Model Patterns
The Quest for the Right Level of Abstraction

Arend Rensink(✉)

University of Twente, Enschede, The Netherlands
arend.rensink@utwente.nl

Abstract. We know by now that evolution in software is inevitable. Given that is so, we should not just *allow* for but *accommodate* for change throughout the software lifecycle. The claim of this paper is that, in order to accommodate for change effectively, we need a modelling discipline with a built-in notion of refinement, so that domain concepts can be defined and understood on their appropriate level of abstraction, and change can be captured on that same level. Refinement serves to connect levels of abstraction within the same model, enabling a simultaneous understanding of that same model on different levels. We propose the term *model pattern* for the central concept in such a modelling discipline.

1 Introduction

Though computer science students start with the idea that when they create a new piece of software, then that software will be completed one day, running happily ever after and never to be looked at again, by the time they graduate we hope to have disabused them of that notion. For all sorts of reasons, software is *always* subject to change; the more successful and widely used, the more urgent the need to maintain it, adapt it, extend it, port it. Rather than always promising ourselves that "we will get it right next time" or thinking "we could do so much better if only we could start from scratch", we should accept and embrace that change is an intrinsic facet of software, and accommodate for it from the start.

This paper takes the position that accommodating for change involves finding the right conceptual level at which the change can be understood. This is, of course, the same level as that at which the architecture of the software can itself be understood. It is typically more abstract than the level of the executable code; instead, in this paper we look at (static, structural) models as the right medium to understand the software as well as the changes therein.

The Poverty of Metamodels. We argue that the current modelling formalisms — primarily UML and ECORE — lack machinery to capture the concepts that really make up the software architecture, and we present an idea for improving upon this. The building blocks of the established modelling formalisms are essentially *classifiers* and *associations.* Though we agree that in the end, a model must be refined to that level in order to bridge the gap to executable code, in our

B. Steffen (Ed.): Transactions on FoMaC I, LNCS 9960, pp. 47–70, 2016.
DOI: 10.1007/978-3-319-46508-1_4

opinion there is a need to first express specific, complex structures from the problem domain directly into a model, before completely descending to the level of classifiers and associations. This is where *model patterns* come in.

Model patterns, as defined in the paper, come in several flavours: formal and informal, abstract and concrete. Being formal means to have a mathematically well-defined specification of the intended structure, in addition to an optional implementation in terms of existing metamodel technology; an informal pattern only defines an implementation. Abstract patterns (which are always formal) *only* embody the mathematially defined structure; concrete ones (which can be formal or informal) define an implementation. A concrete pattern implements an abstract one if its conforming (partial) models provide a one-to-one representation of the abstractly defined structure; if the concrete pattern is itself formal, this can be proved once and for all and relied upon when actually applying the pattern.

Model patterns allow one to take a declarative point of view while designing a domain-specific metamodel: rather than immediately choosing a concrete representation, one can first pencil in the abstract pattern and then choose a concrete pattern to implement it. The level on which the abstract patterns are chosen but not yet concretised is then the "right level of abstraction" of the subtitle of this paper, and change can often be understood as replacing one chosen concrete pattern for another that implements the same abstract pattern.

It should be noted that, in the debate of ECORE vs. UML, our models are inspired by the simpler setting of ECORE. In particular, the notion of an association in UML is quite a bit more complex than the corresponding notion in ECORE. The concepts that UML offers natively can, in fact, be captured by model patterns.

Meta versus Math. Though the primary purpose of introducing patterns is to help accommodating change by finding the right level of abstraction, there is a secondary purpose as well: to bridge the gap between "math" and "meta". By the former we refer to the kind of research, sometimes called "formal methods", where new ideas are presented mainly by using mathematical terminology to define structures, transformations and semantics; by the latter we mean the research in model-driven engineering. Though both fields make use of the same terminology (in particular, the word "model" is central in both), we feel that there is a barrier between them through which it is hard to transfer results. In particular, if one wants to turn conceptual results from the "math" sphere into practical implementations using "meta" technology, one of the recurring tasks at hand is to choose appropriate representations, in terms of metamodels, of common mathematical structures such as sequences, functions, powersets and tuples. Model patterns, as presented in this paper, can provide a systematic library of representation choices to support this process.

Because of the position of this paper in the small intersection between math and meta, we have chosen a certain style of presentation which we feel it may be necessary to say something about. This is a conceptual paper, rather than one that presents a concrete implementation. The ideas proposed here can be implemented on top of UML or ECORE by adapting the notions used here to those respective ecosystems. For instance, here we rely directly on first order logic to express constraints over models, whereas in a practical implementation one would probably want to use OCL instead. By staying away from that level of pragmatism, we avoid some issues (involving the semantics of OCL, among other things) that do not have to do with the ideas we propose, and would indeed threaten to hide those ideas.

The target reader of this article is a modelling expert not necessarily familiar with mathematical definitions, but with a precise enough turn of mind to understand and appreciate them. To serve this target group, we have kept the use of mathematical notation (e.g., the feared Greek alphabet) to a minimum (while making sure that there is an unambiguous formalisation backing everything up) and resisted the temptation to keep all names short. Also, we have refrained from formalising all of the concepts we introduce in the paper.

Roadmap. The remainder of the paper is structured as follows: Sect. 2 presents (our take on) the notions of models, constraints and metamodels on which our contribution is based. Section 3 then introduces model patterns, gives a small catalogue of simple patterns, and shows how they can be instantiated and applied in a metamodel. Section 4 discusses metamodel refinement, pattern discovery, and metamodel evolution using patterns. Finally, Sect. 5 evaluates the contributions of this paper, discusses future directions, and revise some related work.

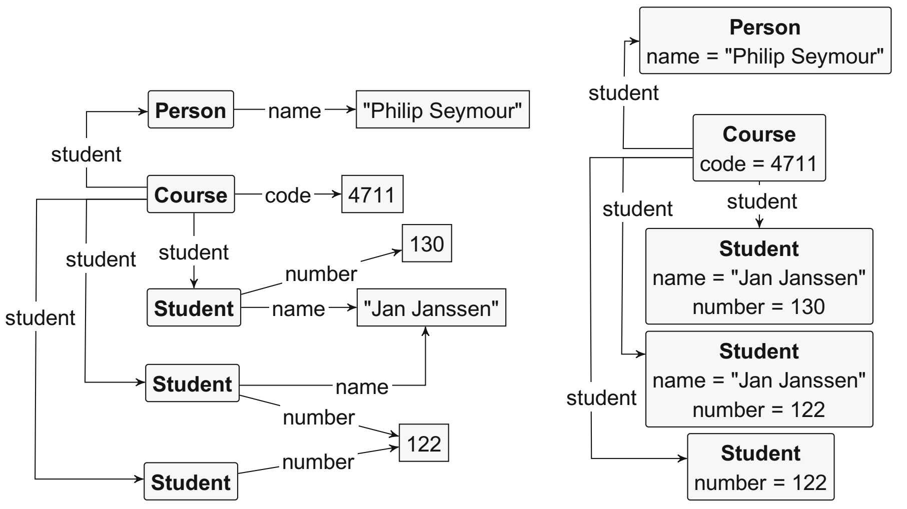
2 Definitions

Throughout this paper, we assume the existence of a well-defined set of identifiers ID. For the sake of simplicity, we do not impose structure on the identifiers; in a more realistic setting, one can think of name spaces to hierarchically group identifiers. As we will see, identifiers are used to stand for a great number of things.

2.1 Models

The reason why metamodels exist at all is that they give rise to a structured set of models (which *conform* to them, in the terminology proposed in [3]), and so characterise the domain of discourse. In mathematical terms, the relation between a metamodel and its conforming models is exactly the relation between a language and its sentences.

Given that, in our view, the set of conforming models of a metamodel is its supremely important aspect, we concentrate first on properly defining the concept of a model. We take a very liberal view of models: they are essentially nothing but graphs, i.e., (labelled) nodes connected by (labelled) edges. As nodes we allow data values as well as more complex, user-defined entities; this means that attributes (edges to data values) are treated in much the same fashion as associations (edges to other user-defined nodes).



(a) Data values displayed as nodes (b) Data values displayed as fields

Fig. 1. Example model, in two distinct versions of concrete syntax

Definition 1 (model). *A model consists of*

– Node*: A finite set of nodes. We will use n (with sub- and superscripts) as a meta-variable to stand for elements of* Node*.*
– type*: A labelling function from* Node *to* ID*, associating a node type* type(n) *with every node n.*
– Edge*: A set of triples $\langle n_1, lab, n_2\rangle$ consisting of source node n_1, edge label lab (an identifier from* ID*) and target node n_2.*

An example model is displayed in Fig. 1. Node-inscribed labels are node types.

Note that we put no restriction whatsoever on the nature of the elements of Node. As stated above and seen in the example, in particular data values may serve as model nodes; the type-function then yields the corresponding data type. In Fig. 1a we have left out the type inscription of nodes corresponding to data values and instead depicted the values themselves as node labels; however, in Definition 1 such nodes do not have any special status. In any case, one very important aspect of a node is its *identity*, as distinct from its *content.*

The identity of a node is that which distinguishes it from other nodes. For data value nodes, the identity *is* the value (there are no two distinct strings `"Jan Janssen"`, for instance) whereas other nodes may have an external identity assigned at the time of their creation, whose precise value or representation is irrelevant except insofar it keeps nodes apart (there may very well be two *persons* called “Jan Janssen”: that value is not their identity). For instance, at run-time the memory address of an object plays the role of node identity, yet the precise value of that address is irrelevant and may very well change as a result of compaction.

The content of a node consists of the information we can access and use. For data value nodes, this coincides with their identity, but for other nodes the content consists of the set of nodes that it has a relation to, in the form of outgoing edges. For instance, in the model of Fig. 1, the content of the **Course**-node consists of its code and the three associated students; the content of the **Person**-nodes consists of their respective names.

It is precisely the fact that, for data nodes, the identity coincides with the content which allows us to use the traditional alternative concrete syntax in Fig. 1b in which edges to data values are inscribed in the nodes.

It should be noted that, though liberal, models as defined in Definition 1 do not offer a lot of structure. Edges are just labelled pairs of nodes, which can easily be understood as records (by combining all outgoing edges of a node) and offer a straightforward encoding of pointers (essentially, edges are nothing but), but there is no notion of collections, lists, maps or other structures that are commonplace in programming languages. Instead, as we will see in the sequel, such structures are typically *encoded.*

2.2 Constraints

In any given concrete application, the models that make sense are typically subject to a lot of constraints imposed by the domain. Such constraints can be formulated in logic. In this paper we choose to use a variant very close to predicate logic, with notations adapted somewhat to make them more reader-friendly, for those not versed in logic. However, we would like to stress that the choice of logic or notation is coincidental to the main idea proposed in this paper; if preferred, one could substitute OCL without any conceptual changes.

Our constraints are first of all built upon expressions, which stand for sets of nodes. Expressions take one of the following forms:

- T, for any identifier T used as node type: The set of all nodes whose type is T or a subtype of T (the notion of subtype will be explained later).
- x, for any node variable x: The singleton set consisting only of the node x.
- $\mathsf{E}.lab$, where E is a sub-expression and lab an identifier used as edge label: The set of all nodes reachable through a lab-labelled edge from a node in E.

Given such expressions, our constraint logic offers the following predicates and combinators:

- E_1 subsetof E_2: The nodes in the set E_1 are all also in the set E_2.
- isempty(E): The set E contains no elements.
- forall x in E : $\mathsf{C}(x)$: All elements in the set E satisfy the constraint C (in which the variable x refers to the E-element in question).
- exists x in E : $\mathsf{C}(x)$: There is at least one element in the set E that satisfies the constraint C (in which the variable x refers to the E-element in question).
- unique x in E : $\mathsf{C}(x)$: There is precisely one element in the set E that satisfies the constraint C (in which the variable x refers to the E-element in question).

Furthermore, constraints can be combined using the usual logical connectives or, and, implies, not and iff (for “if and only if”).

For instance, in the setting of Fig. 1 one may formulate the following constraints:

1. forall x in **Course** : x.student subsetof **Student**, expressing that every target of a student-edge from a **Course**-node is a **Student**-node.
2. forall x in **Student** : not isempty(x.name), expressing that every **Student**-node has a name-attribute.
3. forall x, y in **Student** : x.number $=$ y.number implies $x = y$, expressing that no two distinct **Student**s may have the same set of numbers. (In fact, we also expect that every student has exactly one number; however, that is not expressed by this constraint.)
4. forall x in **Student** : exists y in **Course** : x subsetof y.student, expressing that every **Student** is a student of at least one **Course**.

The essential point about a constraint is that it divides the universe of models into those that *satisfy* it and those that do not. There is a straightforward formal definition of satisfaction, but here we will assume that the concepts we have defined are familiar or straightforward enough so that we can skip that definition.

For instance, the model of Fig. 1 does not satisfy the first three example constraints above (there is a student-edge from a **Course** to a **Person**; there is a nameless **Student**; and there are two **Student**s with number 122) but it does satisfy the last.

In (meta)modelling, certain families of constraints are very common; so much so that (in graphically depicted metamodels) many of them have their own shorthand notation, as we will see. Some well-known examples of such families of constraints are:

$$\begin{aligned}
\mathsf{OutMult}^1[lab] &\equiv \mathsf{forall}\ x : \mathsf{unique}\ y : y\ \mathsf{subsetof}\ x.lab \\
\mathsf{InMult}^{0..1}[lab] &\equiv \mathsf{forall}\ y : (\mathsf{not\ exists}\ x : y\ \mathsf{subsetof}\ x.lab) \\
&\qquad\qquad\quad \mathsf{or}\ (\mathsf{unique}\ x : y\ \mathsf{subsetof}\ x.lab) \\
\mathsf{Opposite}[lab_1, lab_2] &\equiv \mathsf{forall}\ x, y : (y\ \mathsf{subsetof}\ x.lab_1\ \mathsf{iff}\ x\ \mathsf{subsetof}\ y.lab_2) \\
\mathsf{Singleton}[T] &\equiv \mathsf{unique}\ x : x\ \mathsf{subsetof}\ T \\
\mathsf{Key}[lab_1, lab_2, \ldots] &\equiv \forall x, y : (x.lab_1 = y.lab_2\ \mathsf{and}\ x.lab_2 = y.lab_2\ \mathsf{and}\ \ldots) \\
&\qquad\quad \mathsf{implies}\ x = y
\end{aligned}$$

$\mathsf{OutMult}^1[lab]$ expresses that lab-labelled edges have an outgoing multiplicity of 1. Likewise, $\mathsf{InMult}^{0..1}[lab]$ restricts the incoming multiplicity of lab-labelled edges to either 0 or 1. This list is not complete: there are, several more very common multiplicity constraints, and all of them can be applied to incoming as well as outgoing edges. For instance, Constraint 2 above corresponds to $\mathsf{OutMult}^{1..*}[\mathsf{name}]$ and Constraint 4 to $\mathsf{InMult}^{1..*}[\mathsf{student}]$.

$\mathsf{Opposite}[lab_1, lab_2]$ states that, whenever there is an edge (n_1, lab_1, n_2) in a graph, there must be an edge (n_2, lab_2, n_1) in the opposite direction, and vice versa. $\mathsf{Singleton}[T]$ states that there is precisely one T-labelled node in the graph.

Finally, $\mathsf{Key}[lab_1, lab_2, \ldots]$ (the notation is meant to suggest that the Key-predicate can be used with an arbitrary positive number of parameters) states that the combined targets of the outgoing lab_i-edges ($i = 1, 2, \ldots$) together determine the identity of a node. This relates back to the earlier discussion about identity versus content: a Key-predicate can be used to specify that the identity of a node is entirely determined by a specific part of its content.

2.3 Metamodels

We insist on making a sharp distinction between the definition of metamodels and their (graphical) representation. An important place where this distinction shows up is that our metamodels include a set of constraints, of the form discussed above, some of which have a native graphical syntax whereas others do not; nevertheless, formally we treat all of them in the same way. When depicting metamodels graphically, though, we will make use of the well-known graphical conventions.

For the purpose of the following definition, we let Data stand for the set of primitive data types. In the context of this paper, we fix Data to consist only of **Boolean**, **Integer** and **String**. As a reminder, we add the stereotype «datatype» whenever we depict these types graphically.

Definition 2 (metamodel). *A metamodel consists of the following components:*

- Type*: A finite set of user-defined node types, which is a subset of* ID *disjoint from* Data*. We will use T as a meta-variable to stand for elements of* Type *or* Data*.*
- sub*: a subtype relation over types, consisting of pairs of elements of* Type *that impose an irreflexive partial order over* Type*. The latter means that subtyping is transitive (if T_1* sub *T_2 and T_2* sub *T_3 then T_1* sub *T_3) and irreflexive (there is no type T such that T* sub *T).*
- assoc *: A function that assigns to every type T a partial map assoc_T from association names (which are* IDs*) to target types (elements of* Type *or* Data*). If T_1* sub *T_2, then $\mathsf{assoc}_{T_1}(lab) = \mathsf{assoc}_{T_2}(lab)$ for all lab in the domain of assoc_{T_2}.*
- Constr*: a set of constraints, in which all occurring node type identifiers are elements of* Type *or* Data*.*

Note that the definition above speaks of *subtypes*. This is subtly different from the usual notion of *inheritance* (or *extension*) in that the latter stands for *direct* subtyping. We will sometimes use $\mathsf{sub}^=$ to denote the reflexive closure of sub.

There is a limitation in Definition 2 in that subtypes may not redefine associations with a label already occurring in any of their supertypes: instead, they inherit all associations from their supertypes. For practical purposes this does not actually impose a restriction, since one may always prepend association names with their source types, and so disambiguate them.

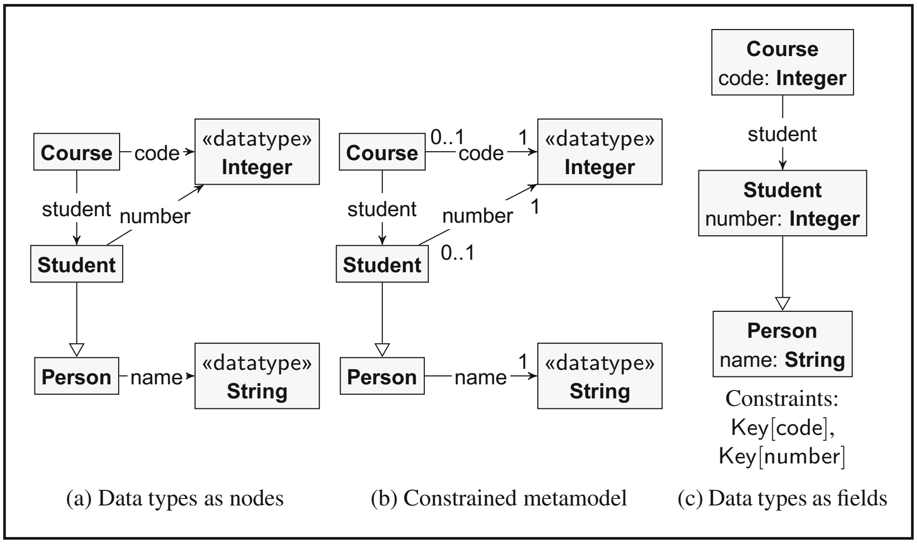


Fig. 2. Example metamodels, in two distinct versions of concrete syntax

An example metamodel, without constraints, is given in Fig. 2a. We have adopted the well-known graphical convention of using open triangular arrows to represent extension, and labelled open-arrowed edges to represent associations. The set of types is given by **Course**, **Student** and **Person**, with subtype relation **Student** sub **Person**. The associations are given by:

$$\begin{aligned}\mathsf{assoc}_{\,\mathbf{Course}} &= (\mathsf{code} \mapsto \mathbf{Integer}, \mathsf{student} \mapsto \mathbf{Student})\\ \mathsf{assoc}_{\,\mathbf{Person}} &= (\mathsf{name} \mapsto \mathbf{String})\\ \mathsf{assoc}_{\,\mathbf{Student}} &= (\mathsf{name} \mapsto \mathbf{String}, \mathsf{number} \mapsto \mathbf{Integer})\end{aligned}$$

A reasonable (though incomplete) set of constraints is given by 2–4 in Sect. 2.2 above. We will see below that Constraint 1 is actually superfluous, since it is already enforced by the association definition of $\mathsf{assoc}_{\,\mathbf{Course}}$.

We started this section by stating that the *raison d'etre* of metamodels is that they define a set of conforming models as their extension. We are now in a position to formally define that notion of conformance.

Definition 3 (conformance). *A model is said to* conform *to a metamodel if all of the following hold:*

1. $\mathsf{type}(n)$ *is an element of* Type *or* Data *for all nodes* n *(meaning that nodes are labelled with metamodel types or data types)*
2. $\mathsf{assoc}_{\,\mathsf{type}(n_1)}(lab)\ \mathsf{sub}^{=}\ \mathsf{type}(n_2)$ *for all edges* (n_1, lab, n_2) *(meaning that model edges must always correspond to an association defined in the metamodel, and the target node type must be a subtype of the type declared in the metamodel).*

3. *The model satisfies all constraints in* Constr*.*

For instance, the model of Fig. 1 does *not* conform to the metamodel of Fig. 2 augmented with the Constraints 2–4 of Sect. 2.2, for the following reasons:

– The condition in Definition 3.2 does not hold: there is a student-edge from a **Course**-node to a **Person**-node which really should go to a **Student**-node.
– The condition in Definition 3.3 does not hold: Constraints 2 and 3 are not satisfied (as remarked before).

A variation of this metamodel, with the same node and edge types but a more complete set of constraints, is given in Fig. 2b. Figure 2c also depicts this adapted metamodel, using a more conventional graphical syntax for data type-valued associations, which here are shown as fields inscribed in their source type nodes. This notation carries some implicit constraints: for every field $lab : T$ appearing in such a metamodel, $\mathsf{OutMult}^1[lab]$ is implied. On the the hand, since these associations are no longer depicted as edges, we cannot directly represent the multiplicity constraints $\mathsf{InMult}^{0..1}[\mathsf{number}]$ and $\mathsf{InMult}^{0..1}[\mathsf{student}]$ in this notation; they are therefore given explicitly in the form of (equivalent) Key-constraints.[1]

3 Patterns

We now come to the core new concept of this paper, that of a model pattern. A pattern can be instantiated (invoked) any number of times in a metamodel. We propose to develop a library of pattern types with known refinement relations between them, so that metamodel evolution can be captured in terms of the replacement of one pattern type by another that is known to refine it, or at least (if pure refinement is not possible) to stand in a well-understood relation to it.

Definition 4 (pattern). *A pattern consists of*

– *A pattern signature, which is a combination of a pattern name (an element of* ID*) and a non-empty sequence* sort *of unique formal parameter names (elements of* ID*), the elements of which stand for types.*
– *An optional specification, being a mathematical description of the structure represented by the pattern, in terms of elements of the types in* sort*. This specification may make use of all the commonly understood machinery of mathematics, including functions, relations, powerset constructions, auxiliary types, and logical constraints.*
– *An optional implementation, being a metamodel such that each of the elements of* sort *are members of* Type*, and one of the elements of* sort *is a designated* handle.
– *If both specification and implementation are given, a proof of correctness. This proof should show that (a) all models conforming to the implementation satisfy the specification, and (b) all structures satisfying the specification can be unambiguously represented as models of the implementation.*

[1] In UML, one may denote this constraint by adding a suffix "{id}" behind the field declaration, as in "code: **Integer** {id}".

If a specification is given, we call the pattern *formal*, otherwise it is *informal*; if an implementation is given, we call the pattern *concrete*, otherwise it is *abstract*. Either the specification or the implementation must be given: abstract informal patterns are not allowed.

The *handle* will come into play when we discuss pattern instantiation: see Sect. 3.2 below. When depicting concrete patterns, we distinguish the handle by shading it gray.

3.1 Example Patterns

We will give a small set of sample patterns for frequently occurring situations, to illustrate the concept and give some intuition about where and how it can be used.

Designated Elements. It may happen that we want to globally mark a single element in a model as being special; for instance, the root of a tree. Mathematically, such a designated element is sometimes called a *point*. We can define this in the form of a pattern as follows:

– Pattern signature: $\mathit{Point}\langle \mathbf{T} \rangle$
– Pattern specification: $\mathbf{T}$ (i.e., the set $\mathbf{T}$ itself)

There is a number of ways to implement this: for instance, the designated element can have a boolean attribute set to true (there must then also be a constraint that only one **T**-element may have the value true for that attribute), or the designated element can be pointed to from a singleton type. These two implementations are depicted in Fig. 3.

Subsets. A subset is needed whenever we want to distinguish or select some of the elements of a given existing type. For instance, in our running example, we might want to distinguish the MSc-students from the others. First we give the formal abstract pattern for subsets:

– Pattern signature: $\mathit{Set}\langle \mathbf{T} \rangle$
– Pattern specification: $2^{\mathbf{T}}$ (the powerset of $\mathbf{T}$)

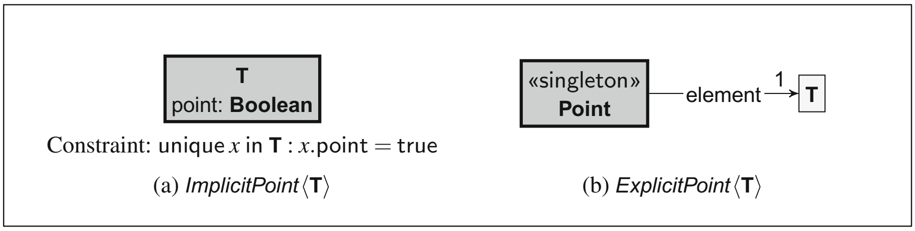


Fig. 3. Concrete patterns for designated elements (abstract pattern *Point*)

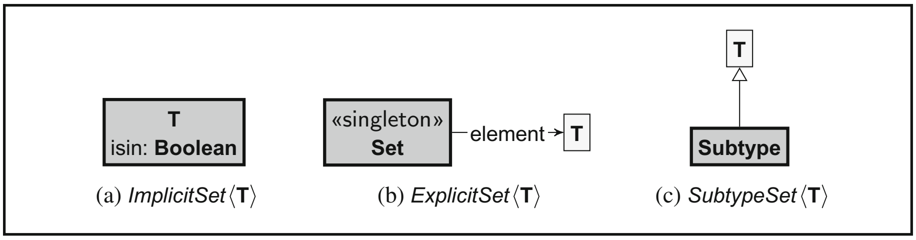


Fig. 4. Concrete patterns for subsets

Sets can be implemented in at least three obvious ways: by adding a boolean attribute to the sort **T**, by creating a singleton class with a set-valued association to **T**, or by introducing a subtype of **T**. These implementations are captured by distinct concrete patterns: *ImplicitSet*, *ExplicitSet* and *SubtypeSet*. Each of these has the same specification as *Set*, but a different implementation, depicted in Fig. 4.

As proofs of correctness for these three implementations, we offer the following.

- For *ImplicitSet*, the set elements are given by All x in **T** : x.isin = true. Clearly, any subset of **T** can be unambiguously represented in this way. (Essentially, *ImplicitSet* is a representation of the so-called characteristic function of a subset.)
- For *ExplicitSet*, the set elements are given by x.element, where x is the unique element of **Set**. The constraint Singleton[**Set**] guarantees that there is indeed such a unique element. Clearly, any subset of **T** can be unambiguously represented in this way.
- For *SubtypeSet*, the set elements are given by **Subtype**; by Definition 3, this is a subset of **T**. Clearly, any subset of **T** can be unambiguously represented in this way.

It should be noted that this pattern models a single, global set, and *not* a set associated with some other object. If, instead, one wants to model a set of **B**-objects associated with every **A**-object (such as, for instance, the set of **Student**s associated with every **Course** in our running example) this requires a set-valued function or relation, rather than a global set; see below.

Relations. Relations are more complicated than sets, in that they involve two types (source and target) rather than just one. From a mathematical standpoint, they are straightforward: the formal abstract pattern is given by

- Pattern signature: *Rel*⟨**T**, **U**⟩ (where **T** is the source type and **U** the target type).
- Pattern specification: $2^{\mathbf{T}\times\mathbf{U}}$ (the powerset of the cartesian product of **T** and **U**)

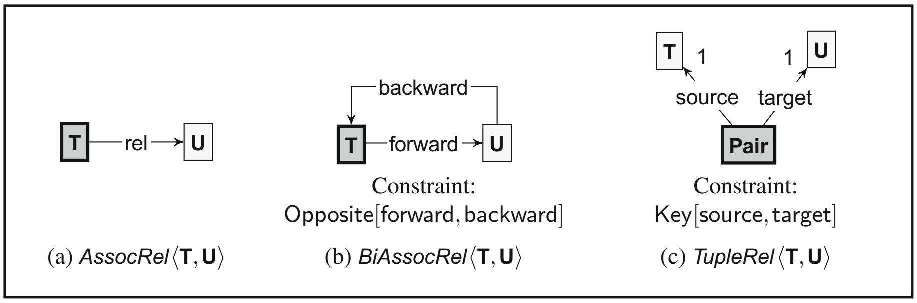

T
rel
U
backward
T
forward
U
Constraint:
Opposite[forward, backward]
T
1
1
U
source
target
Pair
Constraint:
Key[source, target]
(a) *AssocRel*⟨**T**, **U**⟩
(b) *BiAssocRel*⟨**T**, **U**⟩
(c) *TupleRel*⟨**T**, **U**⟩

Fig. 5. Concrete patterns for relations

In this case, there are two bases for concrete implementing patterns: an ordinary association, or a relation class essentially consisting of a set of tuples. Those are shown in Fig. 5, together with a variation upon the first of the two involding an opposite edge, in case one would want to find the set of related **T**-elements from a given **U**-element.

The proof of correctness of *AssocRel* is straightforward, since the set of edge instances of rel precisely encodes a relation.

– For *TupleRel*, the relation is given by the set of $\langle n_1, n_2\rangle$ for which there is an x in **Pair** such that $\langle x, \mathsf{source}, n_1\rangle$ and $\langle x, \mathsf{target}, n_2\rangle$ are edges. Vice versa, given a relation, for every $\langle n_1, n_2\rangle$ contained in it, one can construct a **Pair**-labelled node with outgoing source- and target-edges to n_1 and n_2. The Key-constraint guarantees that this encoding is unambiguous (up to the choice of identity of the **Pair**-nodes, which however is completely determined by the source- and target-edges).

Predicates. Mathematically, tuples are nothing but sequences of values from a fixed number of sets. Thus, we can speak of n-tuples for any natural number n. Special cases are:

– $n = 0$, in which case there is just one value, the empty 0-tuple; this does not correspond to a very useful pattern.
– $n = 1$, in which case the tuples are just single elements of the single set over which the tuples are formed; hence the abstract tuple pattern over a type **T** corresponds to the pattern *Set*⟨**T**⟩ discussed above.
– $n = 2$, in which case the tuples are pairs, i.e., elements of a binary relation between the two sets over which the tuples are formed; hence the abstract tuple pattern over types **T** and **U** corresponds precisely to the pattern *Rel*⟨**T**, **U**⟩ discussed above.

Extending to higher values of n, we can express higher-arity relationships or *predicates* — which is a term borrowed from logic, where a n-ary predicate can be equated with the set of n-tuples that satisfy it. For predicates there is again a range of possible implementations. In any case, we need a special tuple type

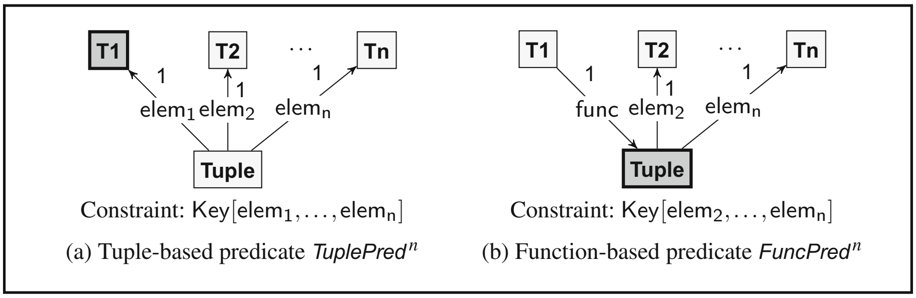

T1
T2
Tn
1
1
1
elem1 elem2 elemn
Tuple
Constraint: Key[elem1,...,elemn]
(a) Tuple-based predicate TuplePredn
T1
T2
Tn
func elem2 elemn
Tuple
Constraint: Key[elem2,...,elemn]
(b) Function-based predicate FuncPredn

Fig. 6. Formal concrete patterns for *Pred*^{n}

with outgoing edges to its constituent sets. One may choose to include any of the opposite edges as well, or turn the predicate into a function from its first element.

– Pattern signature: *Pred*$^n\langle \mathbf{T}_1, \ldots, \mathbf{T}_n\rangle$ (with $n > 2$)
– Pattern specification: $\mathbf{2}^{\mathbf{T}_1\times\mathbf{T}_2\times\cdots\times\mathbf{T}_n}$ (which is equivalent to $\mathbf{T}_1 \rightarrow \mathbf{2}^{\mathbf{T}_2\times\cdots\times\mathbf{T}_n}$)
– Pattern implementations: See Fig. 6

Functions. Functions are essentially a restriction of relations, where every element in of the source type is related to precisely one elements in the target type. Mathematically, this makes them even simpler than relations.

– Pattern signature: *Func*$\langle \mathbf{T}, \mathbf{U}\rangle$
– Pattern specification: $\mathbf{T} \rightarrow \mathbf{U}$ (the function space from $\mathbf{T}$ to $\mathbf{U}$)

The concrete implementing patterns are also closely related to those for relations: either one may use a single-valued association (i.e., with outgoing multiplicity 1), or a map class essentially consisting of a set of tuples, where with respect to *TupleRel* one has to constrain the source arrows to be unique, i.e., to have incoming multiplicity 1.

The correctness proof obligations are a minor variation on those for relations and omitted here.

There are a number of relevant variations on the concept of a function, which we will not treat here in detail but which can be modelled in very similar ways:

– *Injective* functions *InjFunc*, which have the property that for any element of the target type, there is at most one element of the source type that maps to it. This can be captured in the implementation by adding an $\mathsf{InMult}^{0..1}$-constraint to func (Fig. 7a) or target (Fig. 7b).
– *Partial* functions *PartFunc*, which have the property that not every element of the source type has an associated element of the target type. This can be captured in the implementation by relaxing the $\mathsf{OutMult}^{1}$-constraint of func to $\mathsf{OutMult}^{0..1}$ (Fig. 7a) or doing the same for source (Fig. 7b).

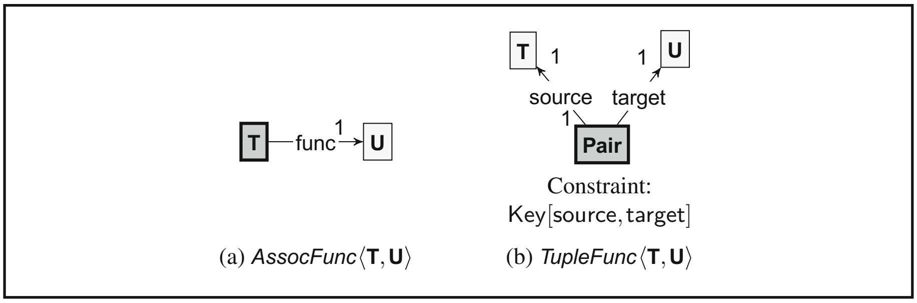


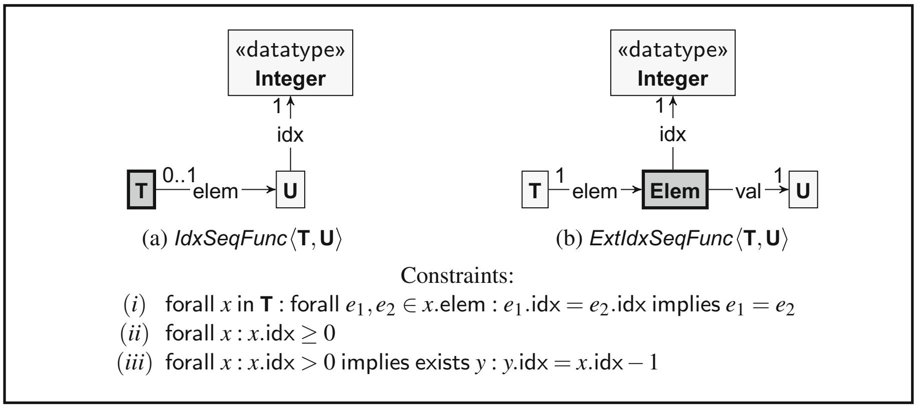
Fig. 7. Concrete patterns for functions

Sequence-Valued Functions. The last category of patterns we discuss here are those that involve (in common metamodel terminology) an *ordered* association from one type to another. Here we pay the price of the structural simplicity (one might say poverty) of our models, which have no in-built notion of ordering.

The formal abstract pattern is straightfoward:

– Pattern signature: *SeqFunc*⟨**T**, **U**⟩
– Pattern specification: $\mathbf{T} \to \mathbf{U}^*$ (the function space from **T** to sequences of **U**)

When choosing an implementation, we are facing well-known issues: (a) Should we use special linking edges or indices to encode the ordering? (b) Should we introduce special nodes to carry the ordering information or integrate it into the existing (target) nodes? (c) Should we mark the start and finish of a list? To demonstrate the range of possibilities, in Figs. 8 and 9 we show four implementing patterns.



Constraints:
(*i*) forall x in **T** : forall $e_1, e_2 \in x$.elem : e_1.idx $= e_2$.idx implies $e_1 = e_2$
(*ii*) forall x : x.idx ≥ 0
(*iii*) forall x : x.idx > 0 implies exists y : y.idx $= x$.idx $- 1$

Fig. 8. Index-based concrete patterns for sequence-valued functions

Figure 8 shows two concrete indexed-based patterns, one in which the ordering structure (i.e., the index) is added to the target type and one in which it is put on a separate intermediate node type especially introduced for this purpose. The first solution is appropriate if no **U**-node can be in more than one list of this kind (as indicated by the incoming multiplicity of the elem-edge).

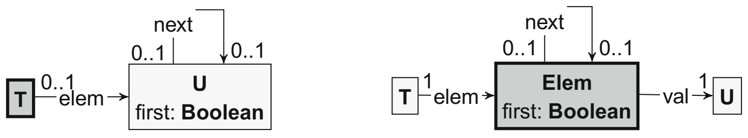
There are a number of non-trivial constraints associated with both of these patterns:

(i) This expresses that the identity of a list node is determined by its containing **T**-node in combination with its index. The effect is that no two nodes in the same list can have the same index.
(ii) This expresses that list indices are non-negative numbers.
(iii) This expresses that list indices form a consecutive sequence. In combination with Constraint (ii), it completely fixes the indices used for a list of n elements to $0, \dots, n-1$. This is necessary to ensure the unambiguous encoding of a sequence in these concrete patterns.

Figure 9 shows two concrete link-based patterns; the difference is again that in one case the next-links are part of the target type **U**, making this solution suitable only if an **U**-node may appear in no more than one list of this kind, whereas the other, heavier-weight solution uses a dedicated intermediate node type. There are again some constraints involved:

(i) Each list should have a unique first element.
(ii) A list element is first if and only if there is no other list element with a next-pointer to it.

Multi-Parameter Functions. We cannot be comprehensive in this paper. A final important pattern, which we just briefly discuss here, is that of a function with two or more parameters — which can equivalently be seen as a function (of the first parameter) yielding a function (of the second parameter). The abstract pattern is:



(a) *LinkSeqFunc*⟨**T**, **U**⟩ (b) *ExtLinkSeqFunc*⟨**T**, **U**⟩

Constraints:

(i) forall x in **T** : unique e in x.elem : e.first = true
(ii) forall x in **T**, e_1 in x.elem :
e_1.first = true iff not exists e_2 in x.elem : e_1 subsetof e_2.next

Fig. 9. Link-based concrete patterns for sequence-valued functions

- Pattern signature: $\mathit{BinFunc}\langle \mathbf{T}, \mathbf{U}, \mathbf{V} \rangle$
- Pattern specification: $(\mathbf{T} \times \mathbf{U}) \to \mathbf{V}$; or equivalently $\mathbf{T} \to (\mathbf{U} \to \mathbf{V})$

The equivalent formulations of the specification already suggest distinct directions of implementation. An intermediate type is almost unavoidable in this case. In fact, pattern *ExtIdxSeqFunc* in Fig. 8b can alternatively be seen as an implementation of $\mathit{BinFunc}\langle \mathbf{T}, \mathbf{Integer}, \mathbf{U} \rangle$. Qualified associations in UML are essentially also a way to implement the multi-parameter function pattern.

3.2 Pattern Instantiation

Patterns may be *instantiated* on top of a given metamodel.

Definition 5 (pattern instance). *Given a metamodel, a* pattern instance *consists of*

- *A pattern instance name (an element of* ID*)*
- *The name of the instantiated pattern together with an argument list, being a sequence of type and pattern instance names from the metamodel, of length equal to the* sort *of the pattern.*
- *If the instantiated pattern is concrete, an optional substitution of the identifiers used in the pattern implementation by identifiers to be used in the pattern instance.*

The substitution specified in the last item is necessary to ensure unambiguous naming: the names in the pattern implementation may overlap with the names already in the model, and so the former must be renamed to be able to merge the pattern implementation with the metamodel.

A *patterned metamodel* is a metamodel with (abstract or concrete) pattern instances. A pattern instance can be visualised in a given metamodel by (essentially) a labelled hyperedge connecting the arguments of the pattern instance, and labelled by the pattern name. Such a hyperedge can alternatively be thought of as a special node. For instance, consider Fig. 10: here the oval nodes are instantiated patterns.

A concrete pattern instance i occurring in a metamodel mm can be *applied* by replacing it by the pattern implementation, in the following way:

1. Create a copy of the implementation metamodel mm_i, replacing the parameter types in the definition by the actual arguments of i and applying the optional substitution of i. If an actual argument of i is a pattern instance (rather than a type), use the handle of that pattern instance.
2. Add the instantiated metamodel to mm. During this step, by the nature of metamodels, different copies of any given type are merged.

For instance, if we thus substitute the concrete patterns in Fig. 10b by their implementations and perform the intended substitution, the result is precisely the metamodel in Fig. 2 that we started with.

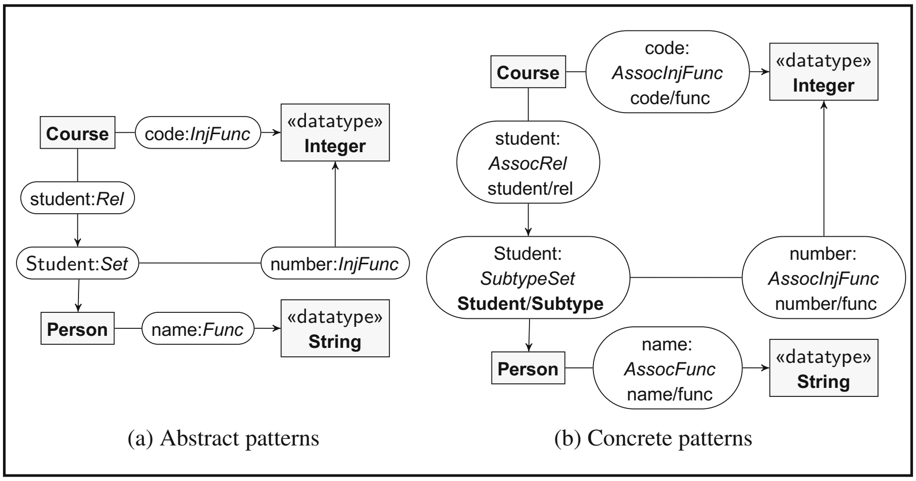


Fig. 10. Patterned metamodel

4 Usage Scenarios

Our claim is that one can benefit from the use of model patterns both when developing new metamodels and when updating existing ones to accommodate changes; and that the latter also applies if the metamodel was not initially pattern-based.

4.1 Refinement

With refinement we mean fleshing out and extending a metamodel, while keeping to the constraints that were imposed before. In other words, when a metamodel mm_1 is refined to a metamodel mm_2, every model of mm_2, when restricted to the node types and edge labels already occurring in mm_1, is a model of mm_1.

Model patterns can help in refinement because choices can be deferred: rather than selecting an concrete implementation from the start, one may first go with

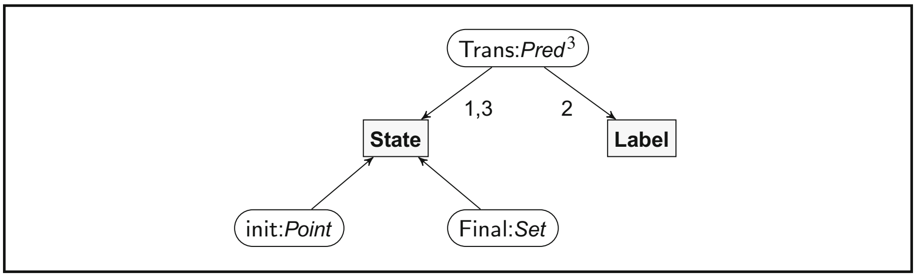
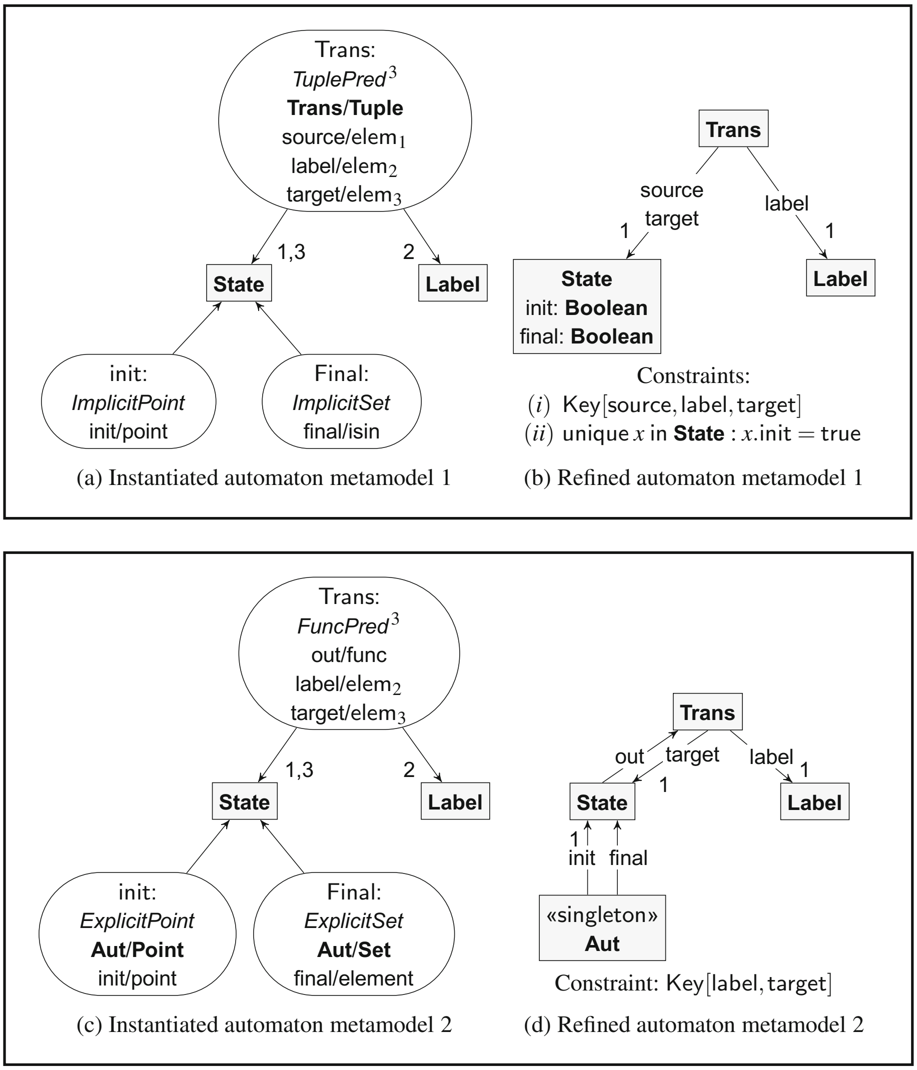


Fig. 11. Abstract patterned metamodel of an automaton

an abstract pattern and later on select an appropriate concrete one. We will illustrate this on an example.

Modelling an Automaton. The example problem is to model the concept of an automaton. Let us assume that we have a mathematical interpretation at hand.


Trans: *TuplePred*^{3} **Trans/Tuple** source/elem$_1$ label/elem$_2$ target/elem$_3$
1,3
2
State
Label
init: *ImplicitPoint* init/point
Final: *ImplicitSet* final/isin
Trans
source
target
label
1
1
State init: **Boolean** final: **Boolean**
Label
Constraints:
(i) Key[source, label, target]
(ii) unique x in **State** : x.init = true
Trans: *FuncPred*^{3} out/func label/elem$_2$ target/elem$_3$
1,3
2
State
Label
init: *ExplicitPoint* **Aut/Point** init/point
Final: *ExplicitSet* **Aut/Set** final/element
Trans
out
target
label
1
1
State
Label
1
init
final
«singleton» **Aut**
Constraint: Key[label, target]

(a) Instantiated automaton metamodel 1

(b) Refined automaton metamodel 1

(c) Instantiated automaton metamodel 2

(d) Refined automaton metamodel 2

Fig. 12. Concrete instantiations of the automaton metamodel

Definition 6 (automaton). *An automaton consists of*

- State*: a set of states;*
- Label*: a set of labels;*
- Trans*: a set of transitions, being triples* $\langle q_0, lab, q_1 \rangle$ *consisting of a source state* q_0 *(from* State*), label lab (from* Label*) and target state* q_1 *(from* State*);*
- init*: an initial state (an element of* State*);*
- Final*: a set of final states (a subset of* State*).*

Using the abstract patterns presented in Sect. 3, we can immediately give a corresponding (patterned) metamodel: see Fig. 11.

Refinement is now a matter of choosing concrete implementations of these patterns. Two examples are shown in Fig. 12.

4.2 Discovery

With "discovery" we mean the process of identifying patterns in an existing, unpatterned metamodel. The process of instantiation discussed in the previous section is a *forward* application of a pattern. However, the instantiation rule may as well be inverted. That is, given an unpatterned metamodel *mm* and a library of concrete patterns, one can *pattern match* (in another use of the word pattern) the implementations of the concrete patterns in *mm*, to find places where the structure of the metamodel suggests a pattern.

For instance, in the example metamodel in Fig. 2b we can recognise instances of *ExplicitSet*, *AssocInjFunc* (twice)[2] and *AssocFunc*. By replacing the corresponding fragments of the metamodel with instances of those concrete patterns, one can (semi-)automatically derive the concrete patterned metamodel of Fig. 10b.

Essentially, therefore, pattern discovery is the process of reconstructing the semantics of a metamodel from its low-level structure. Obviously, this process cannot be fully automatic: any existing metamodel will have many potential instances of patterns. A domain expert will have to be involved to reconstruct the originally intended semantics, by selecting the most appropriate of those potential patterns. Proper tool support is needed to make this a viable, useful step.

We see pattern discovery as a great tool especially to refactor existing, legacy (meta)models.

4.3 Evolution and Migration

The motivation for this paper was *accommodating for change*. In the context of model-driven engineering, change means evolution of metamodels, in most cases accompanied by migration of existing models.

It is here that model patterns yield their greatest benefits. Given a sufficiently large library of (abstract and concrete) patterns, it is possible to *a priori* establish (and prove!) allowed substitutions of one pattern by another. Moreover, the required migration is also implied.

[2] The absract and concrete patterns for injective functions were omitted from Sect. 3.1.

In our modest current setting, such allowed substitutions are already present in the form of alternative implementations of the same abstract pattern. As an example, consider the following scenario pertaining to the metamodel of Fig. 2b:

Evolving the Student Metamodel. Suppose that we realise that it was not a good idea to distinguish the subset of students by turning them into a subtype of **Person**: for now it becomes hard for anyone to change his status from **Person** to **Student** and back. Instead, we want to encode the set of students through a boolean flag in the class **Person**.

In terms of the patterned metamodel in Fig. 10 (which could either be present already if the metamodel was developed through refinement, as advocated in Sect. 4.1, or could alternatively be discovered, as discussed in Sect. 4.2), this change is a matter of choosing another implementation for the Student-pattern. For instance, instead of *SubtypeSet* we could choose *ImplicitSet*, with substitution isStudent/isin. The resulting metamodel is shown in Fig. 13, together with an (automatically) migrated model conforming to the original metamodel to a model conforming to the evolved one.

Note that the handle of the new Student : *ImplicitSet* pattern is the argument type **Person**, meaning that also the number-attribute automatically migrates

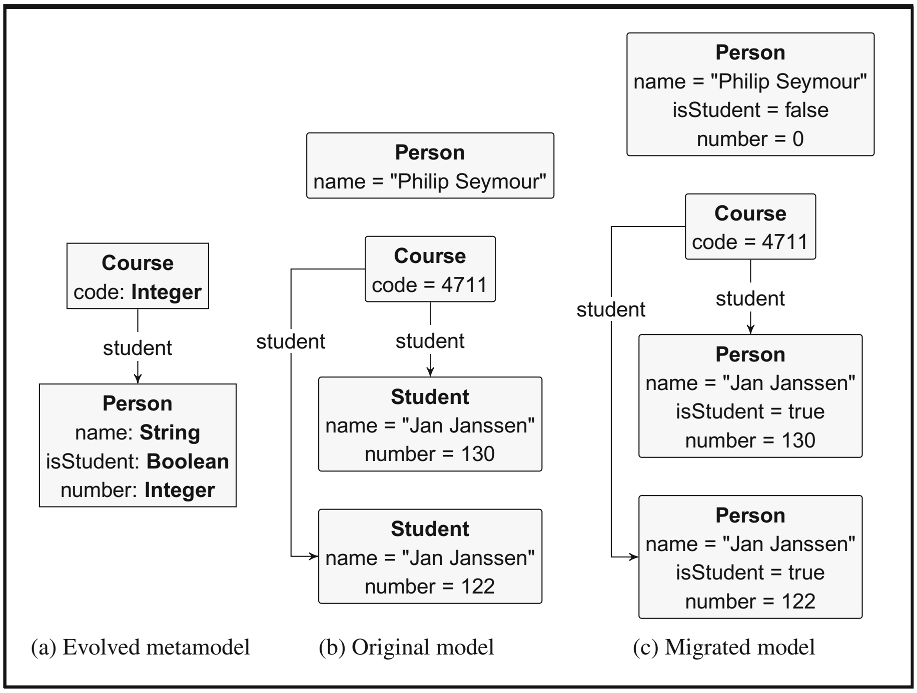


Fig. 13. Metamodel evolution

to **Person**. Thus, in the resulting metamodel, shown in Fig. 13, *all* persons have a number, but it is meaningful only for those for which the isStudent-flag is true.

5 Conclusion

5.1 Evaluation

This paper represents an initial attempt to formalise the notion of patterns. For that reason, rather than trying to be comprehensive and all-encompassing, we have simplified our setting sufficiently to expose the underlying ideas of patterns in a self-contained manner. Those simplifications show up in the stripped-down notions of models, constraints and metamodels, which we have chosen rather than going straight for ECORE or UML.

To summarise, the main benefits expected from patterns are:

- Commonly occurring implementation structures can be catalogued as patterns, while simultaneously capturing their intended semantics.
- Implementation relations can be defined between patterns and (if the patterns are formal) proved formally, enabling correctness by construction.
- Metamodel development, especially when occurring on the basis of existing mathematical definitions, can be eased by deferring the choice of concrete pattern implementations.
- Metamodel evolution can be understood as replacing one concrete pattern instance by another, typically implementing the same abstact pattern. Such replacement steps can be composed, and the corresponding correct model migration is automatically implied.
- Existing metamodels can be subjected to pattern discovery, bringing the benefits of pattern-based evolution also to legacy models.

Using two running examples, we have built a case for the potential of model patterns to eventually reap those benefits, but there is no denying that there is much work to be done before that potential is realised.

Behavioural Models. As remarked in the introduction, in this paper we have restricted ourselves to structural models only. This is, perhaps, misleading as it could strengthen the impression that model-driven engineering is mostly about (data) structures. That is very far from the truth: behavioural models are very widely used in business process modelling [1], and are also reaching the stage where they are as successful in specifying language semantics as structural models are in specifying their syntax: see [13] for an example.

5.2 Future Work

Pattern Library. The example patterns in this paper are actually still quite rudimentary, involving a few types at the most. One of the upcoming challenges will be to create a larger catalogue of more involved, but nevertheless generic patterns.

In Definition 4, we have defined patterns such that their specification is only optional, but in this paper we have actually only discussed patterns that indeed do have a specification; i.e., formal patterns. We believe that informal patterns are nevertheless a useful mechanism to encapsulate commonly occurring metamodel structures, without imposing the requirement to capture that structure in mathematical terms.

Dynamics of Models. We have here entirely ignored the *manipulation* of models — or in other words, the operations typically defined in a metamodel and (the specification of) their intended effect on the models.

Actually, the setup lends itself quite well to an extension with operations. They, too, can be specified formally, on the same level as the current pattern specification, and implemented concretely, for instance through model transformation definitions that work on the pattern implementation. The benefit of *a priori* proving that one pattern implements another can then easily be extended to operations as well.

As a small example, consider adding an element to a set. This is a well-defined operation in the specification of *Set*; in *ImplicitSet* it is a matter of setting the isin-field of the added **T**-element to true; in *ExplicitSet* one should create a new element of **Subset** with an element-edge to the added **T**-element; and in *SubtypeSet* addition is not well possible as this would require changing the runtime type of an object. (This is, in essence, the reason brought forward in Sect. 4.3 against using the *SubtypeSet*-pattern for modelling **Student**s.)

Validation. As the saying goes, the proof of the pudding is in the eating. At the moment of writing, this falls into the category of "future work." In the case of model patterns, validation should consist of

– Developing metamodels from scratch on the basis of patterns (as proposed in Sect. 4.1), or discovering patterns in existing metamodels (as proposed in Sect. 4.2).
– Breaking down existing examples of metamodel evolution in terms of pattern substitutions, as proposed in Sect. 4.3.

5.3 Related Work

This work has its roots in an attempt to formalise the semantics of UML class diagrams, started in [8]. However, we are neither the first nor the foremost to have worked on this: other approaches, with varying degree of comprehensiveness and formality, can be found in [2,4,14,16,17]. Is is, moreover, clear that no such approach can be complete without including first-order logic, mostly (and in the context of model-driven engineering naturally) on the basis of OCL, for which formal foundations have been studied in [5,6,9].

Obviously, a major source of inspiration has been the successful concept of *design pattern* promoted in [7] and a long chain of follow-up work. In contrast to our model patterns, however, design patterns are primarily behavioural (even the category called structural design patterns); together with our focus on model-driven engineering, this gives the current paper quite distinct aims and scope.

When it comes to model *transformation*, the idea of introducing patterns has been studied in [11] (pertaining to patterns in the model transformations themselves) and [10] (lifting this to model transformation *definitions*). Especially the first of these has a connection to our notion of model patterns and their potential use in transformation: if we can a priori prove semantic relationships between patterns, then repeatedly replacing an instance of one pattern by an instance of another is a way to synthesise correct-by-construction model transformations.

[12] introduces the notion of the *intent* of a model transformation, and derives properties as well as transformation themselves on this basis. It also proposes to establish a catalogue of such intents. However, there is no formalisation of intents along the lines of our pattern formalisation.

Finally, the idea of introducing patterns into model-driven engineering has recently inspired a workshop [15], which has, however, unfortunately not resulted in a written proceedings.

References

1. Aguilar-Saven, R.S.: Business process modelling: review and framework. Int. J. Prod. Econ. **90**(2), 129–149 (2004)
2. Atkinson, C., Kühne, T.: Model-driven development: a metamodeling foundation. IEEE Softw. **20**(5), 36–41 (2003)
3. Bézivin, J.: On the unification power of models. Softw. Syst. Model. **4**(2), 171–188 (2005)
4. Bézivin, J., Gerbé, O.: Towards a precise definition of the OMG/MDA framework. In: 16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26–29 November 2001, Coronado Island, San Diego, CA, USA, pp. 273–280. IEEE Computer Society (2001)
5. Brucker, A.D., Tuong, F., Wolff, B.: Featherweight OCL: A proposal for a machine-checked formal semantics for OCL 2.5. Archive of Formal Proofs 2014 (2014)
6. Brucker, A.D., Wolff, B.: A Proposal for a formal OCL semantics in Isabelle/HOL. In: Carreño, V.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2002. LNCS, vol. 2410, pp. 99–114. Springer, Heidelberg (2002). doi:10.1007/3-540-45685-6_8
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
8. Kleppe, A.G., Rensink, A.: On a graph-based semantics for UML class and object diagrams. In: Ermel, C., De Lara, J., Heckel, R. (eds.) Graph Transformation and Visual Modelling Techniques. Electronic Communications of the EASST, Budapest, Hungary, vol. 10. EASST (2008)
9. Kyas, M., Fecher, H., de Boer, F.S., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., Kugler, H.: Formalizing UML models and OCL constraints in PVS. Electr. Notes Theoret. Comput. Sci. **115**, 39–47 (2005)
10. Lano, K., Rahimi, S.K.: Model-transformation design patterns. IEEE Trans. Softw. Eng. **40**(12), 1224–1259 (2014)
11. Lano, K., Rahimi, S.K., Poernomo, I., Terrell, J., Zschaler, S.: Correct-by-construction synthesis of model transformations using transformation patterns. Softw. Syst. Model. **13**(2), 873–907 (2014)

12. Lúcio, L., Amrani, M., Dingel, J., Lambers, L., Salay, R., Selim, G.M.K., Syriani, E., Wimmer, M.: Model transformation intents and their properties. Softw. Syst. Model. **15**(3), 647–684 (2016)
13. Naujokat, S., Lybecait, M., Kopetzki, D., Steffen, B.: CINCO: a simplicity-driven approach to full generation of domain-specific graphical modeling tools (2016, to appear)
14. Seidewitz, E.: What models mean. IEEE Softw. **20**(5), 26–32 (2003)
15. Syriani, E., Paige, R., Zschaler, S., Ergin, H.: First International Workshop on Patterns in Model Engineering (2015). http://www-ens.iro.umontreal.ca/~syriani/pame2015
16. Varró, D., Pataricza, A.: Metamodeling mathematics: a precise and visual framework for describing semantics domains of UML models. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 18–33. Springer, Heidelberg (2002). doi:10.1007/3-540-45800-X_3
17. Varró, D., Pataricza, A.: VPM: a visual, precise and multilevel metamodeling framework for describing mathematical domains and UML (the mathematics of metamodeling is metamodeling mathematics). Softw. Syst. Model. **2**(3), 187–210 (2003)