Electronic Communications of the EASST
Volume 31 (2010)

Proceedings of the
Second International Workshop on
Visual Formalisms for Patterns
(VFfP 2010)

Enforcement of Patterns by Constraint-Aware Model Transformations

Yngve Lamo, Adrian Rutle and Florian Mantz

12 pages

Guest Editors: Paolo Bottoni, Esther Guerra, Juan de Lara
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Enforcement of Patterns by Constraint-Aware Model Transformations

Yngve Lamo[1], **Adrian Rutle**[1] **and Florian Mantz**[1]

[1] yla,aru,fma@hib.no, http://www.hib.no *
Department of Computer Engineering
Bergen University College, Norway

Abstract: Patterns are descriptions and solutions for recurring problems in software design and implementation. In this paper, some ideas towards a formal approach to the specification of patterns in model-driven engineering (MDE) is presented. The approach is based on the Diagram Predicate Framework which provides a formal approach to (meta)modelling, model transformation and model management in MDE. In particular, patterns are defined as diagrammatic specifications and constraint-aware model transformations are adapted to enforce patterns. Moreover, running examples are used to illustrate the facade design pattern in structural models.

Keywords: Pattern, constraint-awareness, model transformation, model refactoring, Diagram Predicate Framework

1 Introduction and motivation

Since the beginning of computer science, developing high-quality software at low cost has been a continuous vision. This has boosted several shifts of programming paradigms, e.g. machine code to compilers and imperative to object-oriented programming. In every shift of paradigm, raising the abstraction level of programming languages and technologies has proved to be beneficial to increase productivity. One of the latest steps in this direction has lead to the usage of models and modelling languages in software development processes.

Initially, models were adopted in software development processes for sketching the architectural design or documenting an existing implementation. In the latest trend in software engineering, however, models are regarded as first-class entities of the development process. These models are used to automatically generate (parts of) software systems by means of model-to-model and model-to-code transformations. In the literature, this trend is referred to as model-driven engineering (MDE).

Software development projects have traditionally been built following the waterfall approach, a sequential process consisting of requirements specification, design, implementation, testing, deployment and maintenance phases. Often a new project was started from scratch by designing the domain model, architecture model, code etc. without any clear methodology or systematic use of earlier experiences. In this context, natural questions which arise are: What is software quality? What is best practice in software engineering? What is a good design? What is an appropriate architecture for a certain kind of software systems? What is a good piece of software?

* The research is partially sponsored by the Norwegian Research Council project FORMGRID.

To address the problem with software quality, in the late eighties Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to software engineering [BC87]. Moreover, the seminal book [GHJV94] on design patterns published in 1994 by the so-called "Gang of Four" had a great influence on software development practise. Design patterns are usually used as a solution strategy for a common problem, e.g. facade, decorator, singleton, etc, and often describe a solution for a part of a bigger system. Although design patterns have been applied in software development for a long time, formalisation of the concept of patterns is still an open research topic [BGL09]. Moreover, patterns are usually explained in a semi formal or informal language.

In MDE the process of developing software is performed by use of (semi-)automatic development steps in form of model transformations. Hence, to fully benefit from patterns in MDE the patterns should be expressed formally, facilitating model transformations and automatic software development steps. In MDE, patterns are used in different phases and for different means during the software development process:

- means for communication, e.g. among developers and domain experts
- guideline for design; i.e. as a specification for software design and software behaviour
- tool for conformance check; i.e. to check whether a model follows a given pattern or not
- guideline for design change; i.e. if the design does not follow the desired pattern, the pattern may be forced by use of model transformations and refactoring.

To be practically useful, patterns in MDE should meet some criteria, e.g they should be formal, abstract, conceptually clear, intuitive, adaptable and reusable. To enhance usability of patterns, it is natural to employ a diagrammatic approach, but still demanding a precise (formal) meaning of the diagrammatic models. The proposed approach of this paper is based on the Diagram Predicate Framework (DPF) [Rut10, RRLW10, RRLW09], which is a generalisation and adaptation of the categorical sketch formalism [BW95], where user-defined diagrammatic predicate signatures represent the constructs of modelling languages in a more direct way. In particular, DPF is an extension of the Generalised Sketches [Mak97] formalism [Dis03]. DPF aims to combine mathematical rigour – which is necessary to enable automatic reasoning – with diagrammatic modelling.

In this paper, we use DPF to formalise concepts related to patterns in MDE. We will define these concepts in general in the sense that they may be applied for design patterns or other kinds of patterns such as input and output patterns of model transformation rules.

Usually patterns describe the structure of an architecture or a problem solution for a (sub)system. In MDE a pattern could be represented by a structural model. To ensure the desired behavior of the system the pattern should also have the possibility to describe some of the constraints that the system needs to fulfil. Hence a proper formalisation of patterns should also have the possibility to express actual constraints.

The remainder of the paper is structured as follows. Section 2 outlines DPF as the formal underpinning of our approach. Section 3 introduces the formal approach to patterns and pattern enforcement. In Section 4, some related research in patterns within MDE is presented. Finally, in Section 5, some concluding remarks and ideas for future work are presented.

2 Diagram Predicate Framework

DPF is a generic graph-based specification framework that tends to adapt first-order logic and categorical logic to software engineering needs. DPF is generic in the sense that it supports any kind of graph structures (see [DW08] for the general case). However, the variant of DPF which we employ in this paper is based on directed multi-graphs.

Before introducing the formal foundation of DPF, the terminology adopted in this paper is clarified in the following. The word "model" has different meanings in different contexts. In software engineering, model denotes "an abstraction of a (real or language-based) system allowing predictions or inferences to be made" [Küh06]. Models in software engineering are typically diagrammatic. The word "diagram" has also different meanings in different contexts. In software engineering, diagram denotes a structure which is based on graphs; i.e. a collection of nodes together with a collection of arrows between nodes. Since graph-based structures can be visualised in a natural way, "visual" and "diagrammatic" modelling are often treated as synonyms. In this paper, visualisation and diagrammatic syntax are clearly distinguished. That is, the proposed approach focuses on precise syntax and semantics of diagrammatic models independent of their visualisation.

In DPF, models are represented by *(diagrammatic) specifications*. A specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ consists of an underlying graph S together with a set of *atomic constraints* $C^{\mathfrak{S}}$ [RRLW09, Rut10]. The graph represents the structure of the model while atomic constraints add restrictions to this structure. Atomic constraints are formulated by predicates from *(diagrammatic predicate) signatures*. A signature $\Sigma = (P^{\Sigma}, \alpha^{\Sigma})$ consists of a collection of predicates, each having a name, a shape graph, a visualisation and a semantic interpretation [RRLW09, Rut10]. The formal definitions are as follows:

Definition 1 (Signature) A signature $\Sigma = (P^{\Sigma}, \alpha^{\Sigma})$ consists of a collection of predicate symbols P^{Σ} with a mapping α^{Σ} that assigns a graph to each predicate symbol $p \in P^{\Sigma}$. $\alpha^{\Sigma}(p)$ is called the *arity* of the predicate symbol p.

Definition 2 (Atomic Constraint) Given a signature $\Sigma = (P^{\Sigma}, \alpha^{\Sigma})$, an atomic constraint (p, δ) on a graph S is given by a predicate symbol p and a graph homomorphism $\delta : \alpha^{\Sigma}(p) \to S$ [1].

Definition 3 (Specification) Given a signature $\Sigma = (P^{\Sigma}, \alpha^{\Sigma})$, a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ is given by a graph S and a set $C^{\mathfrak{S}}$ of constraints (p, δ) on S with $p \in P^{\Sigma}$.

Definition 4 (Specification Morphism) Given two specifications $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ and $\mathfrak{S}' = (S', C^{\mathfrak{S}'} : \Sigma)$, a specification morphism $\phi : \mathfrak{S} \to \mathfrak{S}'$ is a graph homomorphism $\phi : S \to S'$ such that $(p, \delta) \in C^{\mathfrak{S}}$ implies $(p, \delta;\phi) \in C^{\mathfrak{S}'}$, illustrated by the following diagram:

$$\begin{tikzcd}
\alpha^{\Sigma}(p) \arrow[r, "\delta"] \arrow[rr, bend left, "\delta;\phi"] \arrow[rr, phantom, bend left=15, "="] & S \arrow[r, "\phi"] & S'
\end{tikzcd}$$

Nodes and arrows of a specification have to be interpreted in a way which is appropriate for

[1] The definition of atomic constraint corresponds to diagrams in category theory.

Table 1: A sample signature $\Sigma = (P^{\Sigma}, \alpha^{\Sigma})$

p	$\alpha^{\Sigma}(p)$	**Proposed vis.**	**Semantic interpretation**
`[mult(`m,n`)]`	$1 \xrightarrow{a} 2$	$\boxed{\mathsf{X}} \xrightarrow[\text{[m..n]}]{\mathsf{f}} \boxed{\mathsf{Y}}$	$\forall x \in X : m \leq \lvert f(x) \rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
`[injective]`	$1 \xrightarrow{a} 2$	$\boxed{\mathsf{X}} \xrightarrow[\text{[inj]}]{\mathsf{f}} \boxed{\mathsf{Y}}$	$\forall x, x' \in X : f(x) = f(x')$ implies $x = x'$
`[surjective]`	$1 \xrightarrow{a} 2$	$\boxed{\mathsf{X}} \xrightarrow[\text{[surj]}]{\mathsf{f}} \boxed{\mathsf{Y}}$	$\forall x \in X : \bigcup\{f(x)\} = Y$
`[inverse]`	$$\begin{tikzcd} 1 \arrow[r, bend left, "a"] & 2 \arrow[l, bend left, "b"] \end{tikzcd}$$	$$\begin{tikzcd} \boxed{\mathsf{X}} \arrow[r, bend left, "\mathsf{f}"] \arrow[r, phantom, "\text{[inv]}"] & \boxed{\mathsf{Y}} \arrow[l, bend left, "\mathsf{g}"] \end{tikzcd}$$	$\forall x \in X, \forall y \in Y : y \in f(x)$ iff $x \in g(y)$
`[composition]`	$$\begin{tikzcd} 1 \arrow[r, "f"] \arrow[dr, "h"'] & 2 \arrow[d, "g"] \\ & 3 \end{tikzcd}$$	$$\begin{tikzcd} \boxed{\mathsf{X}} \arrow[r, "\mathsf{f}"] \arrow[dr, "\text{[comp(f,g)]}"'] & \boxed{\mathsf{Y}} \arrow[d, "\mathsf{g}"] \\ & \boxed{\mathsf{Z}} \end{tikzcd}$$	$\forall x \in X : h(x) = \bigcup\{g(y) \mid y \in f(x)\}$

the corresponding modelling environment [RRLW09]. In object-oriented structural modelling, each object may be related to a set of other objects. Hence, it is appropriate to interpret nodes as sets and arrows $X \xrightarrow{f} Y$ as multi-valued functions $f : X \to \wp(Y)$. The powerset $\wp(Y)$ of Y is the set of all subsets of Y; i.e. $\wp(Y) = \{A \mid A \subseteq Y\}$. Moreover, the composition of two multi-valued functions $f : X \to \wp(Y)$, $g : Y \to \wp(Z)$ is defined by $(f;g)(x) := \bigcup\{g(y) \mid y \in f(x)\}$.

Example 1 (Sample signature and specification) *Table 1 shows a small sample signature $\Sigma = (P^{\Sigma}, \alpha^{\Sigma})$. Fig. 1a shows a sample diagrammatic specification $(S, C^{\mathfrak{S}} : \Sigma)$ and Fig. 1b shows the underlying graph S of $\mathfrak{S}$; i.e. the graph of $\mathfrak{S}$ without any constraints. In $\mathfrak{S}$, the nodes* X *and* Y *are interpreted as sets X and Y, and the arrows* f *and* g *are interpreted as multi-valued functions $f : X \to \wp(Y)$ and $g : Y \to \wp(X)$, respectively. Moreover, the function g is surjective; this is forced by the constraint* (`[surjective]`$, \delta_1$) *on the arrow* g*. Similarly, the function f is total; this is forced by the constraint* (`[mult(`$1, \infty$`)]`$, \delta_3$) *on the arrow* f*. Finally, the functions f and g are inverse of each other; i.e. $\forall x \in X$ and $\forall y \in Y : x \in g(y)$ iff $y \in f(x)$. This is forced by the constraint* (`[inverse]`$, \delta_2$) *on* f *and* g*. The graph homomorphisms δ_1, δ_2 and δ_3 are defined as follows:*

$$
\begin{array}{llll}
\delta_1(1) = Y, & \delta_1(2) = X, & \delta_1(a) = g & \\
\delta_2(1) = X, & \delta_2(2) = Y, & \delta_2(a) = f, & \delta_2(b) = g \\
\delta_3(1) = X, & \delta_3(2) = Y, & \delta_3(a) = f &
\end{array}
$$

In DPF, we use specifications to represent models at any level of a metamodelling hierarchy. Moreover, we distinguish between two types of relations between models and metamodel: *typed by* and *conforms to*. A specification $\mathfrak{S}_n$ at level n is typed by a specification $\mathfrak{S}_{n+1}$ at level $n+1$ if there exists a typing morphism $\iota^{S_n} : S_n \to S_{n+1}$ between the underlying graphs of the

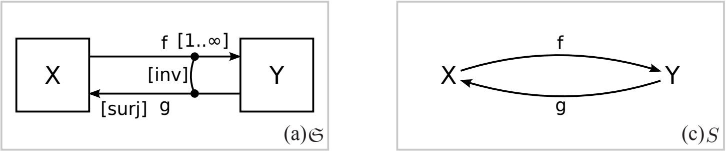

Figure 1: A sample specification (a) $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ and (b) its underlying graph S

specifications. This corresponds to the relation between a model and its metamodel in the graph-based formalisation of the metamodelling hierarchy. In contrast, a specification $\mathfrak{S}_n$ at level n is said to conform to a specification $\mathfrak{S}_{n+1}$ at level $n+1$ if there exists a typing morphism $\iota^{S_n} : S_n \to S_{n+1}$ such that (S_n, ι^{S_n}) is an instance of $\mathfrak{S}_{n+1}$ [Rut10]. That is, in addition to the existence of the typing morphism ι^{S_n}, the constraints $C^{\mathfrak{S}_{n+1}}$ are satisfied by (S_n, ι^{S_n}).

So far we have discussed two concepts for constraining specifications: typing and satisfaction of atomic constraints. These concepts are used to define the relation between models and metamodel. In addition to the conformance requirement, there are other constraints concerning the overall structure of specifications. An example is if one wants to formulate that in EMF models "every model must have a root class" and "every class in a model must have the root class as its container, directly or transitively". In DPF such constraints are expressed by *universal constraints* [Rut10]. A universal constraint is defined as a specification morphism $u : \mathfrak{L} \to \mathfrak{R}$, where $\mathfrak{L}$ and $\mathfrak{R}$ are the premise and the conclusion of the constraint, respectively. A universal constraint is satisfied by a specification if for any occurrence of the premise an occurrence of the conclusion should also be found.

The formal definitions of instance, typed specification and typed specification morphism, conformant specification as well as universal constraints are given in [Rut10].

3 Patterns in DPF

Is this section patterns and pattern matching are formally defined. First a declarative definition of patterns by means of metamodels is given. We also illustrate how patterns (and anti patterns) may be used for model refactoring. Pattern enforcement is performed operationally by means of model transformation from a precondition (anti-pattern) in a specification to the desired pattern.

3.1 Patterns

In MDE metamodelling is used for the definition of modelling languages. Following this line, we define patterns by metamodels in DPF. That is, a pattern may be seen as an instance of a diagrammatic specification representing the pattern's metamodel. Pattern matching is used to show which parts of a model conform to the metamodel of a given pattern. A match of a pattern in a specification is formalised by a specification morphism. Matches may be used to check if a given design follows the desired pattern or not, i.e. pattern finding.

Definition 5 (Pattern) A pattern $\mathfrak{P}$ is a diagrammatic specification typed over a metamodel $\mathfrak{M}$.

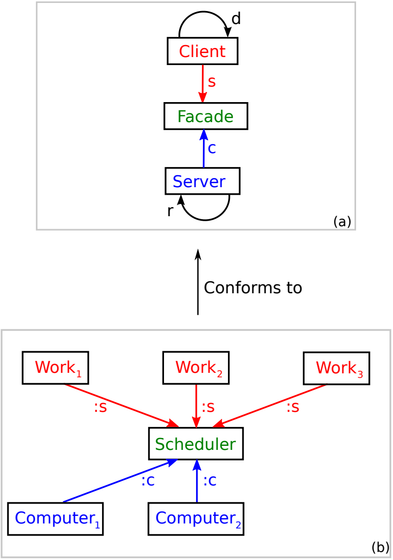

Figure 2: Metamodel for (a) the facade design pattern and (b) and example of a model following the pattern/metamodel

Definition 6 (Match of Pattern) Given a pattern $\mathfrak{P}$ typed over $\mathfrak{M}$ and a specification $\mathfrak{S}$, a match $m : \mathfrak{P} \to \mathfrak{S}$ of the pattern $\mathfrak{P}$ in $\mathfrak{S}$ is a specification morphism $m : \mathfrak{P} \to \mathfrak{S}$. The specification $\mathfrak{S}$ *follows* a pattern $\mathfrak{P}$ if there exists a match $m : \mathfrak{P} \to \mathfrak{S}$ such that $m(\mathfrak{P})$ conforms to $\mathfrak{M}$.

Example 2 (Facade Design Pattern) *The facade element in the facade design pattern serves as an interface for subsystems. That is, if a system consists of several subsystems which are interacting with each other, the facade design pattern should be followed. In Fig. 2a the metamodel for the facade pattern is given. A model following this pattern can be seen in Fig. 2b. The model is a snapshot of a scenario illustrating how certain works are deployed on some computers. There are three Client elements Work$_1$, Work$_2$, Work$_3$ communicating with two Server elements Computer$_1$, Computer$_2$ via the Facade element Scheduler.*

3.2 Pattern Enforcement

If a pattern $\mathfrak{P}$ is not followed by a specification $\mathfrak{S}$, then the pattern may be enforced by applying a model transformation to $\mathfrak{S}$. The precondition for the enforcement of a pattern will then be the source of a model transformation. This is also known as *anti-pattern*, i.e. a design which is not desired and need to be refactored. An anti-pattern is also defined as a pattern (see Fig. 3 for an anti-pattern for the pattern in Fig. 2).

A *model transformation* is the automatic generation of target models from source models, according to a transformation definition. A *transformation definition* is a set of transformation rules that together describe how a source model can be transformed into a target model.

Definition 7 (Model Transformation Rule) A model transformation rule r is given by a spe-

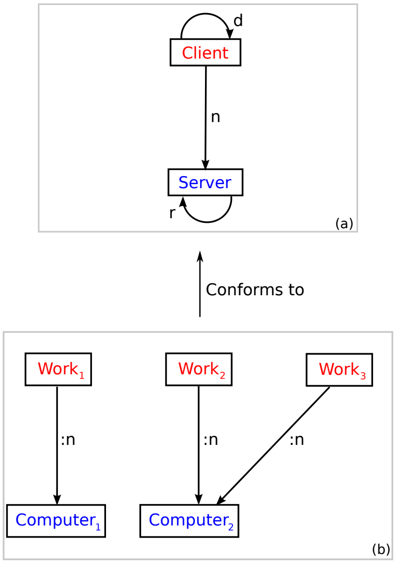

Figure 3: Metamodel for (a) anti patterns and (b) an example of anti pattern

cification morphism $r : \mathfrak{L} \to \mathfrak{R}$. An application of a model transformation rule r is given by a pushout construction, see e.g. [BW95] for a definition of pushout.

$$\begin{tikzcd}
\mathfrak{L} \arrow[r, "r"] \arrow[d, "m"'] \arrow[dr, phantom, "P.O."] & \mathfrak{R} \arrow[d, "m^*"] \\
\mathfrak{S} \arrow[r, "\langle r{,}m\rangle"'] & \mathfrak{S}^*
\end{tikzcd}$$

where for each match $m : \mathfrak{L} \to \mathfrak{S}$, a match $m^* : \mathfrak{R} \to \mathfrak{S}^*$ is created.

Definition 8 (Pattern enforcement) Given a pattern $\mathfrak{P}$ and a specification $\mathfrak{S}$, $\mathfrak{P}$ may be enforced in $\mathfrak{S}$ by performing a model transformation T such that $T(\mathfrak{S})$ follows the pattern $\mathfrak{P}$.

The following example shows the enforcement of a pattern by applying model transformation.

Example 3 (Pattern Enforcement) *Building upon Example 2. The example illustrates how to refactor a client-server architecture such that it follows the facade design pattern. Given the specification in Fig. 4b, a model transformation enforces the facade design pattern and creates the design in Fig. 4c. The model transformation consists of four rules:*

1. *There should be exactly one Scheduler*

2. *Every connection between a Work and a Computer is rerouted via the Scheduler*

3. *There should not be more than one connection between a Computer and the Scheduler*

4. *There should not be more than one connection between a Work and the Scheduler*

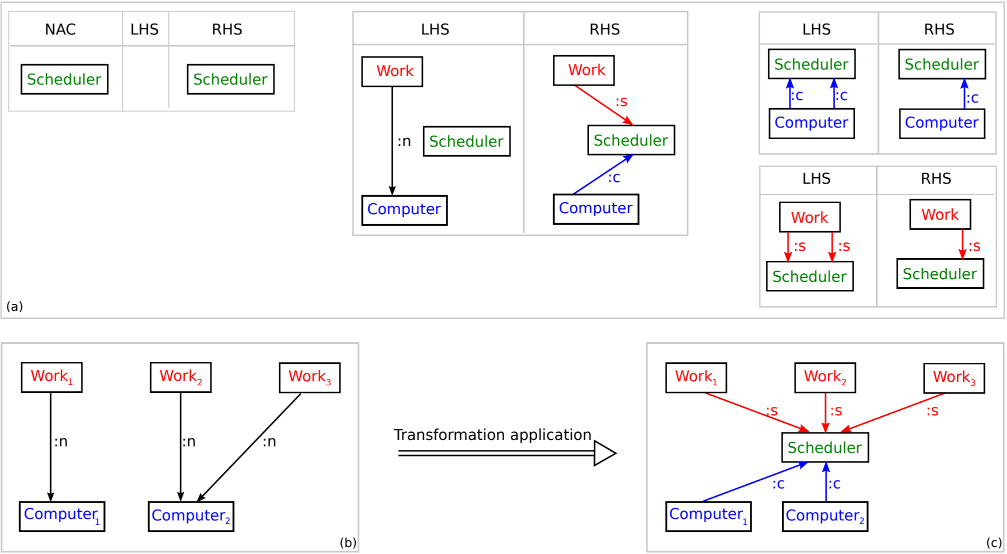

Figure 4: Using transformation rules (a) to enforce the pattern in Fig. 2, a model (b) with an anti pattern in it is refactored to a model (c) following the required pattern

3.3 Constraint-Aware Pattern Enforcement

The next step is to take constraints into account while defining, matching and enforcement of patterns. When a constraint-aware pattern is enforced, the constraints of the original specification should be transformed into corresponding constraints in the refactored specification. One way to achieve this is to use constraint-aware model transformations as described in [RRLW10]. In this approach, constraints of the source models are used to control which structures and which constraints should be defined in the target model. Adding constraints to patterns can also be used to specify the intended behavior of the design. This section outlines how constraint-aware patterns are formalised and enforced in view of DPF.

Example 4 (Constraint-Aware Patterns) *Building on example 3, in Fig. 5 we add a constraint to the pattern metamodel in Fig. 3. The constraint expresses a requirement that if a client c_1 is dependent on a client c_2, the host server(s) of c_1 should reach the host server(s) of c_2. This requirement is expressed by the constraint $comp(d,s) \subseteq comp(s,r)$. That is, the result of the composition of d with s should be included in the results of the composition of s with r.*

Constraints at a modelling level may be expressed as universal constraint on the level below.

Example 5 (Universal Constraints) *The constraint in example 4 is expressed as a universal constraint in Fig. 6A. The specification in Fig. 7a satisfies this constraint since for any occurrance of the premise (Fig. 6a) an occurrance of the conclusion (Fig. 6b) is also found. That is, if Work$_1$ is dependent on Work$_2$ then Computer$_1$ hosting Work$_1$ should be able to reach the computer hosting Work$_1$.*

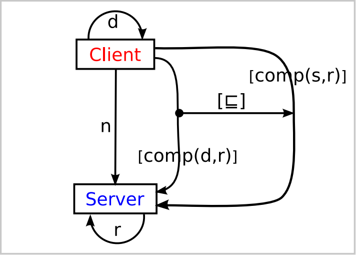

Figure 5: Metamodel of the anti-pattern from Fig. 3a extended with constraints

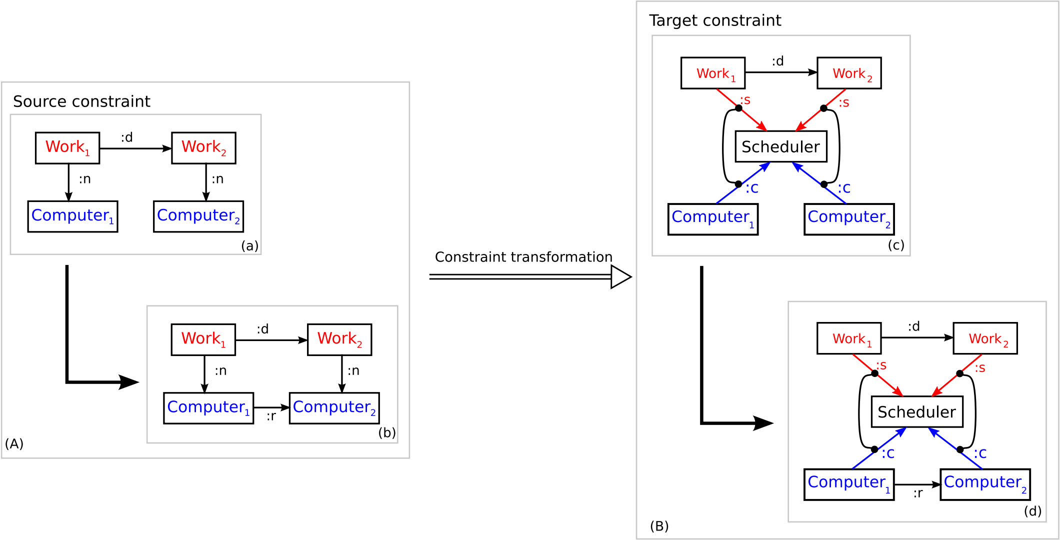

Figure 6: Transformation of (A) source universal constraints to (B) target universal constraints

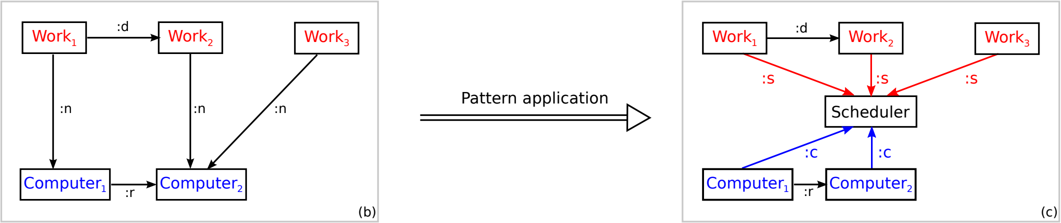

Figure 7: A model (a) following an anti pattern is refactored to a model (b) following the required pattern

Example 6 (Transformation of Universal Constraint) *Building on Example 3. Fig. 7a shows a specification with a match of the anti-pattern in Fig. 5. The rules from Fig. 4a are used to transform this specification to the specification shown in Fig. 7b. The universal constraint in Fig. 6A is transformed to the universal constraint in Fig. 6B. This constraint ensures that if* Work_1 *is dependent on* Work_2 *and the* Scheduler *runs* Work_1 *on* $\mathsf{Computer}_1$ *then the* Scheduler *should run* Work_2 *on a* $\mathsf{Computer}_2$ *which is reachable from* $\mathsf{Computer}_1$.

4 Related Work

In [BGL09] a formal definition of patterns is given as a set of graphs and graph morphisms, defining a fixed part and some variable regions. Variable regions are used to specify multiple occurrences of sub-patterns. Moreover, triple graphs are used for coordinating models with the pattern. The approach also uses synchronisation graphs to relate structural models and behavioral models. In the DPF-based approach patterns are defined by metamodels. Moreover, a variable region may be represented in the metamodel with multiplicity $[0 \ldots *]$. Coordination of patterns with models are done by conformance and match of patterns.

Pattern-based model-to-model transformation is an algebraic, bidirectional and relational approach to model transformation [LG08] based on Triple Graph Grammar (TGG) [Sch94]. This approach is based on triple patterns which express allowed and forbidden relations between two models, where the models are triple graphs. Triple patterns can be seen as graph constraints for triple graphs, which specify both negative and positive constraints. Pattern-based specifications are compiled to operational TGG rules, which perform forward and backward model transformations by graph rewriting. Matches of these patterns are formalised as triple graph morphisms. Furthermore, in [Löw10], graph rewriting is generalised by using span-categories. In the DPF-based approach, universal constraints are used to express requirements which are expressed by triple patterns in pattern-based model-to-model transformations. These constraints are also used to ensure that pattern enforcement is performed in a way that source constraints are transformed adequately to the target models.

In [Grø10] a concrete syntax for definition of input and output patterns of model transformation rules is employed. That is, instead of using the abstract syntax of the modelling languages involved in the model transformation, as done usually, one can define transformation rules employing the syntax used for definition of the models themselves. In addition, this approach offers a collection operator which is used for matching and transformation of collections of similar subgraphs. This operator is used for the definition of patterns with variable regions. The DPF-based approach employs also a concrete syntax for definition of patterns. Moreover, since patterns are defined by metamodels, variable regions are represented by metamodel elements with variable multiplicity.

5 Conclusion and Future Work

In this paper patterns are described by metamodels which are represented as diagrammatic specifications. Pattern enforcement is performed by executing model transformations that transform anti-patterns to models following the desired pattern. Constraints are used to express requirements that should be fulfilled by models. We require that pattern enforcement adequately transforms these constraints.

Since patterns are described by metamodels, the relation between anti-pattern and required pattern may be seen as metamodel evolution. In this context, pattern enforcement may be seen as model migration, i.e. transformation of models conforming to the anti-pattern metamodel to models conforming to the required pattern's metamodel. An interesting line of research in this direction is to find the conditions under which it is possible to automatically generate model migration rules. Some preliminary results about model migration is done in [MRR$^+$10].

An other interesting aspect with patterns is the relation between patterns. Patterns are diagrammatic specifications and it is natural to relate them to each other by specifications morphisms. A further study of different patterns and their relations should be done to obtain a taxonomy of patterns.

References

[BC87] K. Beck, W. Cunningham. Using Pattern Languages for Object-Oriented Programs. Technical report CR-87-43, Tektronix, Inc, September 1987.

[BGL09] P. Bottoni, E. Guerra, J. de Lara. Formal Foundation for Pattern-Based Modelling. In Chechik and Wirsing (eds.), *FASE 2009: 12th International Conference on Fundamental Approaches to Software Engineering*. LNCS 5503, pp. 278–293. Springer, 2009.
doi:/10.1007/978-3-642-00593-0_19

[BW95] M. Barr, C. Wells. *Category Theory for Computing Science (2nd Edition)*. Prentice Hall International Ltd., Hertfordshire, UK, 1995.

[Dis03] Z. Diskin. *Practical foundations of business system specifications*. Chapter Mathematics of UML: Making the Odysseys of UML less dramatic, pp. 145–178. Kluwer Academic Publishers, 2003.

[DW08] Z. Diskin, U. Wolter. A Diagrammatic Logic for Object-Oriented Visual Modeling. In *ACCAT 2007: 2nd Workshop on Applied and Computational Category Theory*. ENTCS 203/6, pp. 19–41. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 2008.
doi:10.1016/j.entcs.2008.10.041

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[Grø10] R. Grønmo. *Using Concrete Syntax in Graph-based Model Transformations*. PhD thesis, Department of Informatics, University of Oslo, Norway, February 2010.

[Küh06] T. Kühne. Matters of (Meta-)Modeling. *Software and System Modeling* 5(4):369–385, 2006.
doi:10.1007/s10270-006-0017-9

[LG08] J. de Lara, E. Guerra. Pattern-Based Model-to-Model Transformation. In *ICGT 2008: 4^{th} International Conference on Graph Transformations*. LNCS 5214, pp. 426–441. Springer, 2008.
doi:10.1007/978-3-540-87405-8_29

[Löw10] M. Löwe. Graph Rewriting in Span-Categories. In Ehrig et al. (eds.), *ICGT 2010: 5^{th} International Conference on Graph Transformations*. LNCS 6372, pp. 218–233. Springer, 2010.
doi:/10.1007/978-3-642-15928-2_15

[Mak97] M. Makkai. Generalized Sketches as a Framework for Completeness Theorems. *Journal of Pure and Applied Algebra* 115:49–79, 179–212, 214–274, 1997.
doi:10.1016/S0022-4049(96)00007-2

[MRR$^+$10] F. Mantz, A. Rossini, A. Rutle, Y. Lamo, U. Wolter. Towards a Formal Approach to Metamodel Evolution. In *NWPT 2010: 22^{nd} Nordic Workshop on Programming Theory*. Pp. 52–54. November 2010.

[RRLW09] A. Rutle, A. Rossini, Y. Lamo, U. Wolter. A Diagrammatic Formalisation of MOF-Based Modelling Languages. In Oriol and Meyer (eds.), *TOOLS 2009: 47^{th} International Conference on Objects, Components, Models and Patterns*. LNBIP 33, pp. 37–56. Springer, 2009.
doi:10.1007/978-3-642-02571-6_4

[RRLW10] A. Rutle, A. Rossini, Y. Lamo, U. Wolter. A Formalisation of Constraint-Aware Model Transformations. In Rosenblum and Taentzer (eds.), *FASE 2010: 13^{th} International Conference on Fundamental Approaches to Software Engineering*. LNCS 6013, pp. 13–28. Springer, 2010.
doi:10.1007/978-3-642-12029-9_2

[Rut10] A. Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, Norway, 2010.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In Mayr et al. (eds.), *WG :$20^t h$ International Workshop on Graph-Theoretic Concepts in Computer Science*. Lecture Notes in Computer Science 903, pp. 151–163. Springer, 1994.
doi:/10.1007/3-540-59071-4_45