 2, logical approaches require modellers to master different formalisms than those typically used in MDD, and do not lend themselves well to the expression of rules taking pattern specifications in their left- and right-hand sides. For a more thorough discussion on the differences between a graph-based and a logical-based approach to pattern definition, we direct the reader to [9].

A distinct, but related, notion of pattern language has been originally introduced in the area of formal (string) languages, to synthetically describe the (non context-free) structure of sentences in a language through the use of variables, where the presence of several occurrences of the same variable in a pattern expression indicates that each occurrence must be instantiated with an identical substring in a string belonging to the language [2]. The field has developed along several lines, considering expressive power, ambiguity (when a string can be decomposed in different ways with respect to the same pattern), decidability, complexity of analysis and formal devices for generation and recognition of languages based on patterns (see e.g. [30, 38]). Languages defined by different composition of patterns have been studied (see e.g. [34]) and hierarchies deriving from allowing operations on variables have been identified in [10]. These results might be adapted to the definition of languages formed by paths on labelled graphs, where patterns might be used to define path expressions. In this line, Santini has recently proposed the use of variables in regular expressions to extract sub-paths in queries on labelled graphs [45].

In formal languages, pattern expressions rely on the linear structure of the string to constrain the position in which an occurrence can happen. Barred ambiguity, this simplifies the problem with respect to graph or tree structures, where the problem of finding repeated occurrences of identical subgraphs within a graph would be at least as hard as subgraph-isomorphism [11]. Attempts to constrain the development of different parts of the graph to achieve similar effects have been proposed by imposing hierarchical

structures on graphs, e.g. with Distributed Graphs [47], where a node at the *network level* is associated with a whole graph at the *local level*, Hierarchical Graphs [18], where specific types of hyperedges contain entire graphs, and multi-level graphs [37], where some nodes hide parts of a graph at some level of abstraction. In all these cases some further constraint could be added on the high-level structure to limit the possible instantiations of the low-level structure. Note that in these two-level mechanisms the form of the high-level graph is also subject to manipulation and is important in defining the overall resulting graph. Pattern trees, on the contrary, are only a way to represent a diagram defined by a collection of nested morphisms, and do not really provide a superior level with respect to the object graph level.

The notion of pattern presented in this paper could be lifted to define an abstract algebra of hierarchical graphs, as presented in [12, 13]. In this case, the overall structure of a graph can be specified as a term over this algebra. Specific graph languages can then be represented in the algebra by establishing a mapping between language and algebra constructors. This would allow us to extend pattern rules across different meta-models, by giving an abstract notion of "shape change" which could be concretised on source and target models. This would also open the way to setting an abstract logic on patterns, based on institutions [26, 14], where satisfaction of a source pattern in a source model would entail satisfaction of the target pattern in the target model. As we are currently dealing with models and patterns built on single meta-models, we leave this development to future work.

9. Conclusions

In this paper, we have presented a formalization of patterns over graphs, intended to allow their integration in model-driven approaches, and defined rewriting rules with such patterns in left- and right-hand sides. An abstraction operation for graphs permits an abstract form of rewriting for classes of graphs. We have shown how to define pattern variants, by either defining pattern rules, or changing the variability offered by the pattern. We have shown the applicability of the approach in the area of enterprise architecture, by defining rules for refactorings of architectures towards a design pattern, and for architecture reconfiguration. We have illustrated the notion of pattern variant with object oriented design patterns.

In the future, we will extend this formalization for richer forms of patterns, where the variability of the regions is governed by equations, as in [9]. We are also working on other strategies for abstraction, e.g., taking into account several branches of the tree at the same time, and for the construction of pattern variants. We would also like to check the feasibility of learning abstractions from a set of (plain) graphs sharing commonalities. That is, to derive a pattern PT from a representative subset of its entailed language $L(PT)$. It would also be interesting to explore other less standard ways of rewriting, but that would provide some more flexibility. For example, deleting by constructing a pullback, e.g., so that a final pullback complement is constructed [15] (and not a pushout complement) would probably weaken conditions for rewriting (especially condition 3 in Theorem 5.5). Finally, we would also like to provide tool support for the presented concepts.

References

[1] C. Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[2] D. Angluin. Finding patterns common to a set of strings. *J. Comput. Syst. Sci.*, 21(1):46–62, 1980.

[3] T. Arendt and G. Taentzer. Integration of smells and refactorings within the eclipse modeling framework. In *Proc. WRT'12*, pages 8–15. ACM, 2012.

[4] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based modeling and refinement of service-oriented architectures. a graph transformation-based approach. *Journal on Software and Systems Modeling*, 5(2):187–207, 2005.

[5] J. Bauer, I. Boneva, M. E. Kurbán, and A. Rensink. A modal-logic based graph abstraction. In *ICGT'08*, volume 5214 of *LNCS*, pages 321–335. Springer, 2008.

[6] D. Bisztray, R. Heckel, and H. Ehrig. Verification of architectural refactorings: Rule extraction and tool support. *ECEASST*, 16, 2008.

[7] I. B. Boneva, A. Rensink, M. E. Kurban, and J. Bauer. Graph abstraction and abstract graph transformation. Technical Report TR-CTIT-07-50, Centre for Telematics and Information Technology University of Twente, Enschede, July 2007.

[8] P. Bottoni, E. Guerra, and J. de Lara. Formal foundation for pattern-based modelling. In M. Chechik and M. Wirsing, editors, *FASE'09*, volume 5503 of *LNCS*, pages 278–293. Springer, 2009.

[9] P. Bottoni, E. Guerra, and J. de Lara. A language-independent and formal approach to pattern-based modelling with support for composition and analysis. *Information & Software Technology*, 52(8):821–844, 2010.

[10] P. Bottoni, A. Labella, and G. Paun. Chomsky hierarchies of pattern languages. *Annals of the Bucharest University*, 47:27–34, 1998.

[11] P. Bottoni and F. Parisi Presicce. Patterns on graphs. In *GRATRA 2000 Workshop*, pages 180–188. Tech. Rep. 2000-02 Fachbereich Informatik, Tech.Univ. Berlin, 2000.

[12] R. Bruni, F. Gadducci, and A. Lluch-Lafuente. An algebra of hierarchical graphs. In *Proc. TGC 2010*, volume 6084 of *LNCS*, pages 205–221. Springer, 2010.

[13] R. Bruni, F. Gadducci, and A. Lluch-Lafuente. An algebra of hierarchical graphs and its application to structural encoding. *Sci. Ann. Comp. Sci.*, 20:53–96, 2010.

[14] A. Corradini, F. Gadducci, and L. Ribeiro. An institution for graph transformation. In *Proc. WADT 2010*, volume 7137 of *LNCS*, pages 160–174. Springer, 2012.

[15] A. Corradini, T. Heindel, F. Hermann, and B. König. Sesqui-pushout rewriting. In *ICGT'06*, volume 4178 of *LNCS*, pages 30–45. Springer, 2006.

[16] B. Courcelle. *The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic.*, volume 1, pages 313–400. World Scientific, 1997.

[17] F. Drewes, B. Hoffmann, D. Janssens, M. Minas, and N. V. Eetvelde. Adaptive star grammars. In *ICGT'06*, volume 4178 of *LNCS*, pages 77–91. Springer, 2006.

[18] F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *J. Comput. Syst. Sci.*, 64(2):249–283, 2002.

[19] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, and S. Varró-Gyapay. Termination criteria for model transformation. In *FASE'05*, volume 3442 of *LNCS*, pages 49–63. Springer, 2005.

[20] H. Ehrig, K. Ehrig, A. Habel, and K.-H. Pennemann. Theory of constraints and application conditions: From graphs to high-level structures. *Fundam. Inform.*, 74(1):135–166, 2006.

[21] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[22] M. Elaasar, L. C. Briand, and Y. Labiche. A metamodeling approach to pattern specification. In *MoDELS 2006*, number 4199 in LNCS, pages 484–498, 2006.

[23] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 1edition, 2002.

[24] R. B. France, D.-K. Kim, S. Ghosh, and E. Song. A UML-based pattern specification technique. *IEEE Trans. Soft. Eng.*, 30(3):193–206, 2004.

[25] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[26] J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[27] A. Habel and K.-H. Pennemann. Nested constraints and application conditions for high-level structures. In *Formal Methods in Software and Systems Modeling*, volume 3393 of *LNCS*, pages 293–308. Springer, 2005.

[28] A. Habel and K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.

[29] B. Hoffmann. Shapely hierarchical graph transformation. In *HCC'01*, pages 30–37. IEEE Computer Society, 2001.

[30] L. Kari, A. Mateescu, G. Paun, and A. Salomaa. Multi-pattern languages. *Theor. Comput. Sci.*, 141(1&2):253–268, 1995.

[31] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE CS, 2008.

[32] S. Mac Lane. *Categories for the Working Mathematician. 2nd Edition. Graduate Texts in Mathematics Vol 5*. Springer, 1998.

[33] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Trans. Software Eng.*, 30(2):126–139, 2004.

[34] V. Mitrana, G. Paun, G. Rozenberg, and A. Salomaa. Pattern systems. *Theor. Comput. Sci.*, 154(2):183–201, 1996.

[35] OMG. MDA home page. `http://www.omg.org/mda/`, 2009.

[36] F. Orejas, H. Ehrig, and U. Prange. Reasoning with graph constraints. *Formal Asp. Comput.*, 22(3-4):385–422, 2010.

[37] F. Parisi-Presicce and G. Piersanti. Multilevel graph grammars. In *WG'94*, volume 903 of *Lecture Notes in Computer Science*, pages 51–64. Springer, 1995.

[38] G. Paun, G. Rozenberg, and A. Salomaa. Pattern grammars. *Journal of Automata, Languages and Combinatorics*, 1(3):219–242, 1996.

[39] D. Plump. Termination of graph rewriting is undecidable. *Fundam. Inform.*, 33(2):201–209, 1998.

[40] U. Prange, H. Ehrig, and L. Lambers. Construction and properties of adhesive and weak adhesive high-level replacement categories. *Applied Categorical Structures*, 16(3):365–388, 2008.

[41] A. Rensink. Canonical graph shapes. In *ESOP'04*, volume 2986 of *LNCS*, pages 401–415. Springer, 2004.

[42] A. Rensink and D. Distefano. Abstract graph transformation. *Electr. Notes Theor. Comput. Sci.*, 157(1):39–59, 2006.

[43] A. Rensink and J. Kuperus. Repotting the geraniums: On nested graph transformation rules. *ECEASST*, 18, 2009.

[44] A. Rensink and E. Zambon. Pattern-based graph abstraction. In *ICGT'12*, volume 7562 of *LNCS*, pages 66–80. Springer, 2012.

[45] S. Santini. Regular languages with variables on graphs. *Inf. Comput.*, 211:1–28, 2012.

[46] G. Taentzer. *Parallel and distributed graph transformation - formal description and application to communication-based systems*. Berichte aus der Informatik. Shaker, 1996.

[47] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *TAGT'78*, volume 1764 of *LNCS*, pages 179–193, 1998.

[48] D. Tamzalit and T. Mens. Guiding architectural restructuring through architectural styles. In *ECBS'10*, pages 69–78, 2010.

[49] M. Völter and T. Stahl. *Model-driven software development*. Wiley, 2006.

[50] I. Warren and J. Ransom. Renaissance: A method to support software system evolution. In *COMPSAC'02*, pages 415–420. IEEE Computer Society, 2002.

[51] H. Zhu and I. Bayley. An algebra of design patterns. *ACM Trans. Softw. Eng. Methodol.*, 22(3):23, 2013.

Appendix

This appendix presents the details concerning the proofs of the different theorems of the paper.

First of all, we discuss satisfiability of pattern trees. There are pattern trees that only admit infinite graphs in their semantics, as for example the one in Figure 44. This example follows the same idea as the one presented in [36]. The pattern models the class of graphs made of chains of infinite length. This is so, as it requires at least one node (region a), where each node has no multiple outgoing (region b) or incoming edges (region c), there is at least one node with no predecessor node (regions d and d'), and each node has exactly one successor node (regions e and e').
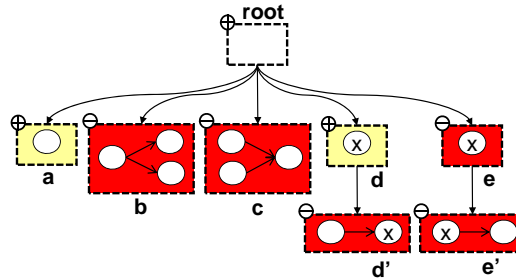

Figure 44.   A pattern that only admits infinite graphs as models.

Theorem 3.2 (Satisfaction of 1-negation pattern trees). *Satisfaction is decidable for 1-negation pattern trees*.

Proof:

(Sketch) Let $PT = (V, E, root, G, Mor, \sigma, \gamma_V, \gamma_E)$, and PT^+ be the biggest subtree of PT, rooted in *root*, where every node of PT^+ has positive sign. More precisely $PT^+ = (V^+, E^+, root, G^+, Mor^+, \sigma|_{V^+}, \gamma_V|_{V^+}, \gamma_E|_{E^+})$, with:

- $V^+ = \{v \in V \mid \forall v_i \in path(v) \, [\sigma(v_i) = \oplus]\} \subseteq V$,

- $E^+ = \{(v_i, v_j) \in E \mid \{v_i, v_j\} \subseteq V^+\} \subseteq E$,

- $G^+ = \{\gamma_V(v_i) \mid v_i \in V^+\} \subseteq G$,

- $Mor^+ = \{G_i \xrightarrow{m} G_j \in Mor \mid \{G_i, G_j\} \subseteq G^+\} \subseteq Mor$,

Let C be the colimit of all graphs and graph morphisms of PT^+ (i.e., in G^+ and Mor^+). Clearly, C contains all necessary nodes and edges requested by the pattern, as satisfaction demands at least one occurrence of each positive variable region of PT (which is given by the colimit). Let $M^+ = \{M \mid \exists m\colon C \to M \text{ with } m \text{ surjective}\}$. The size of M^+ is finite, as there is only a finite number of ways for "folding" M in C, i.e., a finite number of non-isomorphic graphs M and surjective morphisms m.

PT is satisfiable *iff* $\exists M \in M^+$ s.t. $M \models PT$. The "if" part is obvious. For the "only if" part, if $\nexists M \in M^+$ s.t. $M \models PT$, then it means that there is an occurrence of some negative region of PT in every graph of M^+. Hence, any other graph M' bigger than those in M^+ will also have an occurrence of such a negative region. Therefore no graph can satisfy PT, which is then unsatisfiable. $\qquad\square$

Theorem 3.3 (Satisfaction of 2-negation pattern trees). *Satisfaction is decidable for 2-negation pattern trees without negative region dependencies.*

Proof:

(Sketch) We transform this satisfiability problem to that of termination of a particular kind of graph transformation system, resulting from the transformation of the pattern. Termination is undecidable for general graph transformation systems [39], but some classes of terminating transformation systems have been characterized [19]. In particular, for a transformation system TS with non-deleting rules only, in [19] it is shown that TS terminates if each rule has a NAC which is included in the right-hand side of the rule and the rule does not create any element with a type that is in the left-hand side of another rule. A nested negative region $e = (v_i, v_j)$ is equivalent to a set of graph transformations rules R_{ij} given by $\{\langle L_k \to R = \gamma_V(v_j), n\colon L_k \to R\rangle\}$ where:

- The left hand sides are all graphs L_k s.t. $\gamma_V(v_i) \hookrightarrow L_K \hookrightarrow \gamma_V(v_j)$, with $L_K \not\cong \gamma_V(v_j)$.

- The right-hand side R of every rule is $\gamma_V(v_j)$.

- n is a NAC.

Let R_{PT} be the set of all the rules generated from PT's nested negative regions. If the pattern does not have negative region dependencies, the resulting graph transformation system satisfies the termination criterion of [19] and hence it terminates.

We construct the set $N = \{H_i \mid G \Rightarrow^*_{R_{PT}} H_i \wedge G \in M^+\}$, with M^+ constructed as in the proof of Theorem 3.2. PT is satisfiable *iff* $\exists G \in N$ s.t. $G \models PT$. The "if" part is obvious. For the "only

if" part, first note that each H_i has all elements required by the positive part of PT (as rules in R_{PT} are non-deleting). Then, some of the H_i satisfy the conditions given by the nested regions, because some of the derivations apply the rules to all matches. If $\not\exists M \in M^+ \models PT$, it is because there is an occurrence of some negative region of PT in every graph of N. Hence, any other graph H' bigger than those in N will also have an occurrence of such negative region. Therefore no graph can satisfy PT, which is then unsatisfiable. Please note that, as we have 2-negation pattern trees, if some negative region is not satisfied in some graph H_i, it cannot be made satisfied by further application of the rules in the sets R_{ij}, and hence at most one application of the rules in each set R_{ij} is enough. $\qquad\square$

Theorem 3.4 (SPT-Morphism Semantics).

Proof:
(Sketch) Given an SPT-morphism $m = (m_V, m_E, m_F)\colon PT_1 \to PT_2$ and an arbitrary graph $M_1 \in L(PT_1)$, we construct the graph M_2 and the morphism $\lfloor m_{12} \rfloor\colon M_1 \to M_2$ as shown next.

First, we define the expansion set of PT, as the set of all patterns resulting from duplicating any subtree of PT an arbitrary number of times. More precisely:

$$
\begin{aligned}
EXP_0(PT) &= \{PT\} & (3)\\
EXP_{n+1}(PT) &= \{concat(PT_i, Subtree(v), \gamma_E((r, v))) \mid PT_i \in EXP_n(PT) \wedge (r, v) \in E\} & (4)\\
EXP(PT) &= \bigcup_{n=0..\infty} EXP_n(PT) & (5)
\end{aligned}
$$

Let $D = \langle G, Mor \rangle$ be the diagram made of the graphs and graph morphisms of PT. Given any $PT_i \in EXP(PT)$, with diagram $D_i = \langle G_i, Mor_i \rangle$ we can build a family of identity graph morphisms $RMor_i$, where there is one morphism from $\gamma_V(v) \in G$ to each replica $\gamma_V^i(v_j)$ of $\gamma_V(v)$ in PT_i.

We now take a pattern $PT_1' \in EXP(PT)$ s.t. (1) $M_1 \models PT_1'$, and (2) there is at most one morphism in S_{PT_1', M_1}, from each graph G_i in PT_1' i.e. we have chosen an expansion of PT_1 replicating each $Subtree(n_i^1)$ with $n_i^1 \in V_1$ as many times as different morphisms $m_i^j\colon \gamma_V(n_i^1) \to M_1$ are in S_{PT_1, M_1}.

We now choose a pattern $PT_2' \in EXP(PT_2)$ where each $Subtree(n_i^2)$ is replicated as many times as $m_i^j\colon \gamma_V(n_i) \to M_1$ are in S_{PT_1, M_1} with $m(n_i^1) = n_i^2$. As subtrees in PT_1 and PT_2 have been replicated the same number of times, there is an SPT-morphism $m'\colon PT_1' \to PT_2'$. At this stage, there may be several patterns PT_2' that can be chosen, resulting from an arbitrary expansion of the subtrees of PT_2 not mapped from PT_1.

Figure 45 depicts the construction, where D_1' is the diagram made of the graphs and morphisms of PT_1', where we remove the graphs $\gamma_V^1(v_i^1)$ for which there is no morphism $\gamma_V^1(v_i^1) \to M_1$. Similarly, D_2' is the diagram made of the graphs and morphisms of PT_2', where we remove the graphs $\gamma_V^2(v_i^2)$ s.t. $m(v_i^1) = v_i^2$ and $\gamma_V^1(v_i^1)$ was not included in D_1. Graph M_2 is calculated as the colimit of $M_1 \Leftarrow D_1' \Rightarrow D_2'$. Then, we have that $M_2 \in L(PT_2)$, because we can build the set of morphisms S_{PT_2, M_2} by composing the morphisms in S_{PT_2, M_2}' with those of $RMor_2$, and the morphisms in S_{PT_2, M_2}' are jointly surjective, because, due to the colimit construction, so are those of S_{PT_2, M_2}. $\qquad\square$

Remark 9.1. In the previous proof sketch, we cannot directly build M_2 from the colimit of $M_1 \Leftarrow D_1 \overset{m_F}{\Rightarrow} D_2$, because the different occurrences of the graphs of PT_1 in M_1 would be folded in M_2.
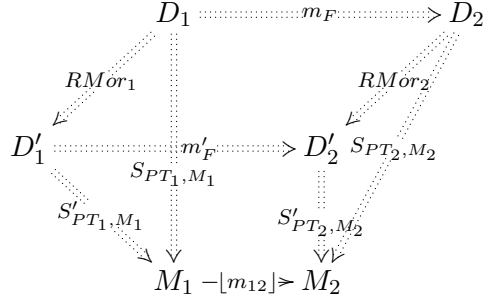
Figure 45.   Building $\lfloor m_{12} \rfloor$.

Theorem 3.5 (Pattern Tree Categories).

Proof:
(Sketch) We first show that **PattTree** $= (\mathbf{PT}, Hom(\mathbf{PT}))$ is a category. We have to show: (1) how morphism composition is performed, (2) the existence of the identity morphism, and (3) associativity.

- **Composition** Given two PT-morphisms $f\colon PT_1 \to PT_2$ and $g\colon PT_2 \to PT_3$, their composition is given by the composition of the set functions and the corresponding graph morphisms: $g \circ f = (g_V \circ f_V, g_E \circ f_E, \{g_G^{f_V(i)} \circ f_G^i\colon \gamma_V^1(v_i^1) \to \gamma_V^3(g_V \circ f_V(v_i^1)) \mid v_i^1 \in V^1\})$.

- **Identity Morphism**. Given $PT \in \mathbf{PT}$, there is an identity morphism $id_{PT}\colon PT \to PT$ for PT, s.t. given any $f\colon PT_1 \to PT_2$, we have $id_{PT_2} \circ f = f = f \circ id_{PT_2}$. The identity morphism can be constructed as $id_{PT} = (id_V, id_E, id_F = \{id_G^i\colon \gamma_V(v_i) \to \gamma_V(v_i) \mid v_i \in V\})$. Given any $f\colon PT_1 \to PT_2$, by composition of set and graph functions, we have $id_{PT_2} \circ f = (id_{V_2}, id_{E_2}, id_{F_2} = \{id_G^i\colon \gamma_V^2(v_i) \to \gamma_V^2(v_i) \mid v_i \in V_2\}) \circ (f_V\colon V_1 \to V_2, f_E\colon E_1 \to E_2, f_F = \{f_G^i\colon \gamma_V^1(v_i^1) \to \gamma_V^2(f_V(v_i^1)) \mid v_i^1 \in V_1\})) = (id_{V_2} \circ f_V\colon V_1 \to V_2, id_{E_2} \circ f_E\colon E_1 \to E_2, \{id_G^i \circ f_G^i\colon \gamma_V^1(v_i^1) \to \gamma_V^2(f_V(v_i^1)) \mid v_i^1 \in V_1\}) = (f_V\colon V_1 \to V_2, f_E\colon E_1 \to E_2, f_F = \{f_G^i\colon \gamma_V^1(v_i^1) \to \gamma_V^2(f_V(v_i^1)) \mid v_i^1 \in V_1\}) = f$. The right-composability of set and graph identity morphisms leads to $f = f \circ id_{PT_2}$.

- **Associativity**. Given three PT-morphisms $f\colon PT_1 \to PT_2$, $g\colon PT_2 \to PT_3$ and $h\colon PT_3 \to PT_4$ we have to show associativity: $h \circ (g \circ f) = (h \circ g) \circ f$. Hence, we have $h \circ (g \circ f) = (h_V \circ (g_V \circ f_V))\colon V_1 \to V_4, h_E \circ (g_E \circ f_E)\colon E_1 \to E_4, \{h_G^i \circ (g_G^i \circ f_G^i)\colon \gamma_V^1(v_i^1) \to \gamma_V^4(h_V \circ (g_V \circ f_V)(v_i^1)) \mid v_i^1 \in V_1\})$, which by associativity of set and graph functions is equal to $(h \circ g) \circ f = ((h_V \circ g_V) \circ f_V\colon V_1 \to V_4, (h_E \circ g_E) \circ f_E\colon E_1 \to E_4, \{(h_G^i \circ g_G^i) \circ f_G^i\colon \gamma_V^1(v_i^1) \to \gamma_V^4((h_V \circ g_V) \circ f_V(v_i^1)) \mid v_i^1 \in V_1\})$,

It is easy to show that **SPattTree** is a subcategory of **PattTree**, with all satisfiable patterns as objects and SPT-morphisms as arrows. Hence, the inclusion functor $I\colon \mathbf{SPattTree} \to \mathbf{PattTree}$ is injective on objects and faithful (injective on morphisms), but it is not full, as some PT-morphisms are not SPT-morphisms. Please note that the identity morphism is an SPT-morphism.   □

Theorem 5.1 (Pushouts for Pattern Trees)

Proof:

(Sketch)

In order to show that the construction in Definition 18 yields a pushout, we have to show: (a) that the square (1) to the left of Figure 46 commutes, and (b) that for each PT' s.t. $g' \circ f = f' \circ g$, $\exists ! u \colon PT \to PT'$ making the diagram in the left of the figure commute.

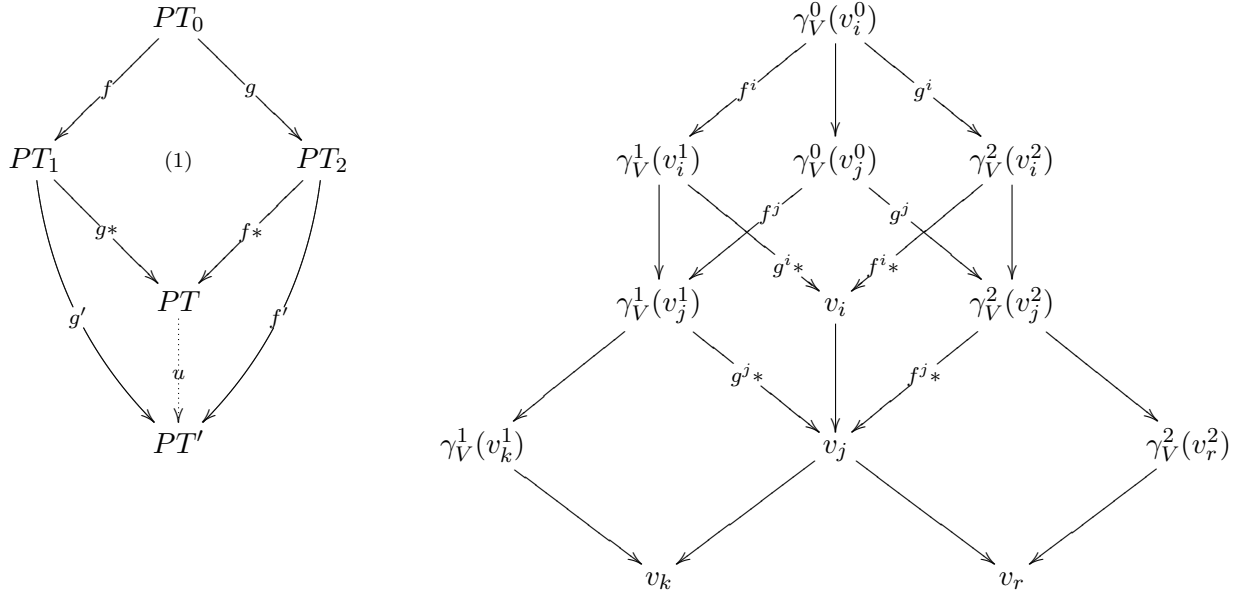Figure 46.   Pushouts for Pattern Trees (left). Detailed construction (right).

(a) It is easy to see that (1) commutes, because the *glue* function creates a pattern tree PT resulting from the pushout of every graph span $\gamma_V^1(v_i^1) \leftarrow \gamma_V^0(v_i^0) \to \gamma_V^2(v_i^2)$. Moreover, at each stage of the procedure (at each level of the traversal of PT_0), we fuse the subtrees resulting from the *merge* operation on those graphs that are not in $Cod(f)$ and $Cod(g)$, see the right of Figure 46. This means that, every graph in PT comes either from a pushout of $\gamma_V^1(v_j^1) \leftarrow \gamma_V^0(v_j^0) \to \gamma_V^2(v_j^2)$, or a pushout of $\gamma_V^1(v_k^1) \leftarrow \gamma_V^1(v_j^1) \to v_j$, where v_j is a graph of PT, and $v_k^1 \notin Cod(f)$, or a pushout of $v_j \leftarrow \gamma_V^2(v_j^2) \to \gamma_V^2(v_r^2)$, where v_j is a graph of PT, and $v_r^2 \notin Cod(g)$ (see the right of Figure 46).

(b) Assume there is a PT', $g' \colon PT_1 \to PT'$ and $f' \colon PT_2 \to PT'$ such that $g' \circ f = f' \circ g$. Assume that PT' has the same regions as PT. Then, the existence and uniqueness of u follows from the fact that every graph in PT has been built by a pushout. If PT' has more regions than PT, there is a morphism from PT, where those extra regions do not belong to $Cod(u)$. $\qquad\square$

Theorem 5.3 (Pushout compatibility)

Proof:

(Sketch) In order to show $G \in L(PT)$, we construct the necessary morphisms in $S_{PT,G}$. The proof proceeds, with reference to Figure 47, by cases on the variable regions of the pushout pattern tree PT.

1. In the first case, we consider regions $v_i \in n_1^i(PT_V^1) \cap n_2^i(PT_V^2)$. As the left of Figure 47 shows, for every occurrence f_i^1, f_i^0, f_i^2, we obtain a unique $f_i : \gamma_V(v^i) \to G$. This is so as the top and bottom squares are pushouts, and by the pushout universal property we obtain the unique f_i.

2. In the second case, we consider regions $v_j \in n_1^j(PT_V^1)$, and $v_i \notin n_2^j(PT_V^2)$. These are regions added to PT due to its presence only in PT_1 (the case for regions coming from PT_2 is analogous). The right of Figure 47 shows the situation, where the top squares are pushouts, and f_j exists due to the universal pushout property.
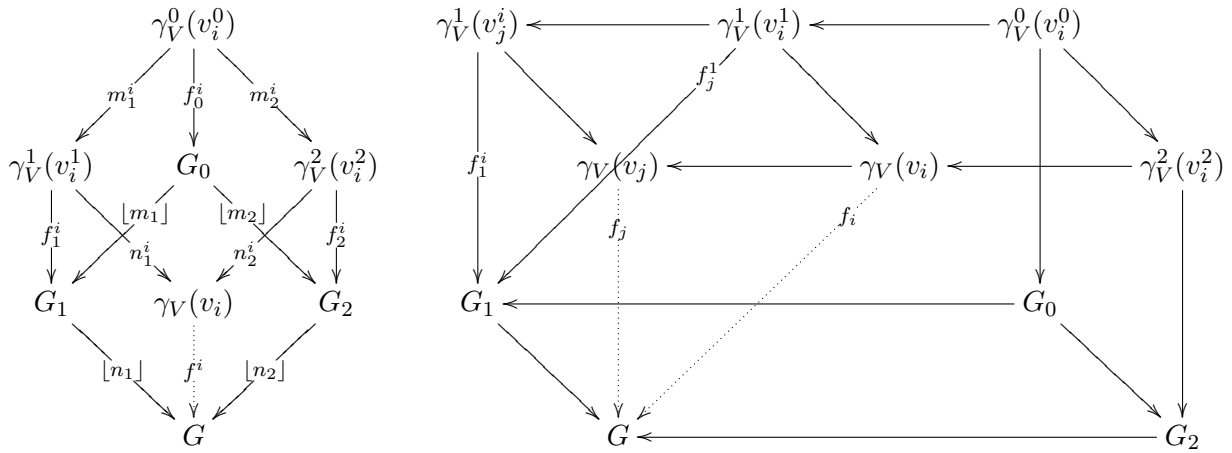

Figure 47.   Case 1 in compatibility condition (left). Case 2 in compatibility condition (right).

Finally, we have to show that the set $S_{PT,G}$ is jointly surjective. This property holds, because $\lfloor m_i \rfloor$ (for i=1,2) are not arbitrary SPT-morphisms, but $\lfloor m_i \rfloor \in \llbracket m_i \rrbracket$. This means that the morphisms $\lfloor m_i \rfloor$ have been built as described in the proof of Theorem 3.4 (see Figure 45), and so each occurrence of any $\gamma_V^0(v_j^0)$ in G_0 is coordinated with an occurrence of $\gamma_V^i(v_j^i)$ in G_i. In particular, this means that there cannot be more occurrences of $\gamma_V^i(v_j^i)$ in any G_i than occurrences of $\gamma_V^0(v_j^0)$ in G_0 (see the colimit construction of Theorem 45). As these extra occurrences are the only possible cause for $S_{PT,G}$ of not being jointly surjective, we can conclude that $S_{PT,G}$ is jointly surjective, and hence $G \in L(PT)$.   □

Theorem 5.5 (Pushouts complements)

Proof:
(Sketch) With reference to Figure 48, in order for D to be a pushout object, we need (a) that D seen as a tree (abstracting from the fact that it contains graphs associated to every node of it) is a pushout complement, and (b) that every graph in D is actually a pushout complement with the right conditions.

(a) By condition 2, there are no "dangling regions". This means that, if we look at K, L, D and M as trees (by abstracting from the fact that they also have graphs and morphisms), and take l and m as tree morphisms, we can find a tree D and tree morphisms d and $l*$ so that D is a pushout complement (because a tree is a particular case of graph).

(b) Next, we have to check every graph of M and ask for specific conditions so that a pushout complement D exists. We first look at the graphs in $m_V \circ l_V(K)$, which are those that should be "merged" from D and L. Those graphs should satisfy the dangling edge condition in graphs. In effect, by condition 1 of the theorem, the dangling edge condition is satisfied in each graph morphism $m_G^i \circ l_G^i$ with $l_G^i \in l_F$ and $m_G^i \in m_F$ (where l_F is the family of graph morphisms of the tree morphism l, and m_F is the family of graph morphisms of the tree morphism m). According to the conditions for the existence of pushout complements in graphs [21], a pushout complement graph exists for every $l_G^i \in l_F$, and so we can build $\gamma_V^K(v_i^K) \to \gamma_V^D(v_i^D) \to \gamma_V^M(v_i^M)$ (see Figure 48).

$$
\begin{array}{ccc}
K & \xrightarrow{\;\;l\;\;} & L \\
\vdots & & \big\downarrow \\
d & \text{P.O.} & m \\
\big\downarrow & & \big\downarrow \\
D & \cdots\!\!\xrightarrow{\;l*\;}\!\!\cdots & M
\end{array}
$$

Figure 48. Pushouts complements for Pattern Trees.

Then, we look at the graphs in $m(GR)$ (with $GR = \{v_i \in V_L \setminus l(V_K)\}$). These are the regions added from L only (not common to D). Condition 3 of the theorem demands that they are constructible by pushouts. Then, the only remaining graphs to consider in M are those in $V_M \setminus m_V(V_L)$. Each of those graphs needs to form a pushout with some graph of D_V, which is exactly condition 4 of the theorem. The main point is that the nodes in $M_D = c(\gamma_V(v_k^L \setminus l_k(\gamma_V^K(v_k^K))))$ are those to be "deleted" from $\gamma_V^M(v_i^M)$, and so every edge in $\gamma_V^M(v_i^M)$ cannot start and end (or vice versa) in a node in M_D and a node out of this set. □

Theorem 5.6 (Pushout complement compatibility)

Proof:
(Sketch) If the pattern tree pushout complement PT exists, then $PT_0 \to PT_1 \to PT_2$ satisfies the conditions of Theorem 5.5. In particular, this means that the dangling edge condition is satisfied in every square produced by the morphism $m_1^*: PT \to PT_2$. Hence, we can build G by deleting from G_2 the necessary elements, as given by $\lfloor m_1 \rfloor: G_0 \to G_1$. No dangling edges can occur in such deletion, because $G_2 \in L(PT_2)$, which means that S_{PT_2,G_2} is jointly surjective, and hence G_2 does not contain extra elements that may cause dangling edges, other than those in PT_2. Now, $S_{PT,G}$ can be constructed, since we maintained in G the occurrences of the regions of PT_2 in G_2. □