A language-independent and formal approach to pattern-based modelling with support for composition and analysis

Paolo Bottoni [a], Esther Guerra [b,*], Juan de Lara [c]

[a] *Computer Science Department, "Sapienza" Università di Roma, Italy* [b] *Computer Science Department, Universidad Carlos III de Madrid, Spain* [c] *Polytechnic School, Universidad Autónoma de Madrid, Spain*

*Corresponding author. Present address: Dpto. Informática, Universidad Carlos III de Madrid, Avda. Universidad 30, 28911 Leganés, Madrid, Spain.

E-mail addresses: bottoni@di.uniroma1.it (P. Bottoni), eguerra@inf.uc3m.es (E. Guerra), Juan.deLara@uam.es (J. de Lara).

Abstract:

C*ontext*: Patterns are used in different disciplines as a way to record expert knowledge for problem solv-ing in specific areas. Their systematic use in Software Engineering promotes quality, standardization, reusability and maintainability of software artefacts. The full realisation of their power is however hindered by the lack of a standard formalization of the notion of pattern.

O*bjective*: Our goal is to provide a language-independent formalization of the notion of pattern, so that it allows its application to different modelling languages and tools, as well as generic methods to enable pattern discovery, instantiation, composition, and conflict analysis.

M*ethod*: For this purpose, we present a new visual and formal, language-independent approach to the specification of patterns. The approach is formulated in a general way, based on graphs and category theory, and allows the specification of patterns in terms of (nested) variable submodels, constraints on their allowed variance, and inter-pattern synchronization across several diagrams (e.g. class and sequence dia-grams for UML design patterns).

R*esults*: We provide a formal notion of pattern satisfaction by models and propose mechanisms to sug-gest model transformations so that models become consistent with the patterns. We define methods for pattern composition, and conflict analysis. We illustrate our proposal on UML design patterns, and discuss its generality and applicability on different types of patterns, e.g. workflow patterns, enterprise inte-gration patterns and interaction patterns.

C*onclusion*: The approach has proven to be powerful enough to formalize patterns from different domains, providing methods to analyse conflicts and dependencies that usually are expressed only in tex-tual form. Its language independence makes it suitable for integration in meta-modelling tools and for use in Model-Driven Engineering.

Keywords:

Pattern formalization, Pattern-based modelling, Pattern composition, Pattern conflicts

1. Introduction

Patterns [1] are used in different disciplines as a way to record expert knowledge for problem solving in specific areas. They are increasingly used in Software Engineering for the definition of software applications and frameworks [14], as well as in Model-Driven Engineering to indicate parts of required architectures [2,19], drive code refactorings [21], or build model-to-model transformations [6]. Their systematic use promotes quality, standardization, reusability and maintainability of software artefacts. However, the full realisation of their power is hindered by the lack of a standard formalization of the notion of pattern, as they are typically presented through natural language to explain their motivation, context and consequences; programming code to show usages of the pattern; and example diagrams to communicate their structure and behaviour.

Moreover, the use of domain-specific modelling languages, such as UML for design patterns [14] or coloured Petri nets for workflows [35], forces pattern proposers to provide only examples of their realisation, appealing to intuition to extend them to the complete semantics of the pattern. For instance, the fact that in the Visitor GoF pattern [14] there must be a distinct operation in the Visitor interface for each ConcreteElement is only under-stood through generalisation of the examples or reading the asso-ciated text [14]. Even though natural language and examples are necessary and useful for pattern documentation, the automated use of patterns for modelling requires a formalization of some of their aspects to enable pattern instantiation, pattern identification, pattern composition and analysis of pattern conflicts.

In this sense, several existing formal approaches resort to languages based on mathematics or logics to formalize patterns. This allows a systematic use of patterns and the automation of pattern-based tasks. Unfortunately, defining patterns in these approaches requires expert knowledge of the used underlying formalism, which is rarely found in the average engineer. Furthermore, engineers have to deal with two representations of the same patterns (the intuitive one used in the application domain for communication purposes, and its formal counterpart) and have to guarantee that both representations are consistent. As a result, tools built on top of these formalizations make it difficult to define new patterns and maintain the existing ones. Therefore, more user-friendly formalizations are required.

Moreover, many formal approaches are tied to a specific modelling language, frequently UML, and reusing the procedures developed for them in other languages and tools is not possible. Thus, it is clear that a formalization would benefit from being generic and language-independent, and hence reusable. This independence is especially useful in Model-Driven Engineering, which relies on domain-specific languages for modelling applications, and in meta-modelling tools for generating tools for these languages. Having a generic framework for manipulating patterns for any language would reduce the development time of modelling tools with support for patterns.

In this paper we tackle these problems by proposing a formal notion of pattern, grounded in category theory [25], which sees them as formed of: (i) a vocabulary of roles; (ii) a collection of diagrams defining associations between roles, and supplemented with indications of variable regions and their instantiation constraints; (iii) a collection of interfaces that identify roles across different diagrams; and (iv) a set of positive or negative constraints defining pattern invariants. From this formalization several benefits are obtained, representing a distinguishing novel feature of our approach. First, patterns in some specific domain are formalized by using the notation used in that particular domain. For instance, the formalization of a GoF pattern will use the UML notation, whereas workflow patterns will use coloured Petri nets instead. This is so because we rely on a categorical framework rather than on the special features of particular languages. Second, a clear specification of the parts of the pattern that can be replicated (and of the admissible number of replicas) is given, without recurring to concrete examples. Third, patterns are defined via several inter-related diagrams, e.g. a class and a sequence diagram for GoF patterns, which enables their synchronous instantiation in a consistent manner.

Moreover, we can formalize additional environmental conditions that must hold for the pattern to be correctly applied. These are to be understood as pattern invariants and, up to a point, allow the formalization of some of the intentions and consequences of patterns and permit automatic detection of pattern conflicts. Finally, the formalization allows pattern composition and the creation of new patterns by reusing existing ones, thus enabling the realization of pattern-based languages. Again, all these operations are expressed in a language-independent manner, so that they can be applied across different languages and tools.

This paper builds on our previous work in [7], and extends it with a more expressive description of the allowed instantiations of the variability regions (equations instead of intervals), a representation of pattern invariants, techniques for pattern composition and analysis, and the formalization of patterns for additional modelling languages. We purposely present the results using intuition and examples. However, given that one of the points of our work is that it is a *formalization* of patterns, we have kept the main definitions in an appendix.

Paper organization. Section 2 overviews related approaches. Section 3 reviews our definition of pattern, presenting the new way of handling instantiation constraints of variable regions. Section 4 presents our formalization of pattern invariants. Section 5 shows the procedure for applying patterns to models. As an extension of previous work, this may imply either the addition of elements so that the model conforms to a pattern, or both the deletion and addition of objects so that the pattern invariants are satisfied. Section 6 introduces methods for composing patterns and analyse conflicts. Section 7 presents examples in different modelling languages, and Section 8 ends with the conclusions. After briefly introducing the relevant notions of Category Theory in Appendix A, we gather in Appendix B the formal details of the concepts and definitions used throughout the paper. However, the article can be fully understood without reading them.

2. Related work

The shortcomings of presenting patterns in the Gang of Four (GoF [14]) style have been addressed by several researchers, who advocate a more formal approach. However, most of the formalizations are specific to UML design patterns since they modify UML to incorporate elements of the formalization, and therefore they cannot be extrapolated to other languages.

For example, [13] extends the UML meta-model with roles and constraints, and conformance of a model to a pattern is checked as the usual model/meta-model relationship. The technique works for UML class and sequence diagrams, and is focused on the specification of patterns, not on their use for model completion. A non-uniform interpretation of interaction diagrams is provided, where reference to interaction fragments is translated to their unfolding with respect to the instantiation of roles. In [26] the authors extend the UML meta-model with stereotypes accounting for possible realizations of patterns. A distinction between roles, types and classes is used in [23] to decouple representations of roles from implementation aspects, although they resort to the lower level for defining the admissible realisations of the pattern. In [10] the UML Profile meta-model is extended with stereotypes and tagged values for pattern annotation and visualization. This also allows pattern composition through single instances. Our approach relies on annotations through triple graphs, introducing a new meta-model for patterns, but without modifying existing meta-models. Finally, an approach similar to ours – but specific for UML design patterns – is taken in [27], where a visual language allows representing roles played by participants in patterns, which can then be instantiated by elements of a UML model. The approach provides notions of variability and (limited) support for definition of behavioural (collaboration) diagrams, but without an explicit notion of synchronization between parts of the specification. Although patterns can overlap on some model elements, there is no explicit notion of composition.

There are also other works that introduce the formalization independently of the modelling language. Such independence is achieved in [22] by using Object-Z, but no composition or conflict analysis techniques are given. Constraints on the use of patterns are exploited in [32] to maintain the consistency of a pattern-based software framework through the use of high-level transformations specific to each pattern. In [30], the authors propose a logic-based approach, using subsets of First Order Logic – for structural aspects – and Temporal Logic of Actions – for behavioural ones – and supporting pattern combinations. As the language does not include implications, support for complex constraints appears limited. These approaches have the shortcoming of using a complicated mathematical formalism for formalizing the patterns, different from the domain language in which the pattern has to be applied.

Another axis of related work concerns the different usage scenarios of the formal specifications. The most basic activity is checking the conformance of a model against a pattern. Further activities include model completion, guidance for pattern use, pattern identification (i.e., detecting patterns in models), and pattern analysis. For example, in [20] the intent of design patterns is described with an ontology, which can be queried to obtain suggestions about the most appropriate pattern solving a certain problem. Constraints are used for formalizing patterns in [15], where patterns are considered as paragons of good design, to which models must conform. The use of a constraint solver allows identifying defects in the design as violation of constraints. As a defective model satisfies a relaxed version of the constraints anyway, this suggests recovery actions. A similar feature is obtained in our approach, as we partially formalize the intents and consequences of patterns via graph application conditions, which declare invariants for the patterns to be correctly applied.

The inherently dynamic nature of graph transformation [12] has also been used to formalise patterns and the actions to be performed on models to achieve pattern conformance. In [28], patterns are represented with rules, applied to abstract syntax trees to annotate the pattern instances found. In [29], models are transformed to conform to patterns, after having exploited graph queries that detect needs for transformations. In [37] Spatial Graph Grammars provide a graph representation of GoF patterns, to transform object structure graphs so that they conform to patterns. Although declarative, the rules are based on a concrete presentation of patterns, and not on a meta-model characterisation.

Finally, several works consider pattern composition and conflicts, but again mostly with reference to GoF patterns. For example, in [36] the authors distinguish between *stringing* and *overlapping* for pattern composition. In the first case, relationships are added to relate elements that play roles in both patterns. This facilitates the identification of the individual occurrences of the composed patterns, but makes the project less cohesive. With overlapping, in a way analogous to the construction we propose, some elements play roles in both patterns. They integrate both approaches by first using stringing at a high level of abstraction, and then merging the different classes. Their approach works at the structural level only, and no technique for conflict analysis is provided. An approach analogous to pattern stringing relies on pattern componentization, proposed in [3], through which a pattern is transformed into a component allowing direct reuse of the code implementing a solution. In this case, an Eiffel contract defines the characteristics of the component. The approach in [4] formalizes patterns in First Order Logic and sees composition as overlapping, as indicated by the presence of elements in models which conform to the specification of two patterns. A static definition of composite patterns results from the conjunction of the formulas defining the individual component patterns. The approach can thus be used to verify conformance of a model to a composite pattern, but not for model completion.

In conclusion, we observe the lack of an integrated, domain-independent formalism able to give account of mutual synchronization constraints within a pattern and across different ones, and to support pattern checking, identification, application, composition and conflict analysis. In the rest of the paper we present one such formalism.

3. Pattern specification

Variable pattern. In our proposal, the simplest form of variable pattern consists of one object structure called *root* with the mandatory part that any pattern realization must contain, and a number of *variable* parts defining additional structures that can be replicated several times for each instance of the root. Variable parts can be nested, thus a nested part can only be instantiated by adding structures to an instance of its parent. In addition, each variable part is assigned a variable of type integer that can be used to define equations restricting the allowed number of its replicas, e.g. more than 0, at most 5, or any number if no restriction is given. Equations may contain relations between the allowed replicas of different variable parts, and the number of times a pattern can be instantiated in a model can be restricted by defining equations on the variability of the root. Note that if the set of equations has no solution in the natural numbers, then the pattern cannot be instantiated.

In this paper, we express both the root and the variable parts of a pattern as graphs or triple graphs. However, Definition 6 in Appendix B formalizes them in a categorical way [25], so that they can be objects of any category with pushouts and pullbacks, such as algebraic specifications, Petri nets, automata or component systems. The choice of graphs is however of great generality. As in [12], graph nodes and edges can be provided with attributes, and be typed by a so-called type graph (similar to a meta-model). Hence, models can be very naturally represented as graphs.

Example. The GoF *Observer* pattern captures one-to-many dependencies between objects so that when the *subject* object changes its state, all its dependent *observer* objects are notified and updated automatically. Fig. 1 presents a simplified version of this pattern. The root, which is associated with the variable *Ob*, is depicted with a white background and contains the classes `Subject`, `Observer` and `ConcreteSubject`, as well as their relations. Variable parts are enclosed in coloured polygons. In the example there is only one variable part, associated with the variable *Conc*, containing the `ConcreteObserver` and its relations with the elements in the root. The equation $Conc > 0$ requires at least one instantiation of the variable part for each instantiation of the root, i.e., at least one `ConcreteObserver` for each `ConcreteSubject`. Each instantiation of the variable part is glued to the root through the `Observer` and `ConcreteSubject` nodes of the root. As the root is not constrained, any number of instances of the pattern is allowed. The figure uses the concrete syntax of UML class diagrams, but the abstract syntax can also be used.

Pattern expansion. One can see a pattern *VP* as a shortcut for the possibly infinite set of valid expansions of its variable parts (those combinations satisfying the equations). This set is called the *expansion set* and is written *EXP*(*VP*). See the details of its construction in Appendix B.

Example. Fig. 2 shows some valid expansions of the *Observer* pattern. In particular, (a) is the empty graph, where the root is expanded zero times; (b) instantiates once the root and once the variable part; (c) expands the root once and the variable part twice; (d) expands the root twice and the variable part once for each root expansion; and (e) expands the root twice and the variable part once, this latter shared among the two root instances. Since the root is instantiated twice in expansions (d) and (e), they account for two instances of the pattern each. On the contrary, expanding the root once but not the variable part is not valid as it does not satisfy the equation $Conc > 0$. Note that expansions allow different instances of the root to share the same instance of a variable part, like in the case of expansion (e), as well as two different instances of a variable part to share the same instance of a nested part.

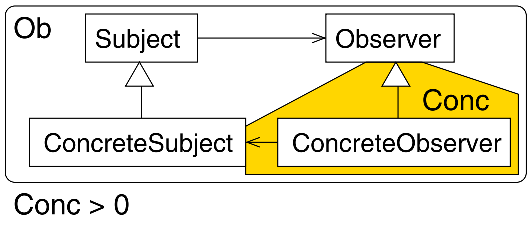


Fig. 1. *Observer* pattern (simplified).

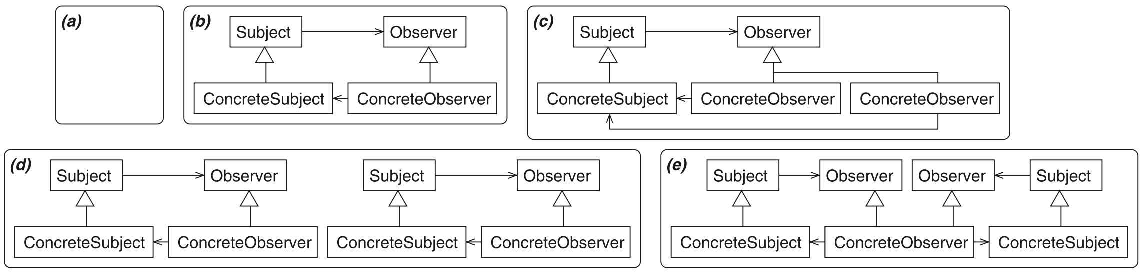


Fig. 2. Some valid expansions of the observer pattern in the set *EXP(Observer)*.

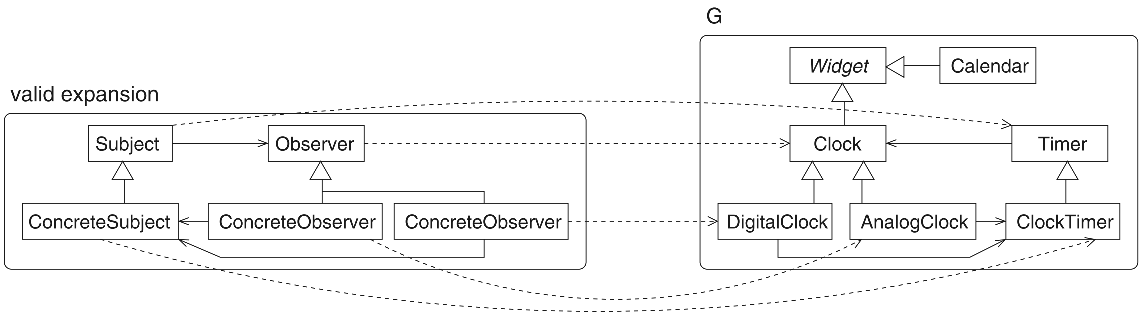


Fig. 3. Pattern satisfaction example.

Pattern satisfaction. Pattern formalizations can be effectively used as a basis for pattern discovery and reverse architecting processes that discover and document pattern instantiations in a given design. Intuitively, a graph G satisfies a pattern VP if some valid expansion $E \in EXP(VP)$ can be found in G. The algorithm for checking pattern satisfaction performs a depth-first traversal of the pattern nesting structure, counting the number of occurrences of the root and the variable parts along the way, and checking that the equations hold. This algorithm is shown in detail in Appendix B.

Example. Fig. 3 shows a model G that satisfies the specification of the *Observer* pattern, where *Clock*s provide different representations of the current system time. In particular, there is a valid expansion, shown to the left, that is the maximal element of $EXP(Observer)$ present in G.

3.1. Annotating structure with roles: annotated patterns

The structure of a pattern is usually enriched with information regarding the roles its different elements play in the collaboration, and defining a specialized vocabulary that promotes a common understanding of its parts. In order to specify such information we use triple graphs [17] (instead of graphs) to define the root and variable parts of patterns. Technically, a triple graph is composed of two graphs, called *source* and *target*, related through morphisms from the nodes of a third graph called *correspondence*. A triple graph is written $M = (M_s \leftarrow M_c \rightarrow M_t)$, representing the source (M_s), the target (M_t), the correspondence (M_c), and the morphisms from the correspondence to source and target, also called correspondence functions. We write $M = \langle M_s, M_c, M_t \rangle$ if we are not interested in explicitly showing the correspondence functions. In this work we use triple graphs where the source graph is the pattern structure, the target is a vocabulary of roles, and the correspondence assigns roles to the elements in the pattern. Note that we also use triple graphs to represent models, where the source is a domain-specific model, the target contains the vocabulary of roles of all patterns in the specification, and the correspondence assigns roles in such vocabulary to the different elements in the model. This is called a pattern-annotated model (see below).

Example. The left of Fig. 4 shows a triple graph that contains the structural part of one pattern. Its source graph at the bottom conforms to the UML abstract syntax meta-model, its target at the top is a pattern vocabulary model, and the correspondence maps UML elements to vocabulary elements. The morphisms from the correspondence are shown as dotted arrows. Intuitively, these are similar to pointers in programming languages.

The right of the figure shows the same triple graph but using the UML concrete syntax and a shortcut for roles where, instead of showing the complete triple graph, tags similar to stereotypes are employed. We will use this shortcut in the rest of the paper, where please note that the names of classes, operations and fields in the pattern are variables, which get instantiated to the names of the elements in the model.

Triple graphs are typed by meta-model triples [17] made of a source and a target meta-models, related through a correspondence meta-model. In our particular case, the meta-model triple relates the meta-model of a specific language (e.g. UML) with that of a pattern vocabulary. The top of Fig. 5 shows the meta-model for the vocabulary, made of classes `Pattern` and `PatternRole`. This meta-model needs to be specialized for the definition of patterns in concrete modelling languages, by subclassing the `PatternRole` class. For the case of UML patterns, this means that roles are applicable to operations, associations, structural features, classifiers and classes. The bottom of Fig. 5 shows part of the UML meta-model. Finally, the correspondence meta-model maps roles to UML elements, and defines a class `PatternInstance` to group the mappings of each pattern instantiation.

Pattern-annotated model. A *pattern-annotated model* is a triple graph whose source is a model in some language (e.g. UML), and the target contains an occurrence of a pattern vocabulary for each pattern used in the source model. The correspondence graph is called *annotation graph* and has a `PatternInstance` node for each instance of the pattern, and one `RoleMap` for each element playing a role in the instance.

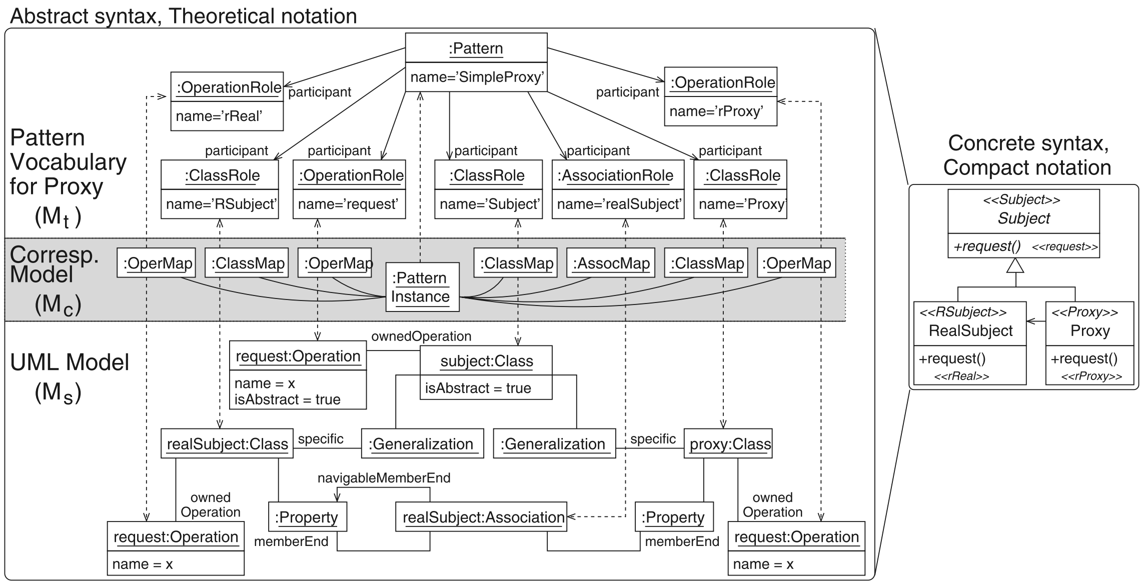


Fig. 4. Annotated pattern with the structural part of the *Proxy* pattern in theoretical (left) and compact (right) forms.

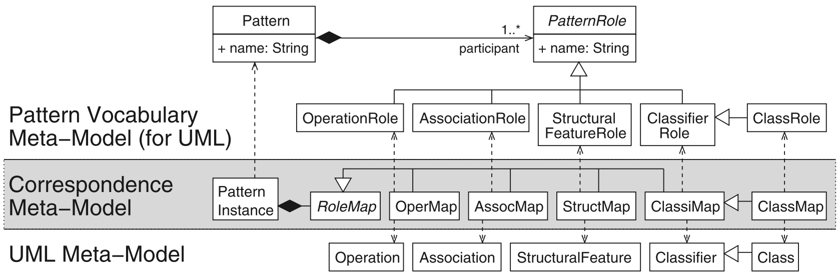


Fig. 5. Meta-model triple for UML patterns.

Note that, as shown at the right of Fig. 4, a pattern-based tool would not need to show the triple graph to the user, but the annotation can be done by marking the source graph e.g. by means of stereotypes [10]. However for the theory, an explicit triple graph has some advantages: we do not modify or extend the source meta-model (e.g. the UML one) with additional classes or attributes for tagging; triple graphs help in enforcing the patterns, as shown in Section 5; and it is easier to distinguish the instances of a pattern, as these are identified by `PatternInstance` nodes.

Annotated pattern. An *annotated pattern* is a variable pattern where triple graphs and triple morphisms are used instead of simple graphs. In this way the roles that the different elements are playing in the pattern are identified. The notion of pattern satisfaction is adapted to the use of triple graphs, but the theory remains the same, as we use a categorical framework [7]. Thus, annotated patterns are satisfied by triple graphs, called pattern-annotated models.

Example. The annotated pattern in Fig. 4, specifying the structure of the *Proxy* pattern [14] in theoretical and compact forms, conforms to the meta-model triple in Fig. 5. This pattern provides a surrogate *proxy* for an object with role `realSubject` in order to control access to it. The pattern has just one root, without variable parts, and requires that the operations in `Subject`, `RealSubject` and `Proxy` have the same name, modelled with a variable `x`. Similar to the theory of graph transformation [12], graphs in a pattern definition may have attributes with either a concrete value, or given by a variable taken from a set, whose values are constrained by formulae. More complex attribute conditions such as in [12] are possible. As we have omitted the name of classes and associations in the pattern, they can be mapped to any name.

Example. Workflow patterns [35] collect recurring constructs from existing workflow systems and provide descriptions of their usage. Their presentation is textual in the style of GoF patterns. For control flow patterns dealing with synchronization policies, an intuitive semantics through coloured Petri nets gives example realizations of the patterns, to be inferred by the reader. Fig. 6 shows formalizations for some of them, where places and transitions can play different roles such as *input*, *split*, *output*, *and* or *merging*. In particular, the pattern *Structured Discriminator* allows merging a number of branches in a process into a single subsequent branch such that the first of them to complete results in the subsequent branch being triggered, but completions of other incoming branches thereafter have no effect on (and do not trigger) the subsequent branch [35]. We formalize this by allowing two or more instantiations of the *inputs* variable part, while we require one less instantiation of region *discard*. Note how this is expressed by means of a system of equations that relate the variability of these two regions. To the best of our knowledge, this kind of constraints cannot be expressed with any other existing pattern formalization.

3.2. Synchronizing different diagrams: synchronized patterns

A pattern specification can be composed of more than one diagram, like for example in the case of the GoF patterns, which are described using class and sequence diagrams. In this case,

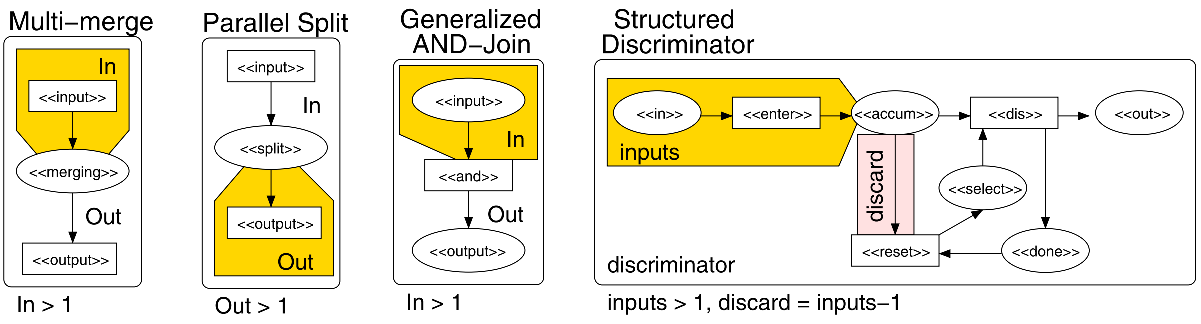
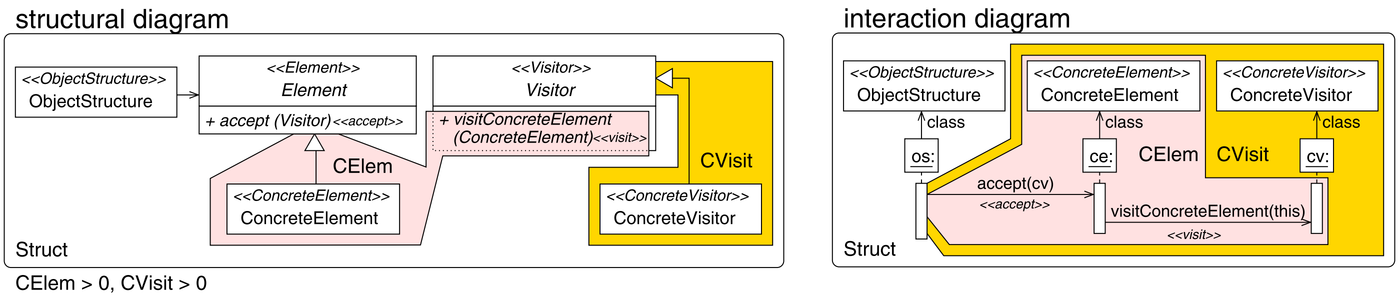


Fig. 6. Some workflow patterns.



structural diagram
<<ObjectStructure>> ObjectStructure
<<Element>> Element
+ accept (Visitor) <<accept>>
<<Visitor>> Visitor
+ visitConcreteElement (ConcreteElement)<<visit>>
CElem
CVisit
<<ConcreteElement>> ConcreteElement
<<ConcreteVisitor>> ConcreteVisitor
Struct
CElem > 0, CVisit > 0
interaction diagram
<<ObjectStructure>> ObjectStructure
<<ConcreteElement>> ConcreteElement
<<ConcreteVisitor>> ConcreteVisitor
class
os:
ce:
CElem
CVisit
cv:
accept(cv)
<<accept>>
visitConcreteElement(this)
<<visit>>
Struct

Fig. 7. The structural and interaction diagrams for the *Visitor* pattern.

relationships have to be established between the elements in the different diagrams in order to identify when the same element appears in more than one of them. We do this through the roles assigned to the elements in the pattern, and through the variable names associated with the root and variable parts.

Example. The *Visitor* pattern represents as objects the operations to be performed on the elements of an object structure, so that new operations can be defined without changing the elements on which they operate. It is one of the most complex GoF patterns as it requires two levels of variation for the two hierarchies of `Visitor` (i.e., the operations) and `Element` (i.e., the object structure), with the set of operations for `Visitor` varying together with the `ConcreteElement` set. Fig. 7 shows the structural and interaction diagrams for this pattern, where the presence in the same variance region of the signatures for the `visit` operations constrains the types of their parameters to be equal to the types of `ConcreteElement`. Both diagrams are formalized through an annotated pattern with two variable parts each, relative to the two hierarchies. However, while in the structural diagram the two parts are independent, in the interaction one they are nested. This reflects the double constraint that each concrete visitor can be accepted by each concrete element, and that each concrete visitor has an operation to visit each concrete element.

The synchronization between the different regions is represented by equality of their names (i.e., *Struct*, *CElem* or *CVisit* in the example), whereas the elements that overlap in regions with the same name are formalized in the notion of a *synchronization graph*, which factors out the common structure of the two patterns to be synchronized. In general, given *n* diagrams (i.e., *n* annotated patterns) to be synchronized with one structuring pattern, the *n* synchronization graphs are automatically calculated by the intersections of those regions with equal name with respect to the roles. This is possible as the diagrams to be synchronized share the same vocabulary model, since they describe different diagrams of the *same* pattern. Thus, the synchronization graph has one region for each two regions to be synchronized. Two elements in two annotated patterns *SP* and *IP*, if mapped to the same role in a region with same name, will be related through an element in a region of the synchronization graph. See [7] for a description of the algorithm.

Synchronized pattern. A *synchronized pattern* is made of a primary *structuring* annotated pattern *SP*, zero or more secondary annotated patterns IP_i synchronized with *SP* through synchronization graphs SG_i. For simplicity, we assume that the equations containing variables of synchronized regions are the same in *SP* and IP_i. For UML patterns, the primary pattern *SP* is a class diagram, and the secondary patterns IP_i are sequence diagrams. See [7] for the formal definition.

Example. The left of Fig. 8 shows a variation of the *Proxy* pattern allowing several proxies for a given subject. The figure shows both the structural and the interaction annotated patterns, related through a synchronization graph. The synchronization graph contains two regions, as we synchronize the two roots and the two variable parts. The *Subject* region contains the elements that are to be identified in the roots of the structural and interaction patterns (the `RealSubject` as it plays the role `RSubject` in both roots). The *Proxy* region has the identified elements in the variable parts (the `Proxy` and the `request` operations). For clarity, we explicitly show the classifiers of the `p` and `rs` objects, as both classifiers play a role in the pattern.

Synchronized pattern satisfaction. The satisfaction of synchronized patterns is similar to that for annotated patterns, but taking into account the synchronization graph. Thus, instead of looking only for occurrences of a variable part of the structural pattern to the graph, we look for a pair of occurrences of the variable part of the structural pattern and the synchronized variable part of the secondary pattern, such that their intersection (their *pullback*) is exactly the synchronization graph node relating both. This is repeated for each combination of structural and secondary patterns.

Example. The right of Fig. 8 shows one step in the satisfaction of the *Proxy* pattern to the left of the same figure. The model *M* contains a class diagram and two sequence diagrams, which a tool would present in three different views. The model has one instance of the *Proxy* pattern, and the variable region that affects the `Proxy` role has been instantiated twice (classes `ImageProxy` and `RemoteProxy`). The figure shows the first step in the satisfaction check-

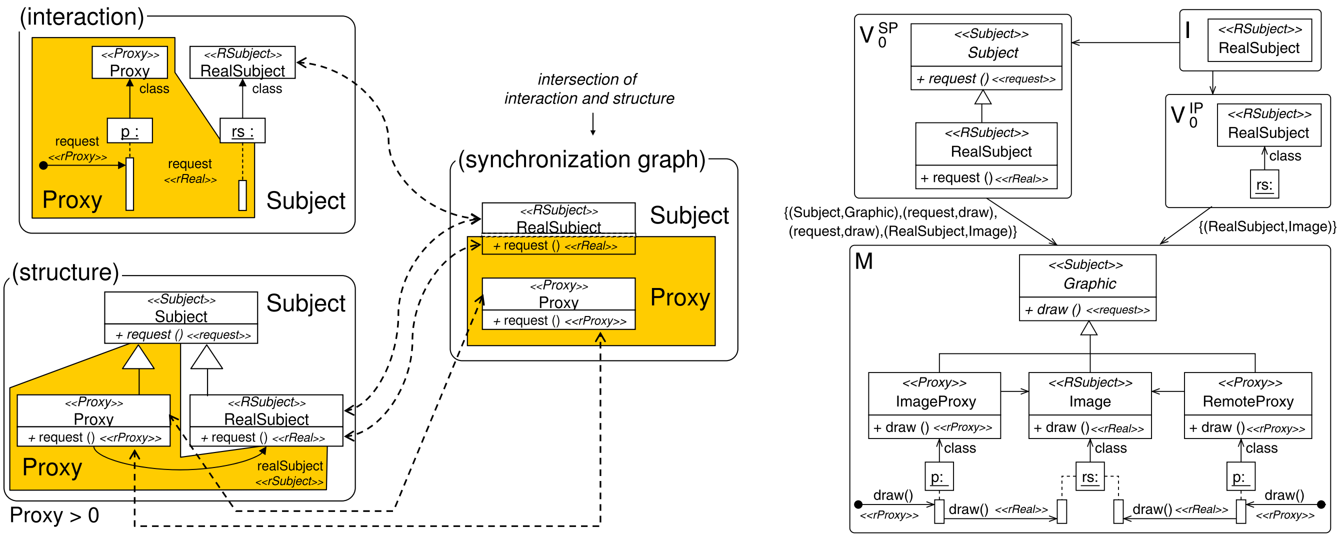


Fig. 8. Synchronized pattern for *Proxy* (left). Annotated model *M* satisfying the *Proxy* pattern and first step in satisfaction checking.

ing, depicting the pullback (i.e., the maximal intersection) of the roots, which is equal to the root of the synchronization graph. The satisfaction check follows by computing two additional pullbacks for the two instantiations of the variable parts and checking the variability equations.

4. Pattern invariants

Patterns may include contextual conditions for their correct application. For example, the intent of the *Singleton* pattern is to make available a unique instance of a class and provide a global point of access to it. This is achieved by defining the unique instance as a static attribute in the class, which is returned by a static method. Moreover, the pattern *forbids* the class to have a public or protected constructor, as this would allow other classes to create additional instances. Hence, a constraint should demand that all constructors in the class be private. This is an invariant that needs to be satisfied in order to make the pattern application correct.

We formalize this kind of pattern invariants using the notions of graph constraints and application conditions [11]. We present in this section an informal intuition, see Section B.2 in the Appendix for a formal treatment.

Pattern with invariants. A *pattern with invariants* is a pattern with a set of *pattern constraints* defined over the root or any variable part of its structural or secondary patterns. The simpler constraints are the atomic ones, and consist of one *premise* graph and a set of *consequence* graphs. As in logic, the intuition of constraint satisfaction is the following: if the premise graph is found in a model, then some of the consequence graphs have to be found as well. We call Negative Application Condition (NAC) an atomic constraint with empty consequence set, as its satisfaction requires that no occurrence of the premise graph be found. A Positive Application Condition (PAC) is the negation of a NAC, and hence demands the existence of the premise graph for its satisfaction. In addition, more complex constraints can be formed by using boolean formulae over atomic constraints.

Example. The left of Fig. 9 shows the *Singleton* pattern. It includes two invariants in compact notation, in particular two NACs, which declare model fragments forbidden to occur. The NACs are represented in the pattern inside crossed grey rectangles. The first one forbids public constructors in the singleton, while the second one forbids finding a protected constructor. We have used the notation "(...)" to mean "*any number of parameters*", which in the abstract syntax is depicted as an `Operation` object without reference to its `Parameters`.

As we will show in Section 6, invariants help detecting pattern conflicts, i.e., whether two patterns can be applied together in a certain way. They can also be used to indicate incompatibility of

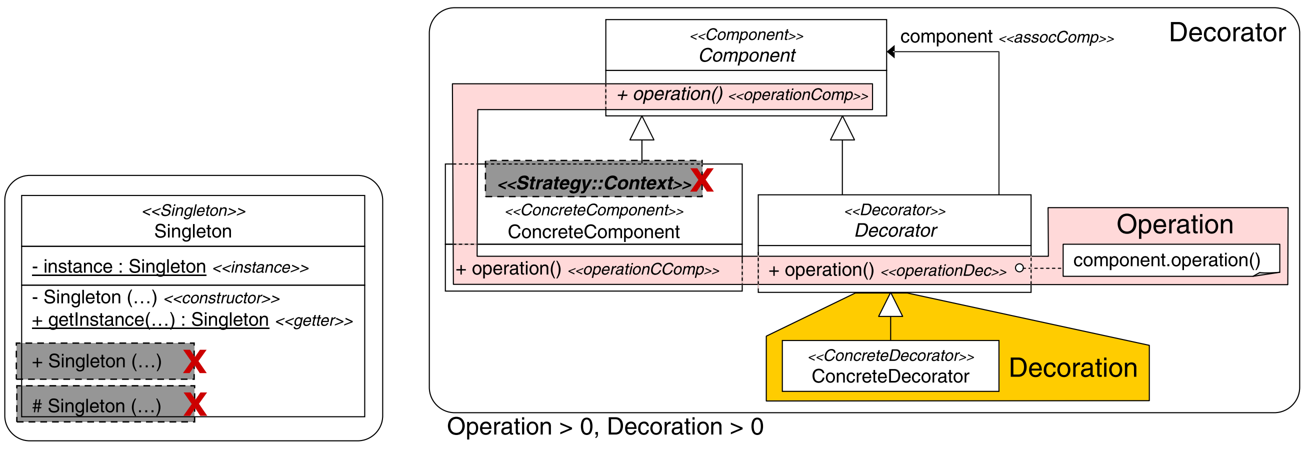

<<Singleton>> Singleton
- instance : Singleton <<instance>>
- Singleton (...) <<constructor>>
+ getInstance(...) : Singleton <<getter>>
+ Singleton (...)
# Singleton (...)
<<Component>> Component
+ operation() <<operationComp>>
component <<assocComp>>
Decorator
<<Strategy::Context>>
<<ConcreteComponent>> ConcreteComponent
+ operation() <<operationCComp>>
<<Decorator>> Decorator
+ operation() <<operationDec>>
Operation
component.operation()
<<ConcreteDecorator>> ConcreteDecorator
Decoration
Operation > 0, Decoration > 0

Fig. 9. *Singleton* pattern with two invariants (NACs) (left). Specifying role conflicts as invariants for the *Decorator* pattern (right).

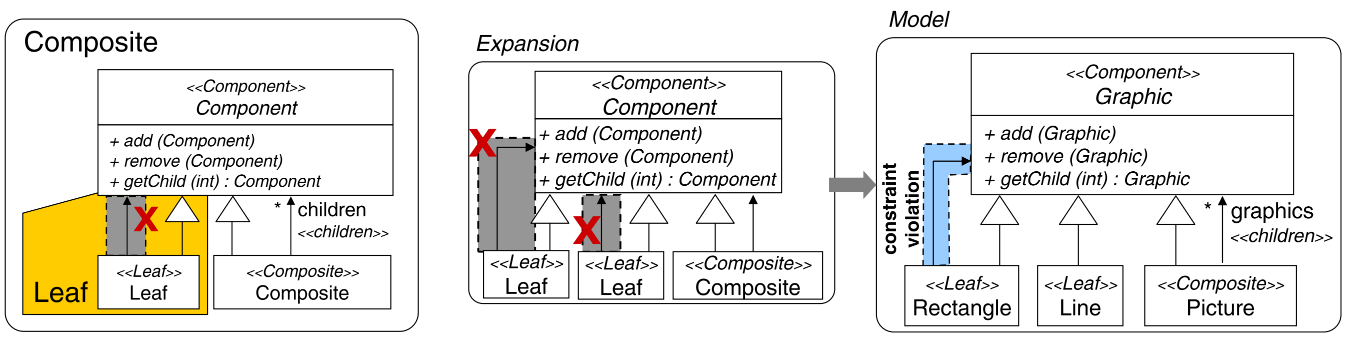


Fig. 10. *Composite* pattern with invariants (left). One expansion of the *Composite* pattern, and a model that does not hard-satisfy it (right).

an element to simultaneously play two roles in different patterns, which is done by including the forbidden role combinations as NACs in the patterns. This is possible as we work with triple graphs where the roles are in a separated graph component. As an example, the right of Fig. 9 shows the *Decorator* pattern, where it is forbidden for the `ConcreteComponent` to play the role of `Context` in any instance of the *Strategy* pattern. This is so because although both patterns allow defining extra behaviour or modifying the behaviour of a class, using *Decorator* precludes using *Strategy*. The client of a *Decorator* uses the `Component` to invoke the operation, which is propagated through the chain of decorators until it reaches the `ConcreteComponent`. On the contrary, the client of a *Strategy* uses directly the operation in the `ConcreteComponent`. Thus, we can think of a *Decorator* as a *skin* over an object that changes its behaviour, while the *Strategy* changes the object's *guts* [14]. Finally note also that, as we use triple graphs, should we forbid an element to play two roles defined in the same pattern, we can specify in the invariant whether the prohibition is for roles in the same instance of the pattern or in different ones. This is done by referring to the same or different `PatternInstance` nodes in the correspondence graph.

Satisfaction of pattern with invariants. In order to check whether a model satisfies a pattern *VP* with invariants, we have to check that each replica of the variable parts satisfies its associated invariants (if any). The details of this procedure can be seen in the Appendix, Section B.2. Moreover, considering invariants leads to the notions of *hard* and *soft* satisfaction. The former is used on annotated models, while the latter is used on unannotated ones for pattern identification (i.e., finding pattern instances in models). Intuitively, in hard satisfaction, we take the maximal graph *E* in *EXP(VP)* for which there is an occurrence in the model, and then we demand that the invariants hold. If they do not, the pattern is not satisfied. In soft satisfaction we take the maximal element in *EXP(VP)* which, once embedded in the model, satisfies all invariants. Hard satisfaction makes sense when the model is annotated with roles, so that we know exactly which elements of the model belong to a certain instance of the pattern. Soft satisfaction is useful for pattern identification in order to suggest the (maximal) parts of the model that form an instance of the pattern.

Example. Fig. 10 shows the *Composite* pattern, which composes objects into tree structures to represent part-whole hierarchies, and allows clients to treat individual objects and compositions of objects uniformly [14]. The pattern is presented to the left, where for simplification we only consider one variable region concerning the number of leaves. The pattern contains one NAC that forbids leaves to be connected with the `Component`, as this is only permitted (demanded indeed) for the `Composite` role. To the right, the figure shows an example of pattern expansion with two leaves and their corresponding NAC. Finally, the right shows a model that does not hard-satisfy the pattern because the leaf `Rectangle` declares a link to the component. On the contrary, the model soft-satisfies the pattern because it contains an expansion of the pattern with just one leaf (`Line`). Hence, soft satisfaction permits neglecting replicas of variable parts that, if considered part of the pattern occurrence, would make the occurrence violate the pattern. Nonetheless, soft satisfaction does not make sense for annotated models: since *Rectangle* has role `Leaf`, it belongs to the pattern instance, and hence it is forbidden to have a link to the `Component`. Should we have a model without role annotations, soft satisfaction could be used to find instances of the pattern, and then a composite with one leaf would be found.

As explained before, expansion graphs have attached replicas of the invariants defined in the pattern. Some of these expansions may already satisfy all invariants, which is the case when for each constraint, the premise and some consequence graph is found in the expansion. We call such expansions *strong expansions*. In some other cases, the expansions need the model they are embedded in to provide certain elements in order to evaluate the invariant. Thus, their validity depends on the environment. This is for example the case if we find the premise of an invariant in the expansion but none of its consequences. Hence, the environment has to provide the occurrence of some consequence. These expansions are called *weak expansions*. Note that expansions containing NACs are always considered weak. Finally, an expansion is *unsatisfiable* if there is no model that can satisfy it. This is so if the expansion is attached a NAC for which there is already an occurrence in the expansion. This classification of expansions is useful for the procedure for pattern-based model completion, shown in next section. A precise discussion of these kinds of expansions is provided in Appendix B.

5. Pattern-Based model completion

Our formalization of patterns can be used by modelling tools to automatically complete models according to patterns. In this scenario, a user interacts with the model, while the vocabulary of pattern roles as well as the annotation graph (the correspondence) are automatically built through the application of patterns, yielding as a result a pattern-annotated model. The tool would offer a catalogue of patterns from which the user chooses a suitable one, and specifies which elements of the model play some role in the selected pattern, if any. After this choice is made, the tool would complete the model (i.e., the source, the target and the annotation graphs) in case all constraints are satisfied. If the pattern constraints are not satisfied, the user would be suggested with some model transformations that ensure their satisfaction (Section 5.1). If applying the pattern violates the invariants needed by other already existing pattern instances, the user would be warned.

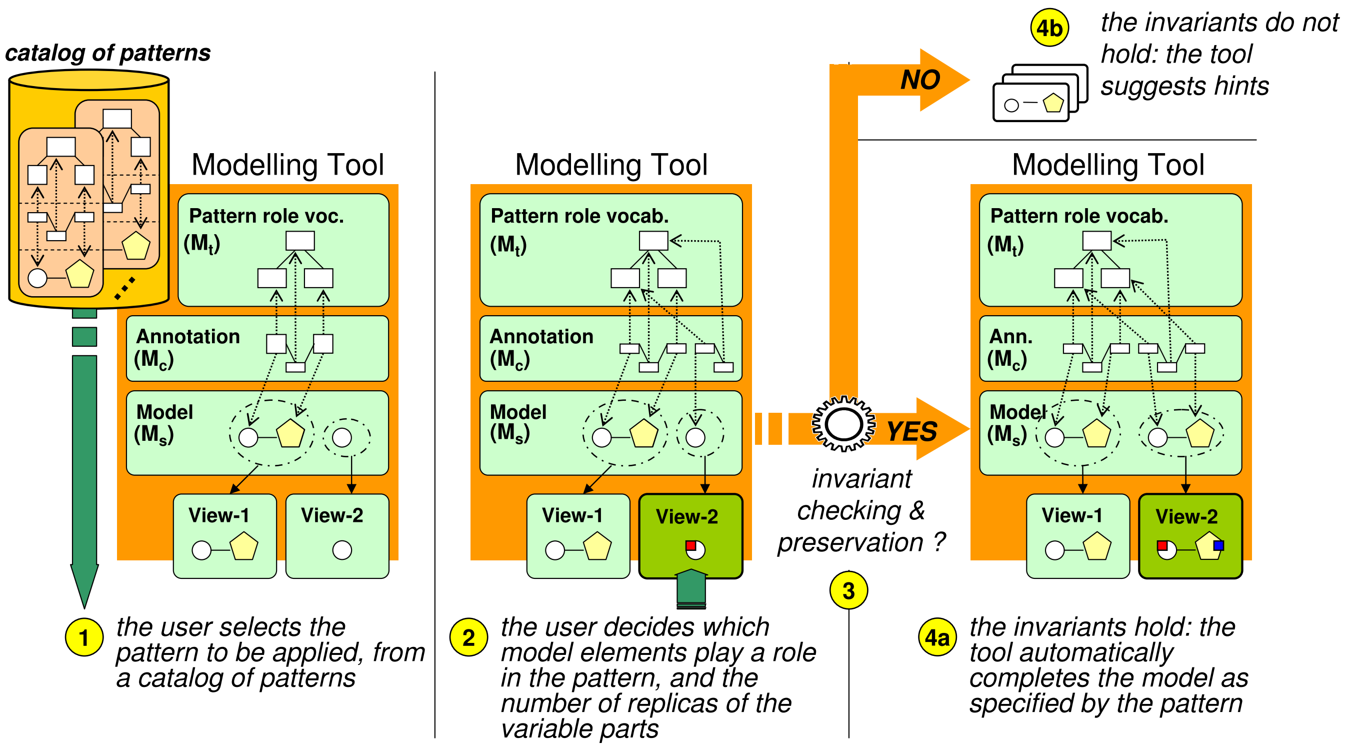

catalog of patterns
Modelling Tool
Pattern role voc.
(M_t)
Annotation
(M_c)
Model
(M_s)
View-1
View-2
1
the user selects the pattern to be applied, from a catalog of patterns
Modelling Tool
Pattern role vocab.
(M_t)
Annotation
(M_c)
Model
(M_s)
View-1
View-2
2
the user decides which model elements play a role in the pattern, and the number of replicas of the variable parts
NO
YES
invariant checking & preservation ?
3
4b
the invariants do not hold: the tool suggests hints
Modelling Tool
Pattern role vocab.
(M_t)
Ann.
(M_c)
Model
(M_s)
View-1
View-2
4a
the invariants hold: the tool automatically completes the model as specified by the pattern

Fig. 11. Usage scenario of a pattern-based tool for model completion.

This process is depicted in Fig. 11, where we assume a unique model acting as a repository (e.g. a UML model conformant to the UML meta-model). The tool would present views of this model, which correspond to different diagrams, e.g. along the lines of [16]. Note that the annotation of roles is an example of *model marking* in MDA, and can be carried out e.g. by adding tags (shown in the figure as coloured squares on the elements of the views).

Next we sketch the algorithm that implements step 4a in Fig. 11, namely the application of a pattern to a model $M = \langle M_s, M_c, M_t \rangle$ in order to perform pattern-based model completion. As a reminder, M_s is the source model, which contains the design diagrams (e.g. in UML); M_t is a model containing the pattern vocabulary elements; and M_c is the correspondence model, which annotates the different elements in the source model with roles in the vocabulary. The procedure starts by applying the primary pattern (steps i–v), then checks the satisfaction of invariants (steps vi and vii), and finishes by applying the secondary patterns (step viii). This procedure can be used to create new instances of a pattern in a model, as well as to extend previous existing instances. See [7] for a more detailed description of the algorithm and a proof of its correctness for patterns without invariants.

(i) *Vocabulary extension.* If M_t does not contain the definition of the pattern and its roles, as it happens the first time the pattern is instantiated, then such a definition is added to M_t.
(ii) *Role annotation.* The user selects the elements playing some role in the pattern from M_s. Then, a `RoleMap` node is automatically created in M_c for each of these elements, associated with a node *pm* of type `PatternInstance`. This is a new node for a new instance of the pattern, or an existing one if we are extending a previous instance of the pattern. This process constructs the morphisms from the modified annotation graph M_c to M_t and M_s.
(iii) *Instance extraction.* The portion of the primary pattern that is already in the model M is identified. For this purpose, a so-called pattern graph PG is built by navigating from *pm* to the elements in M_c belonging to the defined instance of the primary pattern SP, and from these to the elements in M_t and M_s along the correspondence morphisms.
(iv) *Variability instantiation.* The user selects a number r of instantiations for each variable part of the primary pattern, such that the existing number of instances e plus the new ones r satisfy the variability equations. We select from *EXP(SP)* an expansion E with $r + e$ instantiations of each variable part. If E is an unsatisfiable expansion, then the model M cannot be completed and the user is warned about this.
(v) *Model extension.* M is extended with an instance of the primary pattern that has the selected number of variable parts. For this purpose and starting from M, the modified model M' is built by constructing the pushout[1] of E and M through PG. That is, we add the elements missing from E to the original model M, yielding M'.

Example. Fig. 12 shows the application of the *Proxy* to a model M containing a class *Image*, which the user mapped to role `RSubject`. The expansion graph E contains two proxies, as the user selected two instantiations of the variable part; hence two proxies are created in the resulting model M'. The name of the operation in the pattern ("request", a variable) is mapped to the name of the operation in the model ("draw"), and similarly for class names.

(vi) *Invariant checking.* If the chosen expansion E is a strong expansion, then it already satisfies the invariants and no additional checking has to be performed. On the contrary, if E is a weak expansion, then it has to be checked if M' hard-satisfies E and its invariants. If not, M' does not satisfy the pattern, and the procedure finishes by warning the user of this fact, especially of the invariants of E that cannot be satisfied. In Section 5.1 we propose some mechanisms that hint the user on actions enabling the application of the pat-

[1] Given two objects A_i, whose "intersection" is given by $A_1 \leftarrow I \rightarrow A_2$, the pushout $A_1 \rightarrow B \leftarrow A_2$ is their union, where the common elements (given by I) are "merged".

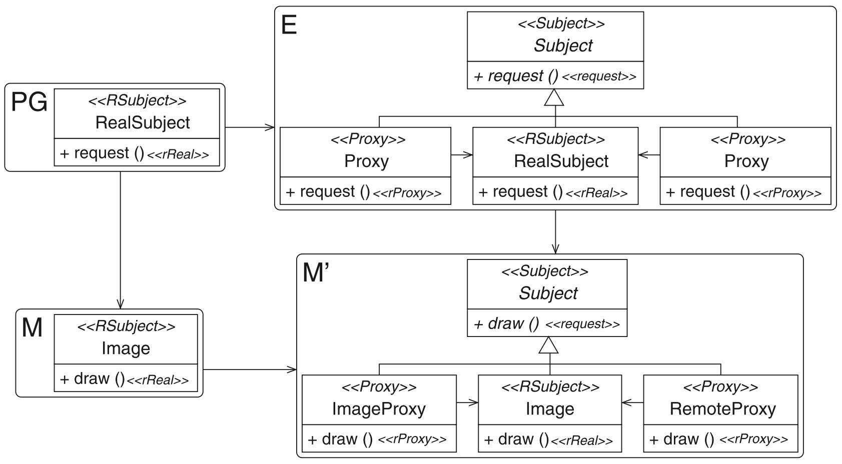


Fig. 12. Applying the structural pattern of `Proxy` to an annotated model M yielding M'.

tern on these cases. In case the invariants hold, the applied expansion E and its morphism to M' are stored in a set *Patt*, together with the morphisms (occurrences) of their constraints. This is done in order to facilitate the invariant preservation in next step.

(vii) *Invariant preservation*. It is checked that no invariant of existing pattern instances in M is broken by applying the new pattern. This is done by taking each element $E' \in Patt$ and checking that: (i) no occurrence of E''s NACs appears in the new model M', and (ii) if for some constraint with a non-empty set of consequences, a new occurrence of the premise appears in M', then some of the associated consequences must be found as well. Note that the PACs of existing pattern instances cannot be violated as this procedure adds elements to M but does not delete anything. If some constraint is violated, the procedure finishes by warning the user of it (and the expansion is deleted from *Patt*). Otherwise the procedure continues by enlarging M' with the application of the secondary pattern.

(viii) *Application of secondary patterns*. Finally, the secondary patterns (e.g. sequence diagrams for GoF patterns) are applied to the model. For this purpose, the model M' is enlarged by a sequence of pushouts, which can be seen as a model transformation. To be precise, firstly it is checked which variable parts of the primary pattern were added to M to yield M'. This is necessary as the user may have extended an existing instance. Then, the synchronization graph is used to locate the variable part of the secondary pattern synchronized with the variable part of the primary, and a pushout is built (see [7]). As a result, the model is enlarged with an instance of the secondary pattern. The procedure is repeated for each secondary pattern synchronized with the primary one. The variable parts of secondary patterns may also contain invariants and, as in the case for the primary pattern, it has to be checked that they are satisfied and that no invariant of existing pattern instances is violated. The procedure is incremental – one can update an existing instance – and supports heterogeneous synchronization, such as the one shown in Fig. 7, where the structural pattern has two independent variable parts and nesting in the interaction pattern.

Example. Fig. 13 shows the creation of the first sequence diagram when applying the *Proxy* pattern starting from model M' of Fig. 12. First, the I node of the synchronization graph (which

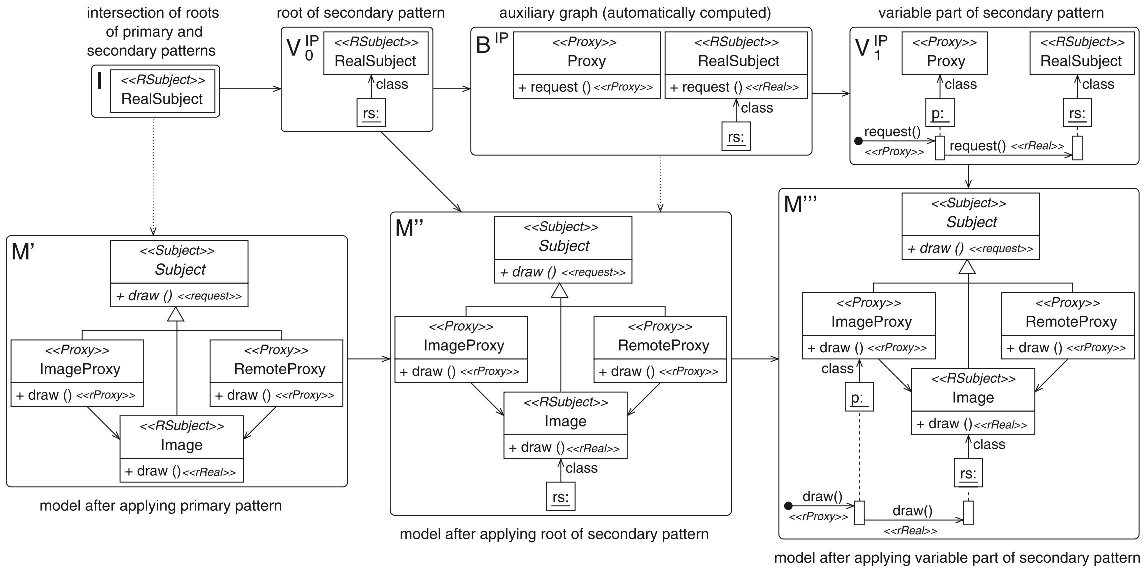


Fig. 13. Synchronization of secondary pattern: building the first sequence diagram.

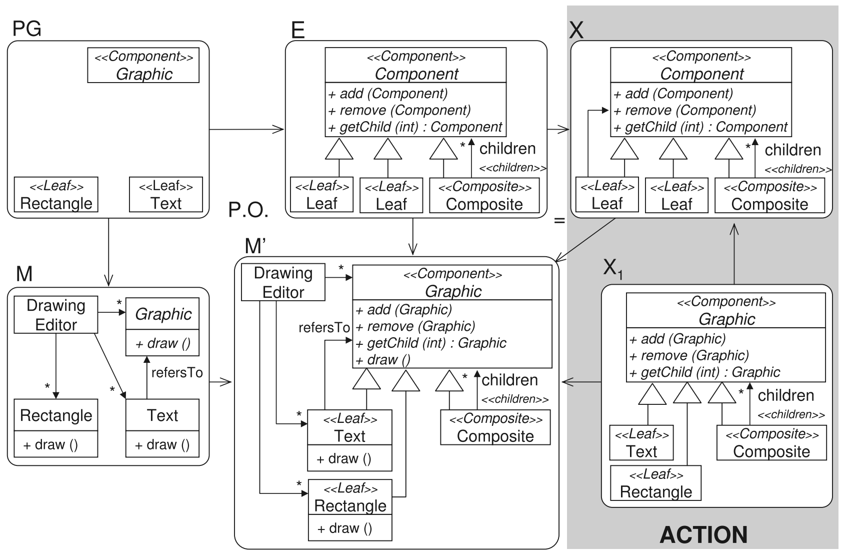


Fig. 14. Suggesting tips to allow an application of the *Composite* pattern.

contains the intersection of the roots of the primary and the secondary pattern) locates the place in M' where the root V_0^{IP} of the secondary pattern is to be applied. Then, the variable part V_1^{IP} of the secondary pattern is added through an intermediate graph B^{IP} (see [7]) to yield M'''. As there were two instantiations of V_1^{SP}, the procedure would follow by adding an additional sequence diagram (i.e., another instance of V_1^{IP} through a pushout).

5.1. Suggesting tips to enforce invariant satisfaction

As explained in the sixth step of the previous pattern-based model completion procedure, it is possible that a weak expansion of a pattern cannot be applied because some of its invariants are not satisfied by the model in which it is embedded. This subsection presents an algorithm that returns some actions (deletion or addition of elements to the models), which can be suggested to the user so that the model satisfies the invariants imposed by the applied pattern and those of other existing pattern instances in the model. We restrict the study to the case of atomic constraints.

There are two scenarios where a weak expansion cannot be applied. The first one is due to the existence of a NAC that is not satisfied. Hence, a starting model M was completed according to an expansion E to yield M', but here an occurrence of the NAC exists. Thus, we generate a set of deleting rules [12] or "graph differences" that eliminate parts of the NAC instance and which are presented to the user so that he can select the most appropriate one to enforce the satisfaction of the invariants. In particular, we generate the partial order of all graphs X_j bigger than or equal to the expansion E and smaller than the forbidden NAC X. As these graphs contain the elements to be preserved, those elements that are in the NAC but not in X_j are deleted. In addition, each fragment to be deleted should let intact (i) each pattern occurrence already existing in M' and (ii) each positive constraint of every existing pattern. For this purpose we use the set *Patt* that our previous procedure built, and which stores the patterns applied in the model as well as the occurrences of their invariants. We do not care about NACs of existing pattern instances as deleting elements never violates them. The details on the construction of these rules are given in Section B.3 in the Appendix.

Example. Fig. 14 shows an example where and expansion E with two leaves of the *Composite* pattern cannot be applied to a model M. The reason is that one of its NACs is found in the model M' that results from applying its primary pattern to M: the leaf `Text` has an association to the `Component`. For simplicity, we assume no other pattern has been applied. Thus, our algorithm hints the user that one solution is to delete the link from the `Text` class to the `Graphic` class. This is represented as the deleting rule $X \Rightarrow X_1$ with the intuitive meaning: "if X is found in the model then replace such occurrence by X_1". As X is "bigger" than X_1, this means deleting the elements that appear in X (the NAC) but not in X_1, so that the NAC is no longer present.

In this example, the conflict is due to an element (the link) that does not play a role in another pattern. However, some conflicts may arise between elements belonging to other patterns. In this case, one can use the static analysis techniques shown in the next section to detect the possibility of such conflicts. Moreover, one can either use the algorithms presented in this section, or give more specialized solutions for each conflict type.

The second scenario where a weak expansion cannot be applied is a pattern defining an invariant with a non-empty consequence set, which is not satisfied. This is so because at least one occurrence of the premise X is found, but no consequence is found. The way to proceed is similar to the previous case. We build a set of suggestions, each building one consequence graph C_n. This set can be seen as a set of non-deleting rules of the form $X \Rightarrow C_n$. Such set is offered to the user so that, if some suggestion is adopted, all constraints for the expansion occurrence E are satisfied. As before, we must ensure that the elements C_n in the set of suggestions do not produce an occurrence of some negative constraint in other patterns instances, or that they do not introduce an occurrence of a premise but not of a consequence for some existing constraint.

6. Pattern composition and conflict analysis

This section describes a procedure for composing patterns, which enables the realization of pattern-based languages. It also presents static methods to analyse whether such composition is possible by checking conflicts derived from the pattern invariants.

6.1. Composing patterns

We present two ways of composing patterns. The first one yields a new pattern where a new role is created for two elements that are identified together. Thus, this method enables the creation

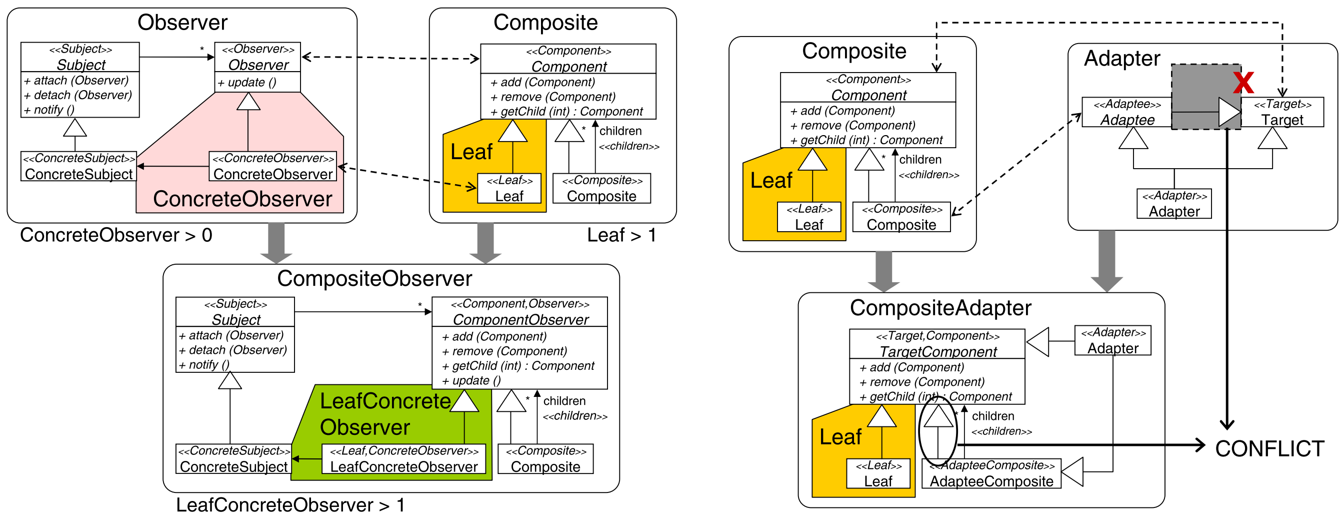


Fig. 15. Composing the *Observer* and *Composite* patterns (left). An example of conflict between the *Composite* and *Adapter* patterns (right).

of a pattern language with an operator for creating new patterns. The second method is like a "macro" that allows applying two patterns in one step.

For the first case, the main idea is to select the elements to be identified in the roots of both patterns. Then, both roots are glued through those identified elements (i.e., a pushout is built) yielding the root of the composite pattern. The process is repeated for the elements in the variable parts that one wants to identify. The second composition operation does not generate a new pattern, but the operation retains the roles of the original elements in the two patterns. The procedure is exactly the same, but the roles are not identified when specifying the intersections of the variable parts. Concerning the variability equations, both merged variable parts receive the same name, hence we perform the union of the original equations (once we do the renaming) so as to consider the most restrictive ones.

Example. The left of Fig. 15 shows a composition of the *Composite* and *Observer* patterns. In the composition, we have chosen to keep the roles in both patterns, i.e., we follow the first of our approaches to pattern composition. In addition, we have chosen to identify the `Observer` with the `Component` in the roots, and the `Leaf` with the `ConcreteObserver` in the variable parts. The resulting composed pattern has as equation the union of the sets of equations of the original patterns, once variables are renamed, $var' = \{LeafConcreteObserver > 0,\ LeafConcreteObserver > 1\}$. This results in taking *LeafConcreteObserver* as the most restrictive ($LeafConcreteObserver > 1$). The formal description of this example is shown in the Appendix, Section B.4.

6.2. *Analysing pattern interactions and conflicts*

As patterns are equipped with constraints expressing invariants, it is possible to check possible conflicts statically at "composition-time". We have identified the following three conflict types, restricting to atomic constraints and PACs. These cases appear when generating the constraints in the composition. Recall that all these resulting constraints are incorporated into the composed pattern.

Fatal conflicts. In this first kind of conflicts, an unsatisfiable composition is obtained, and hence the identification of elements made by the composition is invalid. There are two sources for this:

- *NAC in root or variable part.* The composed pattern has a NAC that is found in its root or in some of its variable parts. This pattern is unsatisfiable as, whenever it is applied, an occurrence of the NAC will be present. This conflict is detected by finding a morphism from the NAC to the root or variable parts of the composed pattern, which is shown to the user.

  Example. The right of Fig. 15 shows a composition of the *Adapter* and *Composite* patterns. The former contains a NAC forbidding the *Adaptee* to generalize the `Target`. This is reasonable as, if one decides to solve the problem by subclassing the `Target`, the *Adapter* pattern is not needed. The built composition identifies the `Target` with the `Component`, and the `Adaptee` with the `Composite`. The resulting *CompositeAdapter* pattern violates the NAC, as there is an occurrence of it in the root of the composed pattern, hence that identification of roles is not consistent with the invariants. Note that once we detect such conflict we can directly add the role conflicts as invariants to the original *Composite* and *Adapter* patterns, as we did to the right of Fig. 9. This will improve performance next time we try to compose the patterns, as this combination will not be permitted from the beginning, as well as when trying to apply one of the patterns on elements that play a role in an existing instance of the other.
- *NAC in PAC.* A NAC included in a PAC also yields incompatibility of the resulting composed pattern. This is so because PACs have to be found in the model, but when the PAC is found, so will be the NAC and hence both constraints can never be satisfied at the same time.

Conflicts affecting satisfaction. This kind of conflict, which is less severe, comprises two cases. In the first one, a (possibly strongly satisfiable) pattern may become weakly satisfiable and hence the environment has to provide some parts in order for the pattern to be satisfiable. In the second one, a weakly satisfiable pattern may become strongly satisfiable.

- *Premise in variable part.* In this case, an occurrence of the premise of a constraint is found in a variable part of the composed pattern, but no occurrence of any consequence is found. Thus, any expansion of the pattern that instantiates such variable part at least once becomes a weakly satisfiable expansion, as we need the environment to provide an occurrence of some consequence.
- *Consequence in variable part.* Here an occurrence of some consequence of a constraint is found in a variable part of the composed pattern. Thus, any expansion of the pattern that

instantiates such variable part at least once becomes a strongly satisfiable expansion (with respect to that particular constraint).

Conflicts between invariants. The last two cases represent conflicts between invariants, which result in losing a part of the semantics of the invariant as a consequence of the invariant interaction.

- *NAC in premise.* If an occurrence of a NAC is found in the premise of a constraint then such constraint can be removed from the composition, as the only way to satisfy both the constraint and the NAC is by not finding an occurrence of the premise. This is so because, if the premise is found, then so is the NAC. This conflict means that we have lost the invariants specified by the consequence part of the constraint.
- *NAC in consequence.* If an occurrence of a NAC is found in the consequence C_k of a constraint, then such consequence C_k can be removed from the composition. Hence, the only way to satisfy the NAC and the constraint once we find an occurrence of the premise is to find an occurrence of some other consequence different from C_k. This is so because, if C_k is found, so will be the NAC and hence the NAC will not be satisfied. Note that if the constraint has C_k as unique consequence, it is converted into a NAC.

7. Examples

In this section we provide additional examples that show the wide applicability of our proposal and the usefulness of the techniques presented in this paper.

7.1. GoF patterns: Invariants on the use of Singleton

As discussed in Section 4, the intent of the *Singleton* pattern is to ensure that only one element of the class deemed as `Singleton` can exist. This is achieved at the structural level by defining a static variable typed as the class, a private constructor and a static public method always returning the same instance of the class. At the behavioural level, one needs to insure that an instance of the singleton can be obtained only through a public static method, which must have a distinctive behaviour, namely, if the class variable is not already initialized, it initializes it calling the private constructor. In any case, it returns a handle to the instance referred to by the static variable.

The left of Fig. 16 shows the structural part of the solution through a class diagram in abstract syntax (where the *name* attributes in the objects are assigned to variables), and depicts in the center the behaviour of the public method `getInstance()` through an activity diagram in concrete syntax. For simplicity we do not show again the NACs already presented in Fig. 9.

While a class constructed exactly as above satisfies the pattern, one can ask under which conditions the solution described to the right of Fig. 16, given directly in Java for simplicity, is still an instance of the pattern. In this case, the method `getOptimizedInstance()` sets a field of `OptimizedSingleton` depending on some characteristic of the request. Depending on how the code for setting the instance is defined, the class satisfies the pattern or not. In particular, if this does not use `getInstance()` or does not replicate exactly its structure, then one could actually create several instances of `OptimizedSingleton`, while still providing a match for the root of the pattern.

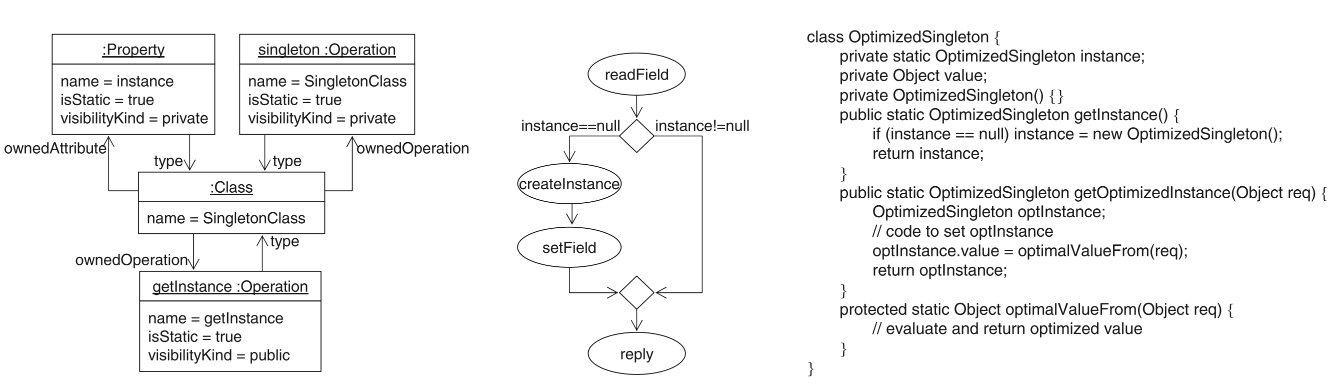
One can prevent this situation, while still ensuring flexible adaptation of the pattern, by combining two types of constraints, as expressed in the left of Fig. 17. The NAC prevents the direct use of the constructor for the `Singleton` class in a method other than `getInstance()`, while the constraint of the form $X \rightarrow C$ ("if premise (X) then consequence (C)") requires that each method returning an instance of the class proceeds through a call to `getInstance()`.

The complete definition of the *Singleton* pattern resulting from the addition of the constraints in Fig. 17 makes the composition of this pattern with the *Prototype* pattern unsatisfiable. Indeed, the *Prototype* pattern, as presented to the right of the same figure, demands that a `clone` operation exists which returns a new instance of a realization of the `Prototype` interface, by calling its constructor, conflicting with the NAC on *Singleton*. A conflict analysis would then rule out the possibility of composition through identification of the `Prototype` and `Singleton` class roles. Note that the identification of the `getInstance()` and `clone()` operations is not possible due to the different values of the attribute `isStatic`. Once such an incompatibility is detected, it can also be transformed into an explicit NAC on the roles, as in the example of *Decorator* and *Strategy* (see the right of Fig. 9).

The reader can find a formalization of all GoF patterns with our approach in [8].

7.2. Workflow Patterns: Describing safety conditions

The left of Fig. 18 formalizes the *Critical Region* workflow pattern. This pattern provides a means to prevent two or more sections of a process from executing concurrently. This may happen if tasks within the sections require exclusive access to a common resource for completing the tasks [35]. The pattern suggests that a place, called `mutex`, acts as mediator for this exclusive access. The variable region `participant` specifies that the mutex can be accessed by any number of processes that need to be synchronized. They should get a token from the mutex in order to enter



```
class OptimizedSingleton {
      private static OptimizedSingleton instance;
      private Object value;
      private OptimizedSingleton() {}
      public static OptimizedSingleton getInstance() {
            if (instance == null) instance = new OptimizedSingleton();
            return instance;
      }
      public static OptimizedSingleton getOptimizedInstance(Object req) {
            OptimizedSingleton optInstance;
            // code to set optInstance
            optInstance.value = optimalValueFrom(req);
            return optInstance;
      }
      protected static Object optimalValueFrom(Object req) {
            // evaluate and return optimized value
      }
}
```

Fig. 16. Structural (left) and interaction (center) diagrams for pattern *Singleton*. A class satisfying the *Singleton* under certain conditions (right).

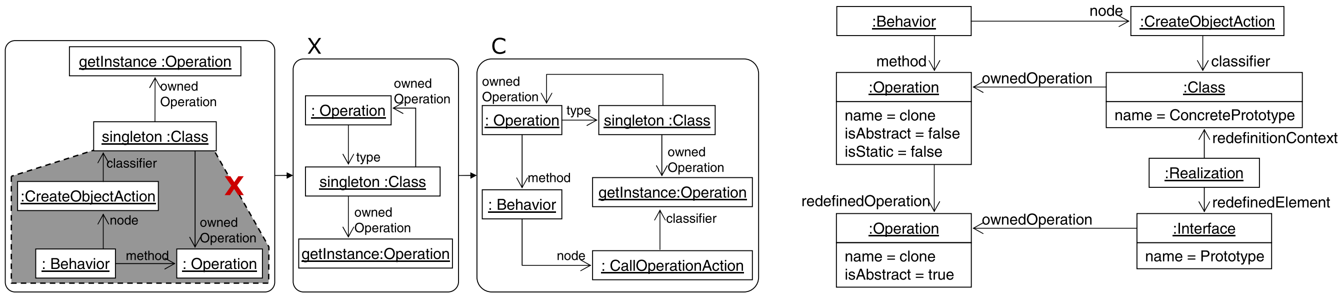


Fig. 17. Ensuring the singleton solution (left). The *Prototype* pattern (right).

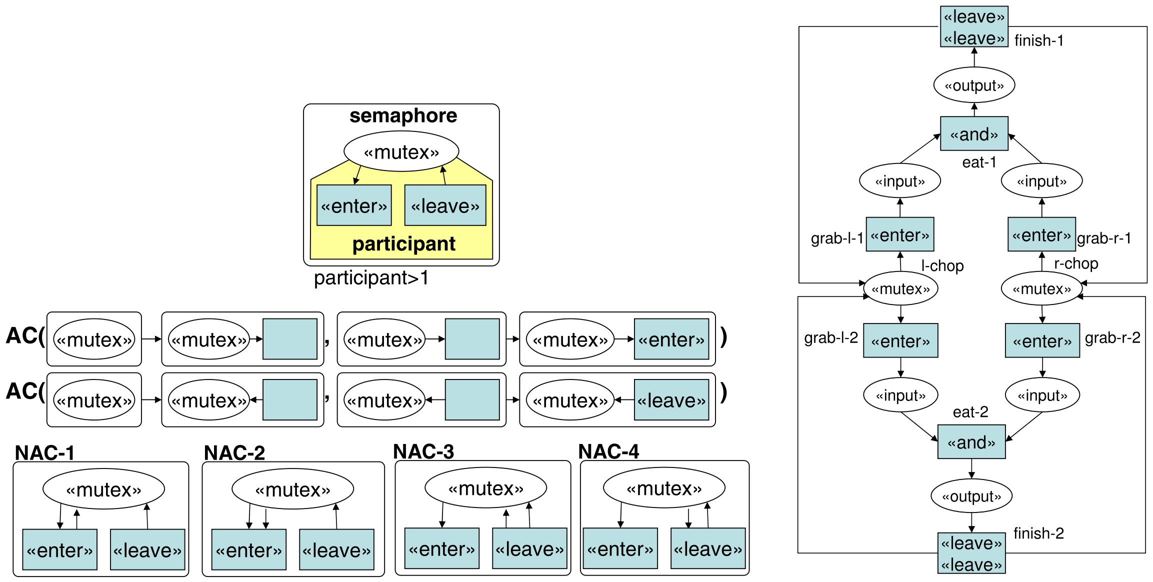


Fig. 18. The *Critical Region* workflow pattern (left). Dining philosophers model (right).

the critical region, and put the token back once they leave. Note however that this pattern imposes many security restrictions for its correct use, shown below. We use the notation *AC*(*Premise*, *Consequence*) (see Definition 9 in Appendix B). Hence, the first constraint states that, if a transition takes tokens from the mutex (premise), such transition should have a role `enter` (consequence). Similarly, the second specifies that if a transition puts tokens in the mutex, it must have the role `leave`. This guarantees that only transitions participating in the pattern access the mutex. In addition, four NACs restrict the participant transitions, so that they have exactly one arc to the mutex.

The right of Fig. 18 shows the classical “dining philosophers” model. The model contains two philosophers competing for two shared resources (two chopsticks) that they need to start eating. Hence, two instances of *Critical region* regulate the access to each chopstick, with two expansions of the `participant` variable region each. Also, each philosopher process uses the *generalized and-join* pattern shown in Fig. 6 so that they start eating once they have both chopsticks. Should we want to modify the model to introduce the fact that the waiter can change a chopstick from left to right, this is done by adding one transition connecting *l-chop* to *r-chop*. However, this modification violates the invariants for the *Critical region* pattern that state that only transitions of participants can access the mutex, and so the user is warned that the modification breaks the two instances of the pattern.

7.3. Enterprise integration patterns: Composing architecture styles and routing policies

Enterprise integration patterns [18] describe integration solutions across different implementation technologies. The solutions consist of many different components such as applications, databases, messages or routing components. In [18] the authors use a particular visual notation to illustrate the use of the patterns, which is the one we use in Fig. 19. The patterns to the left specify three different integration styles so that several applications can work together and exchange information, namely through a shared database, file or message interchange.

The right of Fig. 19 shows the composition of the *Shared Database* integration style pattern, with a message routing pattern called *Recipient list* which allows routing messages to the correct system based on the message content [18]. This second pattern has a root `list` made of the router component, and a variable part made of the recipients and channels through which they receive the messages. In this way, the composed pattern allows routing a message to a number of applications sharing a database. This is useful, e.g., when integrating a system dealing with different types of queries to be answered by several applications on the same database system. The final composed pattern is shown in the lower-right corner, where we may simplify the variability equation to just *applics* > 1.

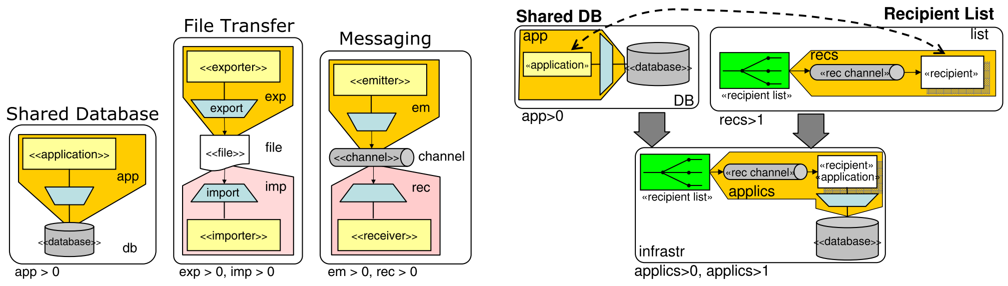


Fig. 19. Enterprise integration patterns describing integration styles (left). Composition (right).

7.4. Interaction patterns: Expressing pattern dependencies

Interaction patterns [31] represent a way to capitalize on experience on all different aspects of interaction design, covering layout, navigation, action coordination, etc. They are usually not formalized, but presented through examples or in textual form. We illustrate here some examples in the direction of their formalization, relying on a particular meta-model at the foundation of a markup language for abstract description of interfaces [34], a fragment of which is presented on the left-hand side of Fig. 20. The meta-model presents an interactive system as composed of several models, in which an abstract user interface is realized through concrete elements, referring to domain objects and workflow descriptions. We only show the classes used in the examples, and omit most associations for clarity. The meta-model only considers abstract user interfaces and not concrete ones. Interestingly, this means that the realization of the pattern can be deferred to the mapping of the abstract to the concrete interface, without loading pattern descriptions with unnecessary detail, e.g. on the specific classes to be used.

Inspired by [31], we have built a pattern vocabulary for the examples, where the `Affordance` role can be played by elements on which user actions can be performed, while the `Presentation` role pertains to elements providing a visual clue to the user. In a classical MVC implementation, these roles would be played by different objects, but different paradigms could collapse them on the same object, e.g. in the Swing architecture.

Fig. 20 shows to the right two patterns concerning the coordination of the presentation of actions on a common container. In the *ButtonGroup* pattern, a small number of actions relative to the same element are presented within a single container, while in the *ActionPanel* pattern, the same container is used to present a greater number of actions on possibly unrelated elements. The examples of concrete interfaces illustrate how the affordance role

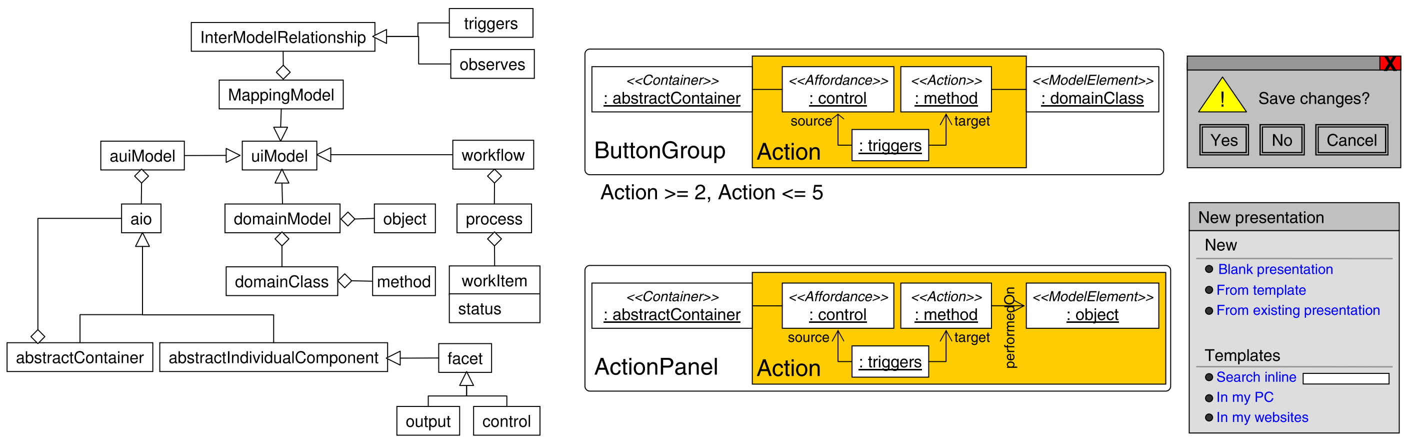


Fig. 20. A fragment of the UsiXML meta-model (left). Examples of action patterns (right).

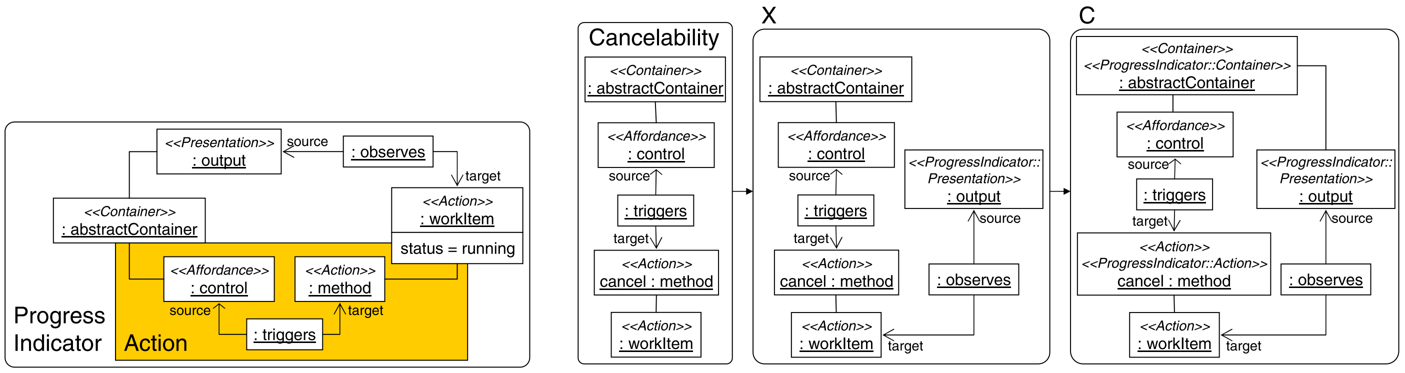


Fig. 21. Two patterns associated with workflow items with duration. *Cancelability* (right) expresses a dependency with *Progress Indicator* (left).

can be played by different types of concrete widgets, buttons in the first case and links in the second.

Fig. 21 shows other two action patterns, *Progress Indicator* and *Cancelability*, related to processes with duration, as represented by a `status` attribute. In the first case, a container is provided in which, while the process is running, it is possible to have a visual representation of the current advancement of the work item, and have controls available, e.g. to suspend, resume or cancel the job. The second pattern provides a way to establish access to an action to interrupt any such process. An atomic constraint expresses the request that if the progress of the item is presented through the progress indicator pattern, the cancel action is presented in the same container. Note that in this case the constraint is used as a means to add a dependency between patterns.

8. Conclusions and future work

In this work we have presented a formal approach to the specification of patterns, as well as procedures for pattern-based model completion, pattern composition and analysis of pattern conflicts. In the proposed formalization, patterns are annotated by roles in a vocabulary, may contain variable parts (possibly nested), support synchronization of the variable parts inside a diagram and across different types of diagrams, and may contain constraints formalizing environmental conditions. The latter may be used to formalize some intentions and consequences of the patterns.

Our approach presents several benefits with respect to existing ones. First, the proposal relies on a general meta-model for patterns, not necessarily based on UML, and the algorithms and constructions are presented in a categorical framework so that they can be applied to a great variety of domain-specific languages. This was exemplified through the formalization of patterns in quite different domains. Moreover, the categorical framework permits using model fragments of the domain-specific language to specify the pattern. This has the advantage that the pattern designer does not have to learn a formal complex mathematical language for the formalization. Second, variability regions – with the possibility of nesting – together with the equations governing their permitted instantiations are more flexible than current proposals, which annotate single elements with cardinalities [13] and for which it is more difficult to express that several elements have to vary together. For example, complex dependencies on permitted instantiations of variability regions (like, e.g., Fig. 6) are possible in our approach, and difficult to express in most approaches. Third, our mechanism for pattern application considers synchronization of several diagrams. The synchronization is flexible, as it allows for example synchronizing nested variable parts with independent ones, like we showed in the *Visitor* example in Fig. 7. Fourth, we separate the roles from the pattern structure by a triple graph, without extending existing meta-models. This clean and non-intrusive solution facilitates the manipulation and querying of the vocabulary models, as well as the identification of pattern instances. Moreover, using separate models for the roles and the traces facilitates the implementation, as one does not need to modify the meta-model of the domain-specific language for which the patterns are defined. Such modification is impractical or even impossible in most cases. Fifth, our use of constraints to express extra environmental conditions enables the analysis of pattern conflicts, which can be done statically. Finally, our formal treatment facilitates the derivation of recovery actions oriented to preserve the satisfaction of existing pattern instances in models. Thus, altogether we believe that this work is an enabling step towards the automated and flexible use of pattern specifications for modelling with domain-specific languages.

We are implementing this approach in a meta-modelling framework, which is a new version of the AToM^3 meta-modelling kernel [9] rewritten in Java. As the tool allows the definition of domain-specific modelling languages through meta-modelling, our language-independent formalization enables the definition of a general infrastructure to define patterns for different languages. For the algorithms presented in Section 5 (to complete models and to enforce invariants) one can resort to graph rewriting techniques. It is important to mention that two crucial features needed are the ability to generate rules on the fly, as well as to store, interactively select and pass matches between rule applications. A future contribution will present this meta-modelling framework in depth, together with its pattern-based modelling capabilities, based on the formalization we have introduced in this paper.

Concerning the current limitations of the approach, consider for example the *Critical Region* pattern shown in Fig. 18. In order to formalize the intent of this pattern completely, one should rely not only on syntactical aspects, as shown with the graph constraints, but also on behavioural ones. For example, a reasonable assumption is to ask each participant process to be deadlock free, and to control that a mutex can never receive more tokens than it had in the initial marking. In general, this cannot be done with purely syntactical means, but the constraints would have to query the reachability graph. One way to do it is to allow constraints with formulae in computational tree logic. It is up to future work to combine structural patterns with behavioural constraints. Note that in our current approach we may have constraints on behavioural diagrams, but only concerning their syntax.

Other lines of future work include formalization of patterns in additional languages and improvement of tool support, both for modelling and for analysis. The presented notion of satisfaction enables the use of our formalization for pattern identification, but heuristics specific to the domain language may be needed here. We are also working towards augmenting the expressivity of our patterns. Whereas the replication mechanism of variable parts is powerful enough to cover many interesting patterns, the replication occurs in “width” but not in “length”. For example, with our current formalization, it is difficult to generate a path of connected nodes of arbitrary length. Even though we have detected that variability in patterns usually occurs in width, considering replication in length is also interesting, and could be done along the lines of [24].

Acknowledgements

This work has been supported by the Visiting Professor Programmes of “Sapienza” University of Rome and its Department of Computer Science, the R&D program of the Community of Madrid (S2009/TIC-1650, project “e-Madrid”), the CAM-UC3M project “EXPLORE” (CCG08-UC3M/TIC-4487), as well as by the Spanish Ministry of Science and Innovation, under project “METEORIC” (TIN2008-02081), and mobility Grants JC2009-00015 and PR2009-0019. The authors are grateful to the reviewers and the editor that helped in improving the paper.

Appendix A. Introduction to category theory

This appendix introduces the technical concepts used in the Appendix B. In particular, we introduce the concepts of category, pushout, pullback and colimit, and give graphs and triple graphs as examples of categories. The interested reader can consult some book specialized on this subject, like [25].

A category is a structure that has objects and arrows (or morphisms), with a composition operation on the morphisms and an identity morphism for each object.

Definition 1 (*Category*). A *category* $C = (Ob_C, Mor_C, \circ, id)$ is defined by:

- a class Ob_C of objects;
- for each pair of objects $A, B \in Ob_C$, a set $Mor_C(A,B)$ of morphisms;
- for all objects A, B, $C \in Ob_C$, a composition operation $\circ_{(A,B,C)}$:-$_{(A,B,C)}$:*$Mor_C(B,C) \times Mor_C(A,B) \rightarrow Mor_C(A,C)$; and*
- for each object $A \in Ob_C$, an identity morphism $id_A \in Mor_C(A,A)$,

such that the following conditions hold:

1. Associativity: for all objects A, B, C, $D \in Ob_C$ and morphisms f: $A \rightarrow B$, g: $B \rightarrow C$ and h: $C \rightarrow D$ holds: $(h \circ g) \circ f = h \circ (g \circ f)$.
2. Identity: for all objects A, $B \in Ob_C$ and morphisms f: $A \rightarrow B$ holds: $f \circ id_A = f$ and $id_B \circ f = f$. ■

Instead of $f \in Mor_C(A,B)$ we write f: $A \rightarrow B$ and leave out the subscript for the composition operation. For such a morphism f, A is called its domain and B its codomain. One example of category is *Set* with objects all sets and arrows the total functions. Another category that we commonly use is *Graph* with objects graphs and morphisms graph homomorphisms. A graph can be described by a tuple $G = (V,E,s,t)$, where V is a set of nodes (or vertices), E is a set of edges, and s: $E \rightarrow V$ and t: $E \rightarrow V$ are the source and target functions, which assign the source and target nodes to each edge. Graph homomorphisms are therefore pairs (m^V, m^E): $G^1 \rightarrow G^2$ of set morphisms mapping the nodes and edges of the graphs G^1 and G^2, such that the structure of the graph is preserved (i.e., $m^V \circ s^1 = s^2 \circ m^E$, and $m^V \circ t^1 = t^2 \circ m^E$). The identity arrow is the arrow that maps each node and edge to itself. Graph homomorphisms are associative as total functions for sets are associative. The structure of graphs can be enriched with attributes in nodes and edges, as well as with types for nodes and edges. See [12] for more details.

One can build new categories out of simpler ones. For example, we can build the category of triple graphs *TriGraph* out of the category *Graph*, by taking as objects tuples of three graphs (named source, target and correspondence) and two graph morphisms from the correspondence to the source and target. Technically, what we build is a functor category $Graph^{\cdot \leftarrow \cdot \rightarrow \cdot}$.

A pushout generalises the union of two objects in a given category. Intuitively, it is the object that results from the gluing of two objects along a common subobject.

Definition 2 (*Pushout*). Given two morphisms f: $A \rightarrow B$ and g: $A \rightarrow C \in Mor_C$. A *pushout* (D,f',g') over f and g is defined by:

- a pushout object D and
- morphisms f': $C \rightarrow D$ and g': $B \rightarrow D$ with $f' \circ g = g' \circ f$,

such that the following universal property is fulfilled: for all objects X with morphisms h: $B \rightarrow X$ and k: $C \rightarrow X$ with $k \circ g = h \circ f$, there is a unique morphism x: $D \rightarrow X$ so that $x \circ g' = h$ and $x \circ f' = k$. See the left of Fig. 22. ■

The pushout object D is unique up to isomorphism. That is, if (X,k,h) is a pushout over f and g, then x: $D \rightarrow X$ is isomorphic to $x \circ g' = h$ and $x \circ f' = k$. Vice-versa, if (D,f',g') is a pushout over f and g and x: $D \rightarrow X$ is an isomorphism, then (X,k,h) is a pushout over f and g, where $h = x \circ g'$ and $k = x \circ f'$. The center of Fig. 22 shows an example of pushout in the *Graph* category, where the morphisms are depicted by using the same node identifiers in the different graphs. As it can be noted, the pushout amounts to glueing the nodes and edges of graphs C and D to the nodes and edges of A.

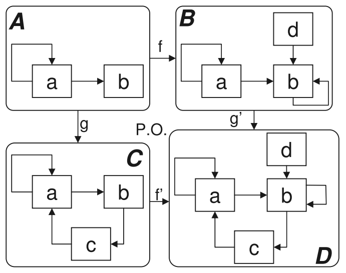
The dual construction for a pushout is that of a pullback. Intuitively, a pullback is the generalised intersection of objects over a common object.

Definition 3 (*Pullback*). Given two morphisms f: $C \rightarrow D$ and g: $B \rightarrow D$, a *pullback* (A,f',g') over f and g is defined by:

- a pullback object A and
- morphisms f': $A \rightarrow B$ and g': $A \rightarrow C$ with $g \circ f' = f \circ g'$,

such that the following universal property is fulfilled: for all objects X with morphisms h: $X \rightarrow B$ and k: $X \rightarrow C$ with $f \circ k = g \circ h$, there is a

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] \arrow[dr, phantom, "="] & B \arrow[d, "g'"] \arrow[ddr, bend left, "h"] & \\
C \arrow[r, "f'"] \arrow[drr, bend right, "k"'] & D \arrow[dr, dashed, "x"] \arrow[dr, phantom, "=", shift left=2] \arrow[dr, phantom, "=", shift right=2] & \\
 & & X
\end{tikzcd}$$



$$\begin{tikzcd}
X \arrow[dr, dashed, "x"] \arrow[drr, bend left, "h"] \arrow[ddr, bend right, "k"'] & & \\
 & A \arrow[r, "f'"] \arrow[d, "g'"'] \arrow[dr, phantom, "="] & B \arrow[d, "g"] \\
 & C \arrow[r, "f"] & D
\end{tikzcd}$$

Fig. 22. Pushout (left). Example of pushout and pullback in category **Graph** (center). Pullback (right).

$$\begin{tikzcd}
A_0 \arrow[r, "f_0"] \arrow[drr, "g_0"'] & A_1 \arrow[r, "f_1"] \arrow[dr, "g_1"] & \cdots \arrow[r, "f_{i-1}"] & A_i \arrow[r, "f_i"] \arrow[dl, "g_i"'] & A_{i+1} \arrow[r, "f_{i+1}"] \arrow[dll, "g_{i+1}"] & \cdots \\
 & & X & & &
\end{tikzcd}$$

$$\begin{tikzcd}
A_0 \arrow[r, "f_0"] \arrow[drr, "g_0"'] \arrow[ddrr, "h_0"'] & A_1 \arrow[r, "f_1"] \arrow[dr, "g_1"] \arrow[ddr, "h_1"] & \cdots \arrow[r, "f_{i-1}"] & A_i \arrow[r, "f_i"] \arrow[dl, "g_i"'] \arrow[ddl, "h_i"'] & A_{i+1} \arrow[r, "f_{i+1}"] \arrow[dll, "g_{i+1}"] \arrow[ddll, "h_{i+1}"] & \cdots \\
 & & X \arrow[d, dotted, "\alpha"] & & & \\
 & & Y & & &
\end{tikzcd}$$

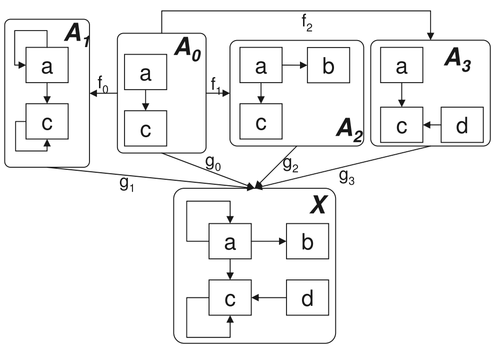


Fig. 23. Cocone (upper left). Colimit (lower left). Example of colimit in category **Graph** (right).

unique morphism x: $X \to A$ such that $f' \circ x = h$ and $g' \circ x = k$, see the right of Fig. 22. ■

The example in the center of Fig. 22 is also a pullback, as A is the maximal intersection graph of B and C in the bigger context graph D.

Finally, we introduce colimits as a generalized pushout in which we can glue together several objects through their common parts. For this purpose, we first introduce cocones.

Definition 4 (*Cocone*). A *cocone* $(X,\{g_i\})$ of a diagram with objects A_i and morphisms f_k is an object X and a family of morphisms g_i: $A_i \to X$ which are coherent with f_k, i.e., $g_i = g_j \circ f_k$ for all f_k: $A_i \to A_j$, as the upper left of Fig. 23 shows. ■

Definition 5 (*Colimit*). A colimit is a cocone $(X,\{g_i\})$ such that for any other cocone $(Y,\{h_i\})$ there is a unique morphism α: $X \to Y$ such that $\alpha \circ g_i = h_i$ for all objects A_i, as the lower left of Fig. 23 shows. ■

The right of Fig. 23 shows an example colimit in the *Graph* category. Note that we can make colimits of arbitrary coherent diagrams, and that the center of Fig. 22 is another example of colimit, where graph D is the colimit object.

Appendix B. Formal definitions

This appendix provides the formal definitions of the concepts intuitively presented throughout the paper.

B.1. Pattern specification

This section formalizes the concepts presented in Section 3. We start by defining a *Variable pattern*.

Definition 6 (Variable pattern). A variable pattern is a construct $VP = (P, root, Emb, name, var)$, where:

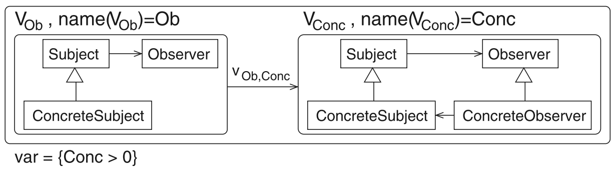


Fig. 24. A simplified version of the *Observer* pattern in theoretical form.

- $P = \{V_1, \ldots, V_n\}$ is a finite set of non-empty graphs, where each V_i is called *variable part*,
- *root* is a distinguished element of P, also called the *fixed part*,
- *Emb* is a set of morphisms $v_{i,j}$: $V_i \to V_j$ with $V_i, V_j \in P$, such that it spans a tree rooted in $root \in P$ with all graphs $V_i \in P$ as nodes and the morphisms $v_{i,j} \in Emb$ as edges,
- *name*: $P \to L$ is an injective function assigning each variable part a *name* from a set of variables L, of sort $\mathbb{N}$,
- $var \subseteq T_{AlgIEq}(name(P))$ is a set of equations governing the number of possible instantiations of the variable parts. These equations use variables in $name(P) \subseteq L$, arithmetic operations, and are restricted to use the $<, \leqslant, =, >, \geqslant$ relation symbols. We call this signature "*Algebraic Inequalities*" (Σ_{AlgIEq}) and hence T_{AlgIEq} ($name(P)$) is the term-algebra with variables in $name(P)$. ■

Each variable $name(V_i)$ – of type $\mathbb{N}$ – receives values accounting for the number of instances of the variable part V_i in a pattern realization. An instance of the pattern is valid if the valuation of the variables (i.e., an assignment of values to variables) satisfies the equations in *var*. We use the notation $Val \vDash var$ to mean that a valuation *Val* satisfies the equations in *var*. Note that if the set of equations *var* has no solution in the natural numbers, then the pattern cannot be instantiated.

Example. Fig. 24 presents a simplified version of the *Observer* pattern using Definition 6. This pattern was presented in compact form in Fig. 1, where all graphs were displayed together. Here we encode the theoretical form, with full definition $VP = (P = \{V_{Ob}, V_{Conc}\}, root = V_{Ob}, Emb = \{v_{Ob,Conc}\text{:} \quad V_{Ob} \to \quad V_{Conc}\}, name = \{(V_{Ob}, Ob), (V_{Conc}, Conc)\}, var = \{Conc > 0\})$. Representing the variability with the morphism $v_{Ob,Conc}$: $V_{Ob} \to V_{Conc}$ allows us to replicate the subgraph in V_{Conc} that is not identified by $v_{Ob,Conc}(V_{Ob})$, and the replicas are glued through their common elements in $v_{Ob,\ Conc}(V_{Ob})$. The number of allowed replicas is given by the equations in *var*, which may use the names of the variables assigned to the variable parts, of type $\mathbb{N}$. In our example, such names are *Ob* and *Conc*.

Next, we provide the construction for the set of expansions *EXP(VP)* of a variable pattern *VP*. Hence, a pattern specification *VP* can be seen as a compact form for expressing a (possibly infinite) language of graphs, *EXP(VP)*. This set is indeed a partial order, taking graph inclusion as the order relation.

Definition 7 (*Expansion set*). Given a variable pattern *VP*, its expansion set *EXP*(*VP*) is given by all graphs $E_{i,j}$ s.t. there is a surjective function $f_{i,j}$: $C_i \to E_{i,j}$ from the set of all colimits $\{C_i\}$ of all possible diagrams α obtained by replicating the graphs in P, and the morphisms in *Emb*, such that:

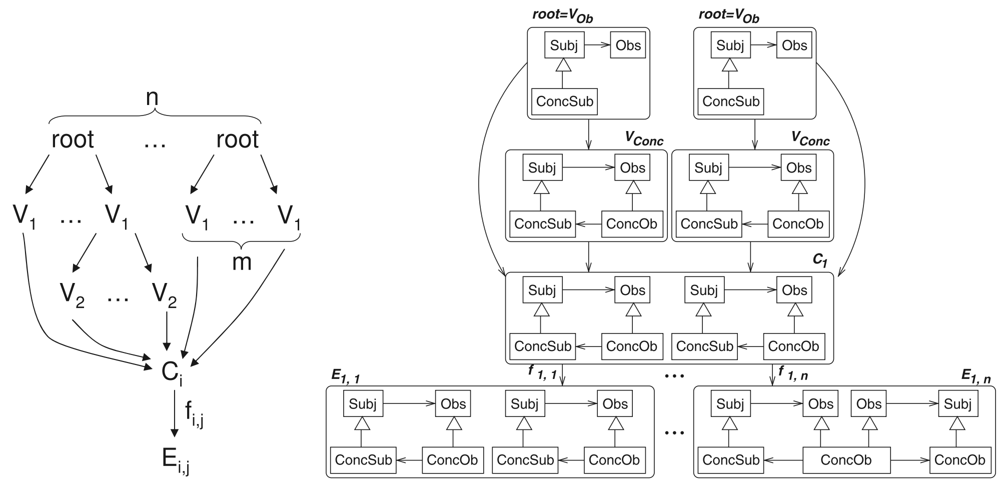


Fig. 25. Construction for the *EXP*(*VP*) set. Example of construction for the *Observer* pattern (right).

- the diagram α is consistent with the morphisms in *Emb*, which means that if $V_i \to V_j$ is included in α, then there is a morphism $v_{i,j}$: $V_i \to V_j$ in *Emb*.
- the number of replicas in each path from *root* to C_i satisfies the equations in *var*. ■

Example. The left of Fig. 25 shows one of the diagrams α_i needed to calculate the expansion set in case the *Emb* set contains two morphisms $\{root \to V_1, V_1 \to V_2\}$. The scheme has *n* instantiations of the root, and the right-most instance is connected to *m* replicas of V_1. We assume that this valuation satisfies the equations. A similar reasoning holds for the left occurrence of root. Then the colimit C_i is obtained and all surjective morphisms $f_{i,j}$: $C_i \to E_{i,j}$ are calculated. This last step is performed to permit the identification of elements in the different graphs of the diagram. This is illustrated to the right of the same figure, where one element of the expansion set of the *Observer* pattern is calculated. The diagram is made of two instances of the root, each having one instance of the variable part, hence these two valuations satisfy the equations. The bottom of the diagram shows two of the obtained model expansions, which were also shown in Fig. 2. The one to the right identifies both `ConcOb` objects in C_1 into a single one in $E_{1,n}$.

A model satisfies a variable pattern when some pattern expansion is found in the model.

Definition 8 (*Pattern satisfaction*). Given a variable pattern *VP* and a graph *G*, we say that *G* satisfies *VP*, written $G \vDash VP$, iff there is an injective morphism $E_{i,j} \to G$, where $E_{i,j} \in EXP(VP)$. ■

As the definition of the elements in *EXP*(*VP*) is non-constructive, we provide a procedure for checking satisfaction. It performs a depth-first traversal of the *Emb* tree, counting the occurrences of the variable parts V_j along the way, assigning such values to *name*(V_j), and building with them a valuation called *Val*. Once a leaf in *Emb* is reached, the equations defined in *var* need to hold, given the variables values stored in *name*(V_j).

Algorithm for pattern satisfaction. Given a graph *G* and a variable pattern *VP*, *G* satisfies *VP*, written $G \vDash VP$, iff $SAT(G, VP, M_{root} = \{p^i: root \to G\}, root, \{(name(root), |M_{root}|)\}) =$ **true**, where SAT is defined as follows:

function SAT(G: Graph, VP: VariablePattern, Occ: set of morphisms, el: VariablePart, Val: Valuation): Boolean

```
1. Let $Children := \{V_j | \exists v_{el,j} \in Emb\}$.    // Direct children of el in Emb
2. if ($Children = \emptyset$) then return $Val \vDash var$.    // el is leaf: check equations
3. $\forall p^k \in Occ$:    // Take each occurrence of el in Occ
   3.1. $\forall V_j \in Children$:    //Take all children
        // All occurrences of $V_j$ adjacent to $p^k$
        3.1.1. Let $M_j^k = \{p^j : V_j \to G | p^k = p^j \circ v_{el,j}\}$.
        3.1.2. if not $SAT(G, VP, M_j^k, V_j, Val \cup \{(name(V_j), |M_j^k|)\})$
               then return **false**.    // Check sat of $p^k$'s path

4. return **true**.
```

Note that some steps of the algorithm rely on graph pattern matching (the initialisation, where a set *Occ* of occurrences of the root is given, and step `3.1.1`), leading to the subgraph isomorphism problem [33], which is known to be NP-complete. Note however that in step `3.1.1` the matching has to be adjacent to a previously found morphism, thus reducing the search space. Optimised algorithms for pattern matching taking into consideration types and attributes have been successfully applied in the graph transformation area [5].

As an example, the left of Fig. 26 describes the morphisms considered by the satisfaction function SAT given a pattern with two levels of nesting, i.e., $Emb = \{root \to V_1, V_1 \to V_2\}$, and assuming the presence of a single morphism $p^1_{root} \in M_{root}$, i.e., only one occurrence of the root in *G*. SAT is called with the graph, the pattern, the M_{root} set, the *root* element and the valuation $Val = \{(name(root), 1)\}$ as parameters. As there is one child of *root* in *Emb*, the step `3.1.1` in the procedure picks the unique morphism in M_{root} and obtains the set M_1^1 of all morphisms $p_1^1, \ldots, p_1^n : V_1 \to G$ commuting with $G \overset{p^1_{root}}{\leftarrow} root \overset{v_{root,1}}{\rightarrow} V_1$. Then, the step `3.1.2` makes a recursive call to SAT with $el = V_1, Occ = M_1^1$ and the updated valuation $\{(name(root), 1), (name(V_1), |M_1^1| = n)\}$ as parameters. This procedure is repeated so that, for each element $p^k \in M_1^1$, a set M_2^k of occurrences of V_2 commuting with p^k is obtained, and a new recursive call is made with the valuation $\{(name(root), 1), (name(V_1), n), (name(V_2), |M_2^k|)\}$. As V_2 is a leaf ($Children = \emptyset$) then the valuation should satisfy the *var* set of formulae (step `2`). Altogether, the valuations built with the set M_2^k of morphisms commuting with each occurrence p^k of V_1 have to satisfy the equations in *var*. In the figure, we have replicated V_2 and V_1 for each different morphism p_2^k and p_1^k, to show the tree traversal more intuitively.

The right of Fig. 26 shows an example of a model *G* that satisfies the specification of the *Observer* pattern of Fig. 1. The root V_{Ob} occurs once, and the variable part V_{Conc} twice (note that they have been assigned the variables `Ob` and `Conc` in the figure, respectively). The variabilities are checked by building the set $M_{Ob} = \{p^1_{Ob} : V_{ob} \to G\}$, picking its only element p^1_{Ob}, and then building the set $M^1_{Conc} = \{p^1_{Conc} : V_{Conc} \to G, p^2_{Conc} : V_{Conc} \to G\}$. The valuation $\{(Ob, 1), (Conc, |M^1_{Conc}| = 2)\}$ is obtained and therefore $Conc > 1$ holds. In the figure, the dotted lines indicate some of the mappings induced by p^1_{Ob}, p^1_{Conc} and p^2_{Conc}. In the same way, Fig. 3 in Section 3 shows to the left the maximal element of *EXP*(*Observer*) that is present in the graph *G* to its right.

B.2. Pattern invariants

This section gives the details of the formalization of pattern invariants, sketched in Section 4. For this purpose we first introduce the notion of graph constraint, taken from [12].

Definition 9 (*Pattern constraint*). Given a structuring or secondary pattern *VP*, an *atomic pattern constraint* over $V_i \in VP$ is of the form $AC(x_i: V_i \to X, \vee_{n \in N} c_n)$, where x_i and c_n: $X \to C_n$ are triple graph

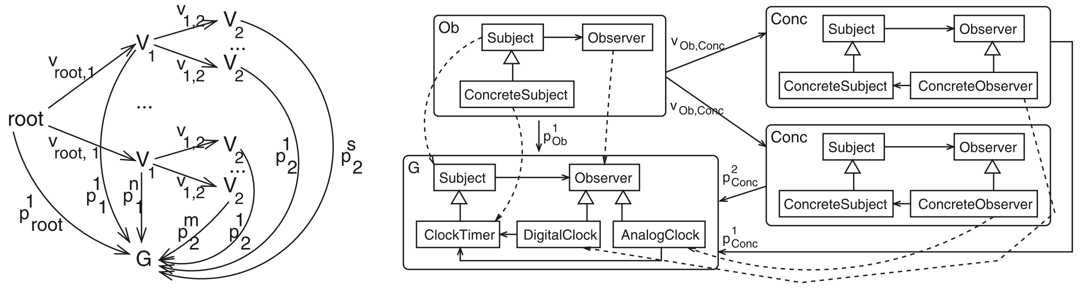


Fig. 26. Scheme for pattern satisfaction (left). Satisfaction example (right).

injective morphisms. A *pattern constraint PC* over V_i is a Boolean formula over atomic pattern constraints over V_i. ■

A constraint over V_i is satisfied if, for each occurrence of *X* commuting with an occurrence of V_i, we find at least one occurrence of some C_i. Thus, constraints can be interpreted as "*if X is found in the graph, then C_1 or C_2 or ... or C_n should be found*". Pattern constraints of the form $AC(x_i, \vee_{n \in N} c_n)$ and $\neg AC(x_i, \vee_{n \in N} c_n)$ with empty index set *N* are abbreviated by $NAC(x_i)$ (for Negative Application Condition) and $PAC(x_i)$ (for Positive Application Condition), respectively. The former express a forbidden structure *X*, while the latter demand finding *X* in the model.

Definition 10 (*Pattern constraint satisfaction*). A triple graph morphism m: $V_i \rightarrow G$ satisfies $AC(x_i\text{: } V_i \rightarrow X, \vee_{n \in N} c_n)$ if for all injective morphisms p: $X \rightarrow G$ with $p \circ x_i = m$ there exists $n \in N$ and an injective morphism q_n: $C_n \rightarrow G$ with $q_n \circ c_n = p$, see the left part of Fig. 27. The satisfaction of a pattern constraint *PC* is as follows. A morphism *m* satisfies $\neg PC$ if it does not satisfy *PC*. *m* satisfies $\wedge_{k \in K} PC_k$ if it satisfies all PC_k. *m* satisfies $\vee_{k \in K} PC_k$ if it satisfies some PC_k. ■

Example. The right part of Fig. 27 shows the invariants for the *Singleton* pattern. The constraints are two NACs that forbid a public and a protected constructor in the singleton (NAC_1 and NAC_2 respectively). These constraints were shown in compact form in Fig. 9. For the sake of illustration, we have included an additional constraint of the form $X \rightarrow C$, which is equivalent to the two NACs. It states that if the singleton defines another constructor different from the one with role `constructor` (*X* graph), then such constructor has to be private (*C* graph).

Satisfaction of patterns with invariants. In order to check the satisfaction of a pattern *VP* with invariants, we modify the generation of its expansion graphs $E \in EXP(VP)$ so that they include copies of the pattern invariants, as we illustrate in Fig. 28. The left of this figure depicts a pattern with two variable parts V_1 and V_2. Its root has a constraint $AC(x_1, c_1)$ stating that, if an `A` object is connected with two `B`'s, then it should be connected with a `C`. Hence, one can expand V_2 as much as one likes, but if V_1 is expanded twice, then V_2 should be expanded at least once in order to obtain the `C` required by the constraint, or else the environment (i.e., the model where the pattern is embedded) should provide a `C` connected to the `A`. Although an equation in *var* could describe a similar constraint, it cannot express the fact that `C` can be given by the context.

The right of Fig. 28 shows the expansion graph *E* that results from expanding V_1 twice. The constraints are transferred from the root to *E* by calculating all surjective glueing graphs $X_{1,i}$ of *E* and X_1 (technically, calculating the set $\{E \xrightarrow{x_{1,i}} X_{1,i} \xleftarrow{f_i} X_1 | f_i, x'_{1,i}$ are jointly surjective$\}$). The same procedure is used to transfer the graph C_1 to the expansion graph *E*. There are three ways to identify the `B`'s of the constraint X_1 with the `B`'s of *E*, and hence three

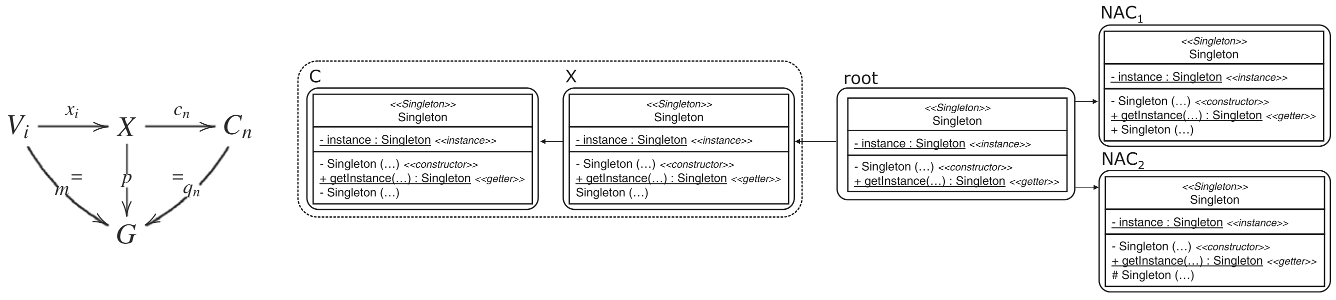


Fig. 27. Scheme for atomic pattern constraint satisfaction (left). Invariants for *Singleton* in theoretical form (right).

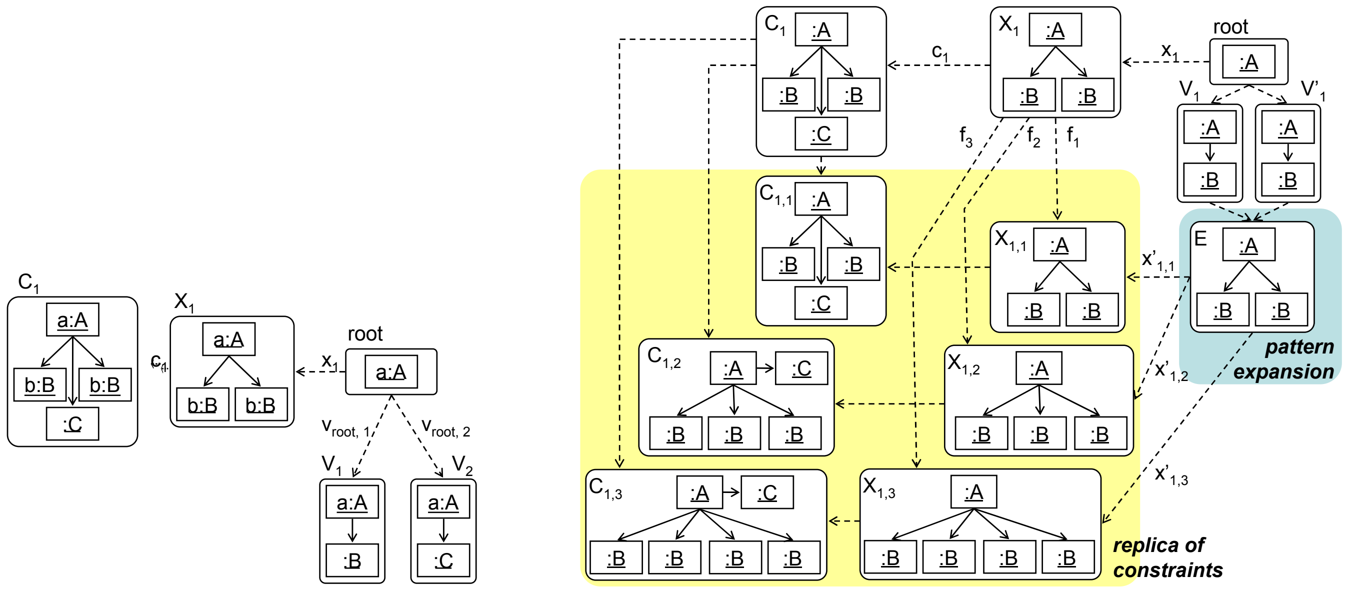


Fig. 28. Example pattern *VP* with constraints (left) and one expansion in the set *EXP(VP)* (right).

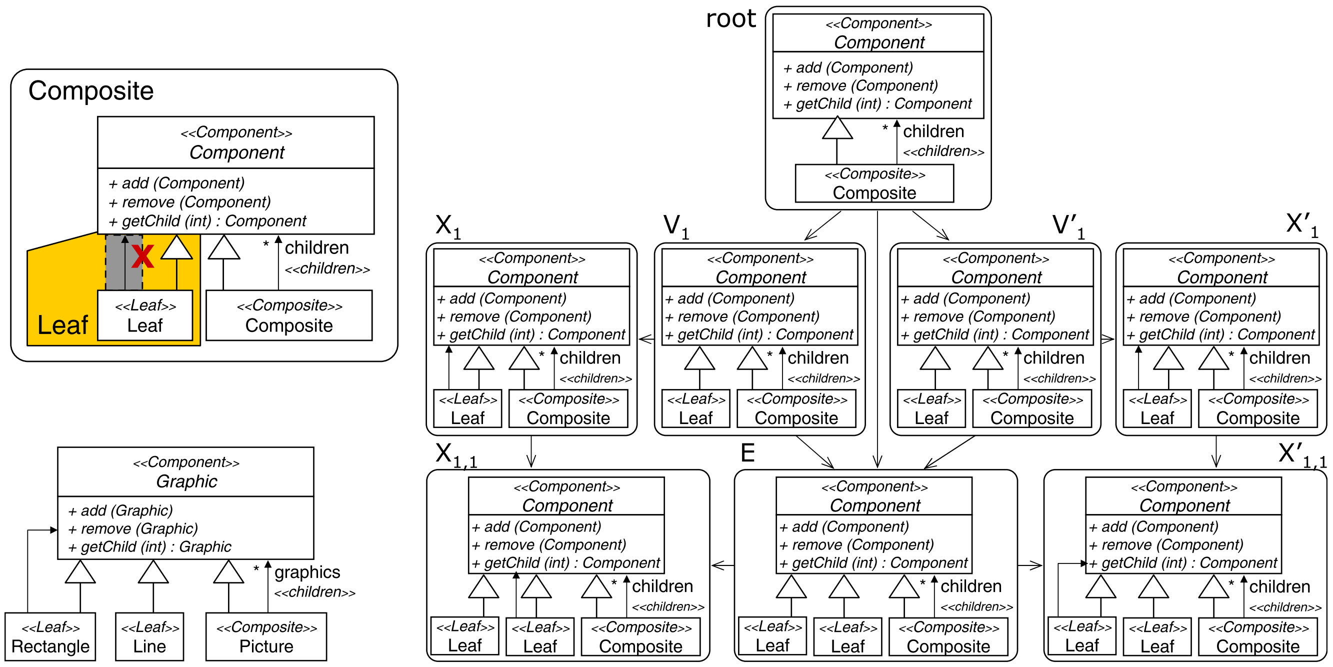


Fig. 29. *Composite* pattern (upper left). One expansion of the pattern (right). A model that does not hard-satisfy the pattern (lower left).

different $X_{1,i}$ are obtained. Then, for each one, there is a unique way to obtain the $C_{1,i}$ graphs. Anyhow even though three different constraints are obtained, $X_{1,1} \rightarrow C_{1,1}$ subsumes the others as it has a weaker premise. Moreover, this condition is not satisfied by the obtained graph E, but the environment has to provide a `C` already connected to the `A`. As stated throughout the paper, we call an expansion that does not satisfy some of its constraints a *weak expansion*, while a *strong expansion* is an expansion that satisfies all constraints. In this example, expanding V_1 twice and V_2 once leads to a strong expansion. Finally, if there is no graph G in which a weak expansion E can be embedded such that its constraints are satisfied, then E is called an *unsatisfiable expansion*. This would be the case if E violates a NAC constraint, because a graph bigger than E would violate it too.

Example. The upper left of Fig. 29 shows the *Composite* pattern. To the right, the figure shows an expansion with two leaves and some replicas of the negative constraint (indeed other two – identical – constraints $X_{1,2}$ and $X'_{1,2}$ are generated with just one leaf, and the two leaves from E identified into it). The lower left shows a model that does not hard-satisfy the pattern because one leaf declares a link to the component. This example was shown in Fig. 10 in compact notation.

B.3. Suggesting tips to enforce invariant satisfaction

This subsection formalizes the procedure described in Section 5.1, which suggests recovery actions when applying one pattern violates the invariants of the pattern definition, or those of other pattern instances.

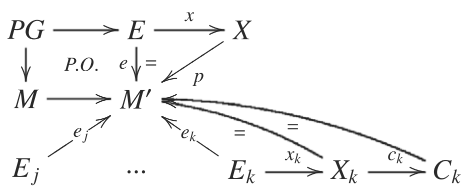

$$\begin{tikzcd}
PG \arrow[r] \arrow[d] \arrow[dr, phantom, "P.O."] & E \arrow[r, "x"] \arrow[d, "e"', "="] & X \arrow[dl, "p"] & & \\
M \arrow[r] & M' & & & \\
E_j \arrow[ur, "e_j"] & \cdots & E_k \arrow[ul, "e_k"'] \arrow[r, "x_k"] & X_k \arrow[ull, bend right, "="'] \arrow[r, "c_k"] & C_k \arrow[ulll, bend right, "="']
\end{tikzcd}$$

Fig. 30. Scheme for suggesting recovery actions.

The first scenario is that of a pattern that cannot be applied because one of its NACs is violated, as Fig. 30 shows. The figure depicts the completion of M according to E to yield M', where $NAC(x)$ is not satisfied because a morphism $p\colon X \rightarrow M'$ exists. The figure also shows other existing pattern instances $E_j, \ldots, E_k$, some of which may have constraints as in the case of E_k. In such scenario, we generate a set of deleting rules [12] or "graph differences" that delete parts of the violated NAC instance and which are presented to the user so that he can select the most appropriate one to enforce the satisfaction of the invariants. In particular, we generate the partial order of all graphs X_j bigger than or equal to E and smaller than X. As these graphs contain the elements to be preserved, those elements that are in X but not in X_j are deleted. In addition, each fragment to be deleted should let intact (i) each pattern occurrence already existing in M' and (ii) each positive constraint of every existing pattern. We do not care about NACs, as deleting elements never violates them. Next we enumerate the steps to build these rules.

1. Build a set $Patt'$ as the union of $Patt$ [2] and the occurrence E with its already satisfied constraints.
2. Take all subgraphs X_j bigger or equal than E and smaller than X, and their induced morphism to M'. We call this set $Pres$. Intuitively, if we delete the elements of X not present in X_j, the constraint $NAC(x)$ will be satisfied as the morphism $p\colon X \rightarrow G$ will no longer exist.
3. Build a set $Pres^P \subseteq Pres$ with the elements $X_j \in Pres$ such that deleting the difference between X and X_j does not destroy any existing pattern instance. For this purpose, iterate on all elements $X_j \rightarrow M' \in Pres$ and do:
   (a) Take each $E_i \rightarrow M' \in Patt'$: if every intersection with $X \rightarrow M'$ is smaller than or equal to X_j (i.e., the occurrence E_i is preserved), then add $X_j \rightarrow M'$ to $Pres^P$. Technically, this is

[2] *Patt* is a set that contains all patterns applied to a model, together with the occurrences of their invariants, as explained in step vi of the algorithm for pattern-based model completion presented in Section 5.

checked by building the pullback $E_i \leftarrow B \xrightarrow{b} X$ of $E_i \rightarrow M' \leftarrow X$ and checking that there is an injection i: $B \rightarrow X_j$ s.t. $i_j \circ i = b$ with i_j: $X_j \rightarrow X$.

4. Build a set $Pres^{PC} \subseteq Pres^P$ with all the elements of $Pres^P$ that preserve all constraints of the pattern occurrences in $Patt'$. For this purpose, we iterate on each element $X_j \rightarrow M' \in Pres^P$ and do:
   (a) For all $E_k \rightarrow M' \in Patt$ having a constraint $E_k \rightarrow X_k \rightarrow C_n$, do:
      i. If some occurrence $X_k \rightarrow M'$ is preserved (see item 3a) but some $C_n \rightarrow M'$ is not, then **break** (that is, take a new $X_j \rightarrow M' \in Pres^P$, as the current one does violate a positive constraint).
   (b) Add $X_j \rightarrow M'$ to $Pres^{PC}$.
5. return $Pres^{PC}$.

The resulting set $Pres^{PC}$ contains the model fragments X_j that can be preserved, and thus the elements of X not in X_j can be deleted. This ensures that the pattern occurrence E satisfies all invariants once one of the proposed deletions is performed. Conceptually, the set contains deleting graph transformation rules of the form $X \Rightarrow X_j$ (with meaning: "*if X is found in the model, replace it by X_j*"). Additionally, we should check that the deletions leave the model in a consistent state, that is, do not violate the language meta-model constraints. The set $Pres^{PC}$ is indeed a partial order, so in practice a tool would propose first the smaller deletions (bigger X_j) and later the bigger ones (smaller X_j, E in the limit).

The second scenario in which a pattern cannot be applied because a constraint $AC(x$: $E \rightarrow X, \vee_{n \in N} c_n$: $X \rightarrow C_n)$ is not satisfied is similar, but in that case a set of non-deleting graph transformation rules is built.

B.4. Pattern composition

Next we present some formal details of our two ways of composing patterns, which were introduced in Section 6. The first one yields a new pattern where a new role is created for two elements that are identified together. The second method is like a "macro" that allows applying two patterns in one step.

The left of Fig. 31 illustrates the first case. Given two variable parts (in this case the roots) of two patterns to be composed, the user selects the elements in $root^1$ and $root^2$ that are to be identified. Formally, this is modelled by an intersection graph K and morphisms to $root^1$ and $root^2$, $root^1 \leftarrow K \rightarrow root^2$. Then, the pushout is built yielding $root'$. If the two original patterns contain constraints, we calculate all jointly surjective graphs from the constraint and $root'$ (similar to the procedure for adding constraints to the expansion in Fig. 28) and the new pattern incorporates the conjunction of all resulting constraints. In the figure, the conjunction of the two constraints X' and X'' is added to the composed $root'$. Concerning the variability equations, both merged variable parts receive the same name, hence we perform the union of the original equations (once we do the renaming) and consider the most restrictive ones.

The second composition operation does not generate a new pattern, but the operation retains the roles of the original elements in the two patterns. The procedure is the same, but the roles are not identified when specifying the intersections of the variable parts. The right of Fig. 31 illustrates the difference with the previous composition mechanism. In the diagram to the left, two elements `c2` and `c3` in two patterns are identified in the composition (signalled by element `c` in the intersection and the morphisms (`c`, `c2`) and (`c`,`c3`) yielding element `c23`. In this case, the roles are also identified and a new role `r23` is created. In the diagram to the right, the roles are not identified in the intersection, and the original roles are retained in the composition.

In case the patterns have additional variability parts, the algorithm proceeds by constructing the pushouts of the variable parts, according to the identified elements in these. The universal property of the pushout [25] allows finding the embedding of the composed variability regions into one another.

Example. Fig. 32 shows the composition of the *Composite* and the *Observer* patterns, which was shown in Fig. 15 in compact form. In the composition, we have chosen to keep the roles in both patterns, and to identify the `Observer` with the `Component` in the root (given by graph K) and the `Leaf` and the `ConcreteObserver` in the variable parts (given by graph K'). The intersection of the variable parts must be coherent with that of the roots, which is given by the injection $K \rightarrow K'$. Regarding the equations, assume that the first pattern has $\{V_1^{Observer} > 0\}$, while the second has $\{V_1^{Composite} > 1\}$. Then the composition has the union of both sets,

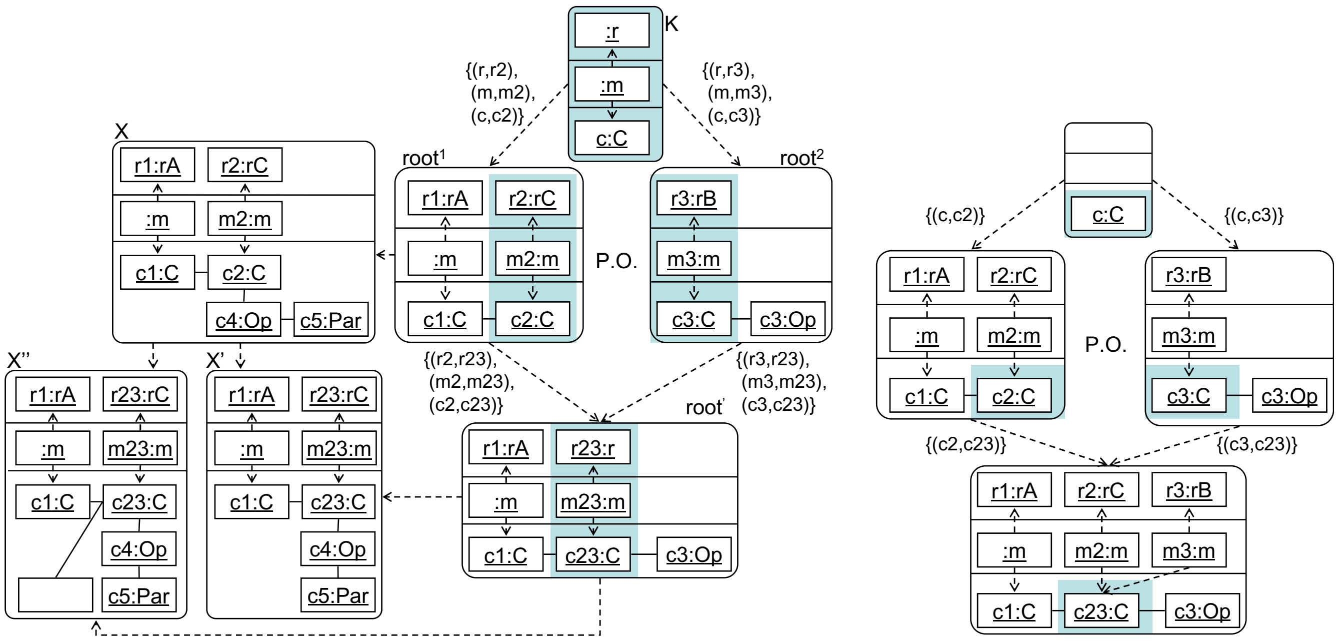


Fig. 31. Composition identifying roles (left). Composition without identification of roles (right).

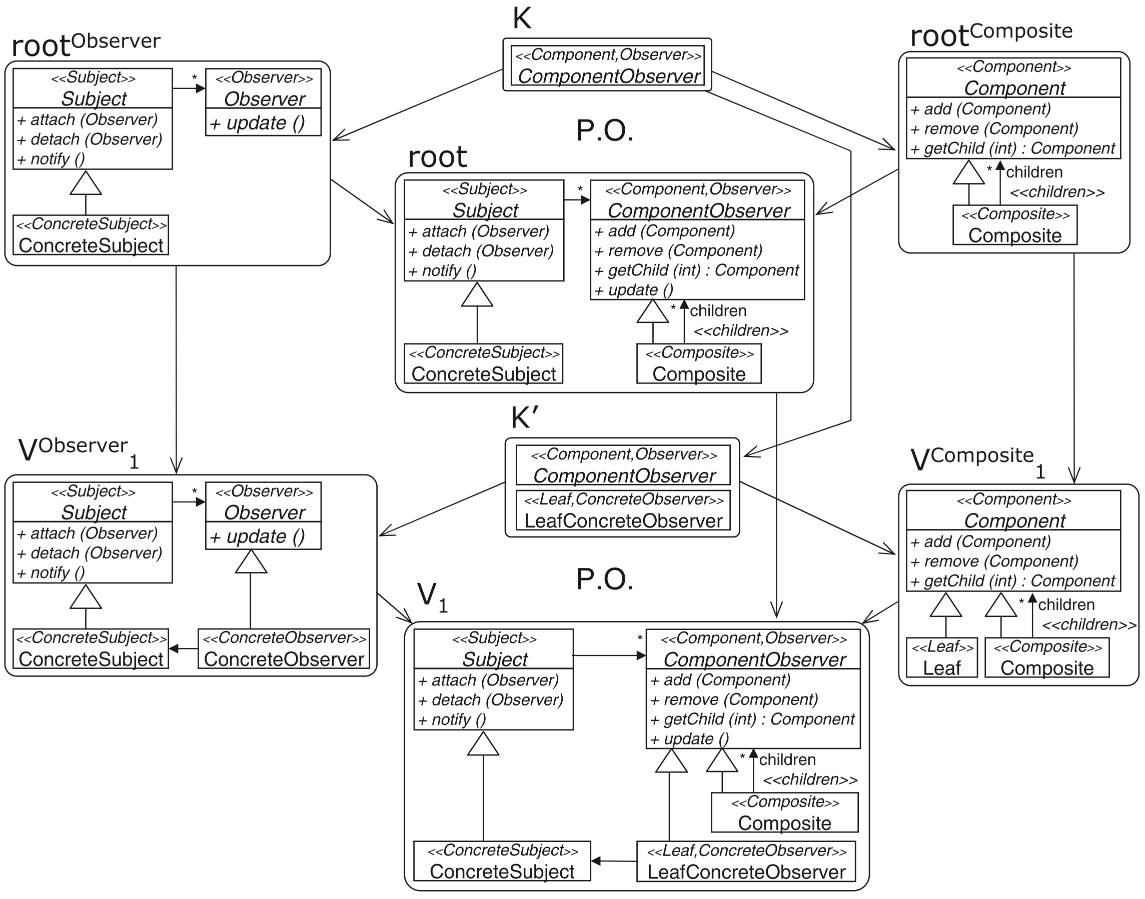


Fig. 32. Composing the *Observer* and *Composite* patterns.

once variables are renamed, $var' = \{V_1 > 0, V_1 > 1\}$. This results in taking V_1 as the most restrictive ($V_1 > 1$). Finally, note that a morphism $root \rightarrow V$ is automatically obtained by the universal pushout property, so that the pattern $root \rightarrow V$ is obtained as the result of the composition.

References

[1] C. Alexander, A Pattern Language: Towns, Buildings, Construction, Oxford University Press, 1977.
[2] J. Arlow, I. Neustadt, Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML, Addison-Wesley Object Technology Series (2004).
[3] K. Arnout, B. Meyer, Pattern componentiation: the factory example, ISSE 2 (2) (2006) 65–79.
[4] I. Bayley, H. Zhu, On the composition of design patterns, in: Proceedings of the QSIC 2008, IEEE Computer Society, 2008, pp. 27–36.
[5] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, G. Varró, Incremental pattern matching in the VIATRA model transformation system, in: Proceedings of GRaMoT '08, ACM, 2008, pp. 25–32.
[6] J. Bézivin, F. Jouault, J. Palies, Towards model transformation design patterns, in: Proceedings of EWMT 2005, 2005.
[7] P. Bottoni, E. Guerra, J. de Lara, Formal foundation for pattern-based modelling, in: M. Chechik, M. Wirsing (Eds.), FASE, LNCS, vol. 5503, Springer, 2009, pp. 278–293.
[8] P. Bottoni, E. Guerra, J. de Lara, A formalization of the GoF design patterns, 2010. <http://arxiv.org/abs/1003.3338> arXiv:1003.3338v1.
[9] J. de Lara, H. Vangheluwe, AToM^3: a tool for multi-formalism and meta-modelling, in: R.-D. Kutsche, H. Weber (Eds.), FASE, LNCS, vol. 23, Springer, 2002, pp. 174–188.
[10] J. Dong, S. Yang, K. Zhang, Visualizing design patterns in their applications and compositions, IEEE Trans. Soft. Eng. 33 (7) (2007) 433–453.
[11] H. Ehrig, K. Ehrig, A. Habel, K.-H. Pennemann, Theory of constraints and application conditions: from graphs to high-level structures, Fundam. Inform. 74 (1) (2006) 135–166.
[12] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, Springer, 2006.
[13] R.B. France, D.-K. Kim, S. Ghosh, E. Song, A UML-based pattern specification technique, IEEE Trans. Soft. Eng. 30 (3) (2004) 193–206.
[14] E. Gamma, R. Helm, R. Johnson, J.M. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.
[15] Y.-G. Guéhéneuc, H. Albin-Amiot, Using design patterns and constraints to automate the detection and correction of inter-class design defects, in: Proceedings of TOOLS (39), IEEE Computer Society, 2001, pp. 296–306.
[16] E. Guerra, J. de Lara, Model view management with triple graph transformation systems, in: ICGT, LNCS, vol. 4178, Springer, 2006, pp. 351–366.
[17] E. Guerra, J. de Lara, Event-driven grammars: relating abstract and concrete levels of visual languages, Software Syst. Model. 6 (3) (2007) 317–347.
[18] G. Hohpe, B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2004.
[19] P. Hruby, Model-Driven Design Using Business Patterns, Springer, 2006.
[20] H. Kampffmeyer, S. Zschaler. Finding the pattern you need: the design pattern intent ontology, in: MoDELS, LNCS, vol. 4735, Springer, 2007, pp. 211–225.
[21] J. Kerievsky, Refactoring to Patterns, Addison-Wesley Signature Series, 2004.
[22] S.K. Kim, D. Carrington, A formalism to describe design patterns based on role concepts, Form Aspects Comput. 21 (5) (2009) 397–420.
[23] A. Lauder, S. Kent, Precise visual specification of design patterns, in: ECOOP, 1998, pp. 114–134.
[24] J. Lindqvist, T. Lundkvist, I. Porres, A query language with the star operator, ECEASST 6 (2007).
[25] S. Mac Lane, Categories for the Working Mathematician, Graduate Texts in Mathematics, second ed., vol. 5, Springer, 1998.
[26] J.K.-H. Mak, C.S.-T. Choy, D.P.-K. Lun. Precise modeling of design patterns in UML, in: ICSE, 2004, pp. 252–261.
[27] D. Maplesden, J. Hosking, J. Grundy, A visual language for design pattern modelling and instantiation, in: T. Taibi (Ed.), Design Patterns Formalization Techniques, Idea Group Inc., 2007, pp. 20–43 (Chapter 2).
[28] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, J. Welsh. Towards pattern-based design recovery, in: ICSE, 2002, pp. 338–348.
[29] A. Radermacher, Support for design patterns through graph transformation tools, in: AGTIVE LNCS, vol. 1779, Springer, 1999, pp. 111–126.
[30] T. Taibi, D.C.L. Ngo, Formal specification of design pattern combination using BPSL, Inf. Soft. Technol. 45 (2003) 157–170.
[31] J. Tidwell, Designing Interfaces, O'Reilly, 2006.

[32] T. Tourwé, T. Mens, High-level transformations to support framework-based software development, ENTCS 72 (4) (2003).
[33] J. Ullmann, An algorithm for subgraph isomorphism, J. ACM 23 (1) (1976) 31–42.
[34] UsiXML. UsiXML, user interface extensible markup language, 2007. <http://www.usixml.org/index.php?mod=download&file=usixml-doc/UsiXML_v1.8.0-Documentation.pdf>.
[35] W. van der Aalst, A. ter Hoefstede, B. Kiepuszewski, A. Barros, Workflow patterns, Distrib. Parallel DataBases 14 (3) (2003) 5–51.
[36] S. Yacoub, H. Ammar, UML support for designing software systems as a composition of design patterns, in: Proc. UML 2001, Springer-Verlag, 2001, pp. 149–165.
[37] C. Zhao, J. Kong, J. Dong, K. Zhang, Pattern-based design evolution using graph transformation, J. Vis. Lang. Comput. 18 (4) (2007) 378–398.