Software and Systems Modeling manuscript No.
(will be inserted by the editor)

From Types to Type Requirements: Genericity for Model-Driven Engineering

Juan de Lara [⋆], **Esther Guerra**

Universidad Autónoma de Madrid (Spain), e-mail: {Juan.deLara,Esther.Guerra}@uam.es

Received: date / Revised version: date

Abstract Model-Driven Engineering (MDE) is a software engineering paradigm that proposes an active use of models during the development process. This paradigm is inherently type-centric, in the sense that models and their manipulation are defined over the types of specific meta-models. This fact hinders the reuse of existing MDE artefacts with other meta-models in new contexts, even if all these meta-models share common characteristics.

In order to increase the reuse opportunities of MDE artefacts, we propose a paradigm shift from type-centric to requirement-centric specifications by bringing genericity into models, meta-models and model management operations. For this purpose we introduce so called *concepts* gathering structural and behavioural requirements for models and meta-models. In this way, model management operations are defined over concepts, enabling the application of the operations to any meta-model satisfying the requirements imposed by the concept. Model templates rely on concepts to define suitable interfaces, hence enabling the definition of reusable model components. Finally, similar to *mixin layers*, templates can be defined at the meta-model level as well, in order to define languages in a modular way, as well as layers of functionality to be plugged-in into other meta-models.

These ideas have been implemented in MetaDepth, a multi-level meta-modelling tool that integrates action languages from the Epsilon family for model management and code generation.

Key words Model-Driven Engineering – Language Engineering – Meta-Modelling – Genericity – Reutilization

Send offprint requests to:

[⋆] *Present address:* Computer Science Department, Universidad Autónoma de Madrid, 28049 Madrid (Spain)

1 Introduction

Meta-modelling is a core technique in Model-Driven Engineering (MDE), where it is used for language engineering and domain modelling. The main approach to meta-modelling is the OMG's Meta-Object Facility (MOF) [40], which proposes a strict meta-modelling architecture enabling the definition and instantiation of meta-models. MOF has a widespread use, and has been partially implemented in the Eclipse Modeling Framework (EMF) [46]. However, even though meta-modelling is becoming increasingly used at industrial scale, current approaches and tools are scarcely ever concerned with scalability issues like reusability, abstraction, extendibility, modularity and compatibility (i.e. ease of composition) of models, meta-models and model management operators, like transformations or code generators.

Generic programming [24, 25, 48] is a style of programming in which types (typically classes) and functions are written in terms of parametric types that can be instantiated for specific types provided as parameters. This approach promotes the abstraction of algorithms and types by lifting their details from concrete examples to their most abstract form [48]. The advantage is that such a *generic* algorithm can be reused with any type that fulfils the algorithm's requirements. Hence, generic programming shifts the emphasis from type-centric to requirement-centric programming [37], enhancing generality and reusability.

In this paper we propose such a paradigm shift from types to type requirements for MDE as well, by bringing into MDE some of the successful, proven principles of generic programming. The goal is to solve some of the weaknesses of current approaches to meta-modelling, transformation and behaviour specification concerning reusability, modularity, genericity and extendibility. For example, current approaches to behaviour specification tend to define behaviour using the types of one particular meta-model. However, as generic programming often does, one should be able to define generic behaviours applicable to several meta-models sharing some characteristics and without resorting to intrusive mechanisms. In this respect, we show that the use of *generic concepts* specifying requirements from parametric types permits defining behaviours in an abstract, non-intrusive way, being applicable to families of unrelated meta-models. We consider two kinds of concepts: structural and hybrid. The former express requirements on the structure of models and meta-models, whereas the latter define required meta-model operations for specific meta-classes.

Models also suffer from an early concretization of details which hinders their reusability and compatibility. The use of model *templates* allows delaying some details on the model structure by defining model parameters. In this way, a model template can be instantiated with different parameters, allowing its reusability in different situations, and enhancing its compatibility and modularity. The expected requirements for those parameters are expressed through concepts. Model templates are also a mechanism to implement patterns for domain-specific languages and libraries of reusable model components. Moreover, templates can be defined over concepts using genericity as well, so that they can be applicable to families of languages. Hence, generic model templates are a means to specify composable model patterns and fragments in a language-independent way.

Finally, *mixin layers* [44] allow defining meta-model templates provided with generic functional capabilities to be plugged into different meta-models. We found especially useful the definition of *semantic mixin layers* containing the necessary run-time infrastructure for the definition of the semantics of meta-model families, together with associated model simulators expressed on the generic types of the mixin.

As a proof of concept, we have implemented these ideas in a multi-level meta-modelling framework called MetaDepth [14]. This framework allows building systems with an arbitrary number of meta-levels, using deep characterization through potency [3]. The framework provides a simple textual syntax that we use to illustrate the different elements we introduce in the paper, although please note that we do not make use of the multi-level features of our tool in this paper. Hence, our aim is not to describe an extension of MetaDepth or to stress its multi-level meta-modelling capabilities. We believe that *genericity* has a wide potential in meta-modelling, and hence what we describe here has immediate applicability to other frameworks (multi-level or not) like the MOF.

This paper is an extended version of [15]. Here we include hybrid concepts as a way to provide further flexibility by omitting some structural requirements from concepts (which can be implemented in different ways by different meta-models) and providing appropriate operations for encapsulation instead. We also introduce two relation types between concepts: *realization* and *generalization*. A realization is a static binding from a hybrid concept into a structural one which provides an implementation for the operations in the hybrid concept, thus making easier the reuse of any associated generic behaviour. The generalization relation between concepts is similar to interface inheritance in Java. We have also developed the idea of generic model templates, capturing domain-specific modelling patterns in a meta-model independent way. Finally, we have integrated the Epsilon Generation Language (EGL) [42] in MetaDepth, so that generic code generators can be defined as well, and included a section with further case studies.

The paper is organized as follows. Section 2 reviews generic programming. Section 3 introduces MetaDepth so that its syntax is used in the rest of the paper to illustrate the different elements. This section also introduces the model manipulation and code generation capabilities of MetaDepth by using the Epsilon Object Language (EOL) and EGL. Section 4 presents structural concepts, together with the binding from concepts to specific meta-models. Section 5 shows how to define generic behaviours and generic code generators. Section 6 provides flexibility to concepts by the notion of hybrid concept. This section also introduces static bindings from hybrid concepts to structural ones, as well as concept generalizations. Section 7 presents model templates and Section 8 introduces semantic mixin layers. In Section 9 we provide further examples that illustrate the presented ideas. Section 10 discusses related research and Section 11 concludes.

2 From Generic Programming to Generic Model-Driven Engineering

Genericity [24] is a programming paradigm that firstly appeared in languages like CLU and Ada, and was subsequently adopted by many languages like C++, Haskell, Eiffel or Java. Its goal is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct reuse in software construction. It involves expressing algorithms with minimal assumptions about data abstractions, as well as generalizing concrete algorithms without losing efficiency [24]. Genericity promotes a paradigm shift from types to algorithms' requirements, so that even unrelated types may fulfil those requirements, hence making algorithms more general and reusable. Generic programming has enabled some of the most widely used, reusable and flexible libraries, like the C++ Standard Template Library (STL) [47] or Boost [7].

In its basic form, generic programming involves passing type parameters to functions or data types which are then called *templates*. Template functions and template classes may require the parameter types to fulfil a number of requirements for a correct instantiation and execution of the template. This set of requirements is usually expressed using a *concept* [37]. Examples of requirements are a type which must define a "<" binary relation, or a list of data objects with a first element, an iterator and a test to identify the end.

As an example, Listing 1 shows a C++ template function *sdiff* that returns the difference between two objects of type *T*, in absolute value. The operation is not defined for *integral* types only, but the requirements for the type *T* are expressed by the concept[1] *LessThanCompSubst*. The concept demands the type *T* to define the "<" relation operator and a binary subtraction operation. Other languages such as Java support a simpler notion of concept limited to express the requirements of a single type by demanding it to inherit from a specified class or to implement a set of interfaces.

```
template <typename T> requires LessThanCompSubst<T>
  T sdiff(T x, T y) {
    return y < x ? (x-y) : (y-x);
  }

concept LessThanCompSubst <typename T> {
  bool operator<(T, T);
  T    operator-(T, T);
}
```

Listing 1 A template and a concept example in C++.

Mixins are classes designed to provide functionality to other classes, typically through parameterized inheritance, promoting code reuse and modularity. Mixin layers [44] extend mixins by encapsulating fragments of *multiple classes* to define a layer of functionality, which can be added to other sets of classes. They were proposed as a technique for implementing collaboration-based designs, where objects play different roles in different collaborations. In this context, mixin layers provide the needed functionality for each collaboration, so that the final system is obtained by composing mixin layers.

2.1 Applying genericity in Model-Driven Engineering

In this work, we adapt the previous ideas to MDE in order to promote the modularity, extendibility, abstraction and reusability of models, meta-models, and model management operations.

Fig. 1 shows the different elements we introduce. First, we use *concepts* to gather requirements for (meta-)models, and to be able to define both generic behaviours and generic (meta-)models. In particular, similar to template functions in C++, we can make a model management operation generic by defining the operation over a concept instead of over a particular meta-model. In this way we obtain genericity because the concept can be bound to several meta-models (namely those satisfying the concept requirements) and the operation becomes applicable to all of them. Moreover, similar to template classes in C++, we can build *(meta-)model templates* that include parameter types whose requirements are expressed through a concept. Again, these templates can be instantiated with any (meta-)model to which we can bind the concept.

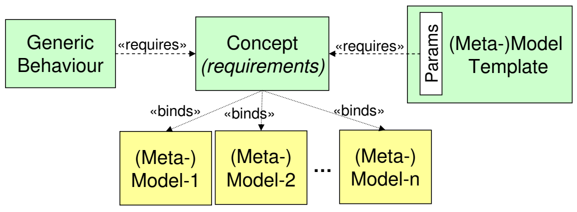

Fig. 1 Fundamental elements for genericity in MDE.

In the rest of the paper we describe the elements in Fig. 1 and some usage patterns. In particular, Section 4 introduces concepts for expressing requirements for meta-models, and the rules for binding a concept to a meta-model. Then, Section 5 explores the use of operation templates to define generic behaviour, like generic simulators and code generators. We discuss bindings from concepts to concepts as well as concept generalizations in Section 6. Next, Section 7 explores the use of concepts to express requirements for models (instead of meta-models) and define model templates. A useful usage scenario for model templates is the definition of generic model fragments, composable via interfaces whose requirements are expressed through concepts. In this way, increasingly complex models can be easily built by instantiating and connecting different model templates. Finally, Section 8 presents meta-model templates. A particular usage of these templates are mixin layers, which are meta-model templates that can be plugged into any meta-model satisfying the requirements expressed by a given concept.

Before delving into details, the next section introduces MetaDepth as we will use its syntax to explain the genericity building blocks.

3 MetaDepth

MetaDepth [14] is a new multi-level meta-modelling framework with support for multiple meta-levels at the same time using *potency* [3]. The potency of an entity is a natural number that indicates the entity's relative meta-level. At each instantiation of the entity in a deeper meta-level, the potency decreases in one unit. When it reaches zero, we obtain an instance that cannot be instantiated further (i.e. without type facet). This approach is very useful to describe what we call *deep languages*, which are languages that involve two or more meta-levels at the user level. An example of a deep language is the combination of UML class and object diagrams, if one thinks of object diagrams as instances of class diagrams [14]. In the present paper we do not make use of the multi-level capabilities of MetaDepth, but stick to a two-level setting where meta-models and their

[1] Concepts were postponed from C++0x, the last revision of C++ [49].

elements have potency one, whereas their instances have potency zero.

MetaDepth uses a textual syntax and is integrated with the EOL and EGL languages of the Epsilon family [22]. EOL [32] extends OCL with imperative constructs to manipulate models, and is used in MetaDepth to express constraints and define behaviours. EGL [42] is a template-based code generator language which can be used in MetaDepth to generate code from models. In this section we give an overview of the textual syntax of MetaDepth and the features used in this paper, see [14] for further details.

As an example, Listing 2 shows the definition of a meta-model for Petri nets using MetaDepth's syntax. The same meta-model is shown in Fig. 2 using a UML representation to ease understanding. Petri nets are a kind of automaton with two types of vertices: *Places* and *Transitions*. Places contain tokens and can be connected with transitions through arcs. In their turn, transitions can also be connected to places through arcs.

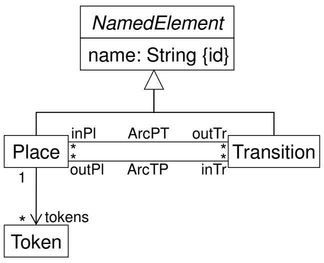

Fig. 2 Meta-model for Petri nets in UML.

```
1  Model PetriNet {
2
3    abstract Node NamedElement {
4      name : String { id };
5    }
6
7    Node Place : NamedElement {
8      outTr : Transition[*] { ordered, unique };
9      inTr  : Transition[*] { ordered, unique };
10     tokens: Token[*]      { unique };
11   }
12
13   Node Transition : NamedElement {
14     inPl : Place[*] { ordered, unique };
15     outPl: Place[*] { ordered, unique };
16   }
17
18   Node Token {}
19
20   Edge ArcPT(Place.outTr,Transition.inPl) {}
21   Edge ArcTP(Transition.outPl,Place.inTr) {}
22
23   minPlaces : $Place.allInstances()->size()>0$
24 }
```

Listing 2 Meta-model for Petri nets in metaDepth.

The listing declares a meta-model named `PetriNet` by using the keyword `Model` (line `1`), which has potency 1 as this is the default potency if none is explicitly given. The meta-model declares an abstract node `NamedElement` owning a field `name` (lines `3-5`). The field's `id` modifier states that no two instances of `NamedElement` can have the same value for the field. Both `Place` and `Transition` inherit from `NamedElement`. The former declares three references (`outTr`, `inTr` and `tokens`) with cardinality `0..*`. References are a kind of field, whose type is a user-defined `Node`. The modifier `ordered` keeps the collection elements in the order of assignment, while `unique` forbids duplicated elements. The opposite ends of `outTr` and `inTr` are declared by the edges `ArcPT` and `ArcTP`. Similar to `Nodes`, `Edges` can also be provided with fields. Thus, in MetaDepth's syntax, `Model` is similar to a meta-model, `Node` to a meta-class, and `Edge` to a meta-association (in fact to an associative class).

MetaDepth supports the definition of constraints and derived attributes in Java and EOL. Constraints can be declared in the context of `Models`, `Nodes` and `Edges`. Line `23` in the listing declares an EOL constraint named `minPlaces`, which demands `PetriNet` models to have at least one `Place`. Please note that, while in MOF-based meta-modelling environments this constraint should be placed in the context of some meta-class (like `Place` itself, or in an additional root class), MetaDepth allows a more natural placement of the constraint in the context of the model itself. Moreover, as MetaDepth allows specifying multiplicities in the definition of `Nodes`, the same effect can be obtained by replacing line `7` by "`Node Place[1..*] : NamedElement {`".

The defined meta-model can be instantiated as Listing 3 shows. This Petri net model represents a system with two processes (producer and consumer) communicating through a buffer of infinite capacity. Fig. 3 shows the system using the usual Petri nets visual notation, with places represented as circles, transitions as black rectangles, and tokens as black dots inside places. The dotted rectangles delimit the different conceptual components of the system. All elements in Listing 3 have potency zero (as they are instances of elements of potency 1) and cannot be further instantiated.

```
PetriNet ProducerConsumer {
  Place WP { name="waitProduce"; }
  Place RP { name="ReadyProduce"; }
  Transition ReadyP { name="readyP"; }
  Transition Produce { name="in"; }
  ArcPT (RP, Produce);
  ...
  Place Buffer { name="Buffer"; }
  ...
  Place C { name="Consume"; }
  Place WC { name="waitConsume"; }
  Transition Consume { name="out"; }
  Transition ReadyC { name="waitC"; }
  ...
}
```

Listing 3 A Petri net with the Producer-Consumer example in MetaDepth's syntax.

Listing 3 makes use of the normal instantiation capabilities found in most meta-modelling frameworks (like EMF [46]). However, one soon notices that the definition of our model could be improved concerning abstraction

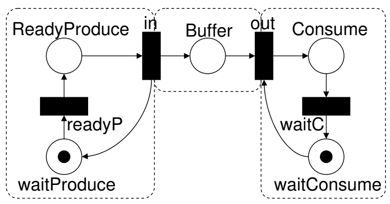

Fig. 3 A Petri net with the Producer-Consumer example, in visual notation.

and modularity. First, the user could have been offered higher-level modelling elements than places and transitions, like *Buffers* and *Processes*. Moreover, inspecting the model, one realizes that the two processes have exactly the same structure (two places connected by transitions). Therefore, it would have been useful to have a meta-modelling facility to define model components – similar to modelling patterns – that the user can instantiate and interconnect through suitable interfaces. Such Petri net component models are enclosed in dashed rectangles in Fig. 3. Section 7 will demonstrate how the use of templates allows performing this *at the model level*, without any need to modify the meta-model.

3.1 Defining in-place transformations

MetaDepth allows defining behaviour for models using either Java or EOL [14]. EOL is however very well suited for this purpose, as it permits defining methods on the meta-classes of the meta-models. Listing 4 shows a simulator written in EOL to execute Petri net models. The entry point for its execution is the operation `main` (line 2), which is annotated with the meta-model to which the operation is applicable (`PetriNet` in our case, so that the operation can be applied to instances of this meta-model). The listing declares several auxiliary operations. Two of them are defined on a global context: `writeState` (line 15) prints the state of the net, and `getEnabled` (line 17) returns a set of enabled transitions. The other two – `enabled` and `fire`, in lines 22 and 26 – are defined on the context of the `Transition` meta-class. While operation `enabled` checks if the transition is fireable (all input places have at least one token), `fire` executes the transition, removing tokens from the pre-places and adding tokens to the post-places. These operations are invoked in the while loop of the `main()` operation (lines 6-12), firing randomly one of the enabled transitions. The loop is restricted to a maximum number of iterations to prevent infinite executions.

```
 1 @metamodel(name=PetriNet,file=PetriNet.mdepth)
 2 operation main() {
 3   var maxStep : Integer := 100;
 4   var numStep : Integer := 0;
 5   var enabled : Set(Transition) := getEnabled();
 6   while (enabled.size()>0 and numStep<maxStep) {
 7     var t := enabled.random();
 8     t.fire();
 9     writeState(numStep);
10     numStep := numStep+1;
11     enabled := getEnabled();
12   }
13 }
14
15 operation writeState(step: Integer) {...}
16
17 operation getEnabled() : Set(Transition) {
18   return Transition.allInstances().select( t |
19          t.enabled()).asSet();
20 }
21
22 operation Transition enabled() : Boolean {
23   return self.inPl->forAll(p|p.tokens.size()>0);
24 }
25
26 operation Transition fire() {
27   for (p in self.outPl) {
28     p.tokens.add(new Token);
29   }
30   for (p in self.inPl) {
31     var t : Token := p.tokens.random();
32     p.tokens.remove(t);
33     delete t;
34   }
35 }
```

Listing 4 A simulator for Petri nets.

This simulator works well for instances of the Petri net meta-model. However, there are many languages whose semantics can be defined in terms of Petri nets, such as workflow languages [9] and UML activity diagrams [39]. Also, domain specific languages like production systems [17] (where parts are consumed and produced by machines), communication systems [8] (where messages are sent and received by nodes), and data-flow languages [16] (where data are consumed and produced by processors) share semantics with Petri nets. Therefore, couldn't we abstract the essential elements of Petri net-like languages and define their behaviour in a generic way? Section 4 will show that *concepts* are a solution to this issue.

3.2 Defining code generators

We have recently integrated EGL into MetaDepth. EGL is a template-based language for producing textual artefacts from models. It combines the model navigation capabilities of EOL with facilities for emitting textual code. The EOL code in the templates is delimited by the markers "[%" and "%]", whereas the text outside these markers is either emitted in the standard output or saved into a file.

As an example, Listing 5 shows an excerpt of an EGL template that generates a PNML [27] text file from a Petri net model. PNML is a standard XML representation for different kinds of Petri nets, used by many tools like CPNTools [13] or PIPE [6]. Line 2 indicates that the template is defined over the Petri Net meta-model, and is to be executed on its instances. Then, lines 4-6 emit the XML header, and lines 7-16 iterate on all instances of *Place* generating the text in lines 8-15 at each iteration.

```
1 [%
2 @metamodel(name=PetriNet,file=PetriNet.mdepth)
3 %]
4 <?xml version="1.0" encoding="iso-8859-1"?>
```

```
<pnml>
<net id="Net-One" type="P/T net">
[% for (h in Place.allInstances()) {%]
 <place id="[%=h.name%]">
  <name>
    <value>[%=h.name%]</value>
  </name>
  <initialMarking>
    <value>[%=h.tokens.size()%]</value>
  </initialMarking>
 </place>
[%}%]
...
```

Listing 5 Code generator for Petri nets (excerpt).

The PNML code generated from a Petri net model can be loaded in CPNTools or PIPE for analysis. For instance, we can calculate the reachability graph [36] of the net, a graph-based representation of all its possible states, which can be used for verification of reachability properties and model-checking.

Again, we would like to use the template in Listing 5 with other languages (apart from Petri nets) whenever their semantics can be mapped to Petri nets, and without requiring the types of a specific meta-model. The next section will show how structural concepts are a means to gather requirements for the executability of a given model management operation over certain meta-models. Hence, they allow the development of generic model operations and code generators which can be used with different meta-models.

4 Structural Concepts

A *concept* in meta-modelling is a pattern specification that expresses requirements for a model (at any meta-level). Concepts serve as a dual typing in the context where they are used (e.g. generic model management operations), providing an extra level of indirection which we use to define behaviour independently of specific meta-models. This is useful for reusability and composition of behaviours, which can be defined in terms of concepts instead of in terms of particular meta-models.

In order to motivate and introduce the use of concepts, we first start discussing an illustrative scenario.

4.1 Motivation

Assume one needs to describe the behaviour of two languages, one in the domain of Production Systems (where parts are produced and consumed by machines) and the other for Communication Networks (where packets are produced and consumed by computers). The most immediate approach is to define one program to simulate the first kind of models, and another one to simulate the second kind of models. This situation is illustrated in Fig. 4. In the figure we assume that behaviours are realized using EOL programs, however our discussion and subsequent proposal based on concepts are applicable to other means of specification of in-place model transformations, like e.g. graph transformation [20].

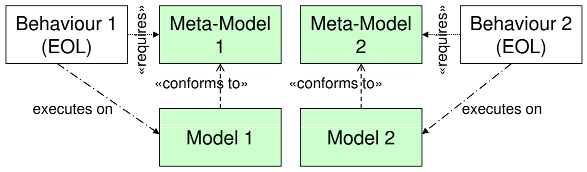

Fig. 4 Direct approach to behaviour specification.

An analysis of the semantics of these two languages reveals similarities between the two programs implementing them. This is due to the fact that both behaviours can be mapped into the standard semantics of Petri nets. Hence, instead of defining such similar behaviours twice, we can transform the models into a common language (Petri nets) and define the behaviour for the common language only once. This situation is depicted in Fig. 5, where `Model 1` is transformed into `Model 1′` and `Model 2` is transformed into `Model 2′`, being both transformed models conformant to the same meta-model for which the behaviour is specified (Petri nets in our example). Unfortunately, this situation is not ideal either, as one has to define specific model-to-model transformations between each language and the common language. Moreover, after modifying the transformed models according to the behaviour, these have to be translated back to their original language.

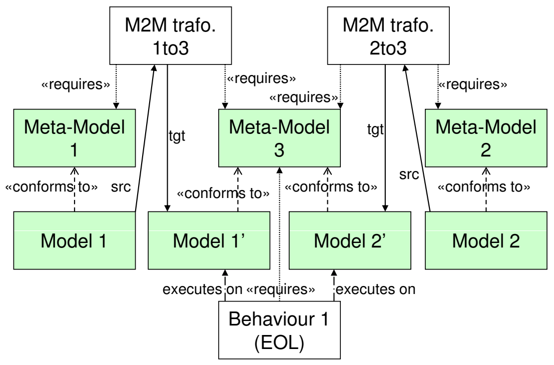

Fig. 5 Transformational approach to behaviour specification.

An improvement that avoids transforming models is to use an extension, nominal subtyping or inheritance mechanism for meta-models [33]. In this case which Fig. 6 illustrates, the meta-models `1` and `2` explicitly extend a third meta-model for which the behaviour is defined. In particular, their classes extend (or subclass) the classes that participate in the defined behaviour for meta-model `3`, so that this behaviour also applies to the classes in `1` and `2`. However, this solution is intrusive as it requires all meta-models for which we want to define behaviour to inherit or extend the same meta-model. Hence, this approach requires the parent meta-model to be defined beforehand, and the adopted solution may become unfeasible if more than one semantics (e.g. timed and untimed) are to be defined for the same language.

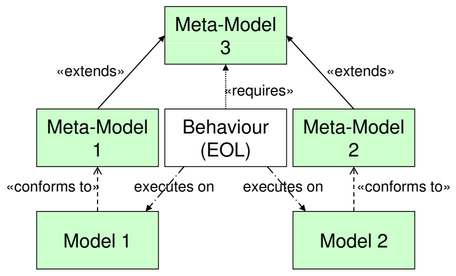

Fig. 6 Inheritance of behaviour by model extension.

In this scenario, *concepts* can simplify the situation as they can express requirements on meta-models or models that some specifications (in this case the behaviour) need. In our example, we can define a concept expressing the requirements that a simulator for Petri net-like languages needs. This simulator abstracts from the specific details of the languages, and uses only the elements defined in the concept, hence being *independent of any meta-model* and therefore non-intrusive. Thus, if our two original languages satisfy the requirements of the concept, then the behaviour can be applied to their instances as shown in Fig. 7. This scheme is the simplest and cleanest of the four, and its benefits increase as we find new meta-models in which the concept is applicable as we can reuse the defined behaviour for them. Moreover, the mechanism is non-intrusive: the meta-models for which we are defining the behaviour are not modified and are oblivious of the concepts. That is, as a difference to Fig. 6, meta-models do not have any dependence on the concepts, and do not require the presence of concepts beforehand. A similar effect could be achieved by using structural subtyping mechanisms [10,45], where the relations between the subtypes (meta-models) and the supertypes (concepts) do not need to be explicitly declared because they are inferred. Section 10 will discuss the similarities and differences between structural subtyping and our proposal based on concepts in more detail.

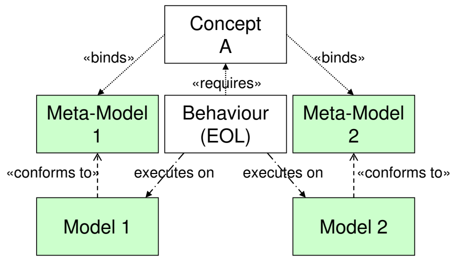

Fig. 7 Behaviour specification based on concepts.

4.2 Defining and binding structural concepts

A structural concept is a specification gathering the structural requirements that need to be found in a model, at a particular meta-level. In this section we will discuss concepts for meta-models, however in Section 7 we will show that concepts can be defined at the model level in the same way as for meta-models.

The simplest way for expressing requirements for a meta-model is in the form of a meta-model as well. Therefore, in our approach, a concept (at the meta-model level) has the form of a meta-model, where the elements in the concept are interpreted as variables, to be bound to elements of specific meta-models. Meta-model concepts may include inheritance relations as well. Moreover, as we will see, concepts can be enriched with further constraints, to be evaluated when the binding is performed.

In our approach, a *concept* has a name and a number of parameters that represent generic types of models, nodes, edges or fields. Concepts can be bound against meta-models by a pattern-matching mechanism. In this way, a concept C defines a language $L(C)$ of all meta-models that satisfy the requirements imposed by the concept C. Thus, $L(C)$ contains a family of meta-models sharing similar characteristics. This situation is depicted in Fig. 8. The set of meta-models belonging to set $L(C)$ can be characterized using a function *bind* from (the set of nodes, edges and fields in) the concept to (the set of elements in) the meta-model. If such a function *bind*: $C \rightarrow M$ exists, then we say that $M \in L(C)$.

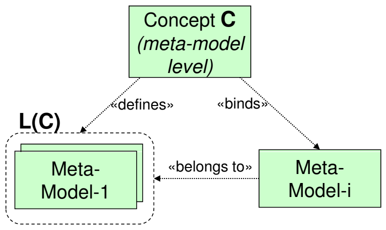

Fig. 8 Meta-model concept, associated language and binding.

The *bind* function maps each model, node, edge or field in the concept to a model, node, edge or field in the meta-model, respectively. There is no need to bind the inheritance relations appearing in the concept though, but the binding must preserve the subtyping relation. Thus, if a node a inherits from a node b in the concept, then the node bound to a must be a direct or indirect child of the node bound to b. Moreover, if two fields (or references) are bound, then so must be their container nodes. Alternatively, the bound meta-model node can be a subnode of the actual field's (or reference's) container node. In any case, the binding must preserve the type of fields and references. As a consequence, since edges are made of two opposite references, if an edge e

in the concept is mapped to an edge *bind*(e) in the meta-model, then the source node of e should be mapped to the source node of *bind*(e), and similarly for the target nodes. The binding must also preserve the cardinality of fields and references. Finally, the binding can be non-injective, therefore two different nodes in the concept can be mapped to a single node in the meta-model, provided that such a node defines all the features defined by the two nodes in the concept.

We use concepts to define generic model management operations using their parameters as generic types, as well as to describe conditions to be fulfilled by template parameters. In contrast to generic programming, where concepts are used to restrict the allowed types to only those defining a certain set of operations, concepts in meta-modelling refer to structural features of meta-models. Thus, concepts can impose a certain structure for nodes, edges and fields, as well as define arbitrary constraints to restrict their applicability.

Fig. 9 shows a concept gathering the structural requirements for meta-models to be simulated with similar semantics to Petri nets, which we call Token-Holder semantics. Listing 6 shows the same concept using MetaDepth's syntax. The concept declares seven parameters, which are treated as variables and start by "&". The body of the concept requires `&M` to be a model with three nodes. Node `&T` plays the role of token. Node `&H` plays the role of a holder of tokens, as it is demanded to define a reference of type `&T`. Node `&P` plays the role of a process or transition, and it must define two references modelling the connection to input and output holders. The body of a concept may include extra conditions expressed in EOL, as well as constant elements as opposed to variables. For example, we could demand node `&H` to have a field called *name* of type `String`.

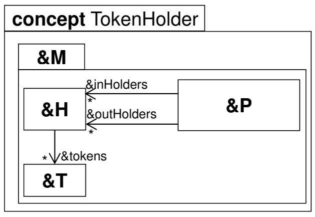

Fig. 9 Structural concept for Token-Holder semantics.

```
concept TokenHolder(&M, &H, &P, &T, &tokens,
                    &inHolders, &outHolders) {
  Model &M {
    Node &H {
      &tokens : &T[*];
    }
    Node &P {
      &inHolders : &H[*];
      &outHolders: &H[*];
    }
    Node &T {}
  }
}
```

Listing 6 Structural concept for Token-Holder semantics in MetaDepth syntax.

We use this concept to characterize the family of meta-models sharing the Token-Holder semantics. For example, the concept can be bound to the `PetriNet` meta-model of Listing 2, where `&H` is bound to `Place`, `&P` to `Transition`, and so on. Listing 8 shows how this binding is specified in MetaDepth through the `bind` command, to which we pass specific meta-model elements.

```
bind TokenHolder(PetriNet,
                 PetriNet::Place,
                 PetriNet::Transition,
                 PetriNet::Token,
                 PetriNet::Place::tokens,
                 PetriNet::Transition::inPl,
                 PetriNet::Transition::outPl)
```

Listing 7 Binding the `TokenHolder` concept to the Petri nets meta-model defined in Listing 2.

The same concept can be bound to other unrelated meta-models as well. As an example, Listing 8 defines a meta-model for Production Systems and its binding over the `TokenHolder` concept. The meta-model declares machines and conveyors, which can be connected to each other. Conveyors hold parts, which are fed into machines. Machines process parts, which are produced into conveyors. In this way, this working scheme is adequate for its simulation using Token-Holder semantics. Hence, we use the `TokenHolder` concept and bind it to the meta-model in lines `22-28`: conveyors act like token holders, machines as processes or transitions, and parts as tokens.

```
 1 Model ProductionSystem {
 2   Node Machine {
 3     ref : String;
 4     type : String;
 5     inConveyors : Conveyor[*];
 6     outConveyors : Conveyor[*];
 7   }
 8   Node Conveyor {
 9     outMachines : Machine[*];
10     inMachines : Machine[*];
11     parts : Part[*];
12   }
13   Node Part {
14     creationTime : int;
15     owner : Conveyor[0..1];
16   }
17   Edge MC(Machine.outMachines,Conveyor.inConveyors);
18   Edge CM(Conveyor.outConveyors,Machine.inMachines);
19   Edge iP(Part.owner,Conveyor.parts);
20 }
21
22 bind TokenHolder(ProductionSystem,
23                  ProductionSystem::Conveyor,
24                  ProductionSystem::Machine,
25                  ProductionSystem::Part,
26                  ProductionSystem::Conveyor::parts,
27                  ProductionSystem::Machine::inConveyors,
28                  ProductionSystem::Machine::outConveyors)
```

Listing 8 Binding the `TokenHolder` concept to the Production System meta-model.

5 Generic Model Management Operations

We can define generic model management operations by using the variable types of a concept, instead of the types of a specific meta-model. In this way, the model management operation is applicable to instances of any meta-model that satisfies the concept's requirements. We next provide two examples, which generalize those in Sections 3.1 and 3.2. The first one is a generic simulator and the second a generic code generator, both defined on the `TokenHolder` concept and hence applicable to instances of any meta-model this concept can be bound to.

5.1 Generic simulators

Listing 9 shows an excerpt of the EOL simulator for the `TokenHolder` concept. The program first states that it needs concept `TokenHolder` (line `1`), therefore it will be executed on instances of meta-models satisfying the concept. Then, the program uses the generic types and features defined by the concept. This program is actually an abstraction of that of Listing 4, because this one does not require concrete meta-model types. The working scheme is the same, but the operations `enabled` and `fire` are added to the node `&P` gets bound to. The simulator can be used to execute any instance of the `ProductionSystem` and `PetriNet` meta-models, hence being more reusable than the one in Listing 4.

```
1  @concept(name=TokenHolder,file=TokenHolder.mdepth)
2  operation main() {
3   var maxStep : Integer := 100;
4   var numStep : Integer := 0;
5   var enabled : Set(&P) := getEnabled();
6   while (enabled.size()>0 and numStep<maxStep) {
7    var t := enabled.random();
8    t.fire();
9    writeState(numStep);
10   numStep := numStep+1;
11   enabled := getEnabled();
12  }
13 }
14
15 operation writeState(step: Integer) {...}
16
17 operation getEnabled() : Set(&P) {
18  return &P.allInstances().select(t |
19         t.enabled()).asSet();
20 }
21
22 operation &P enabled() : Boolean {
23  return self.&inHolders->forAll(p |
24         p.&tokens.size()>0);
25 }
26
27 operation &P fire() {
28  for (p in self.&outHolders) {
29   p.&tokens.add(new &T);
30  }
31  for (p in self.&inHolders) {
32   var t : &T := p.&tokens.random();
33   p.&tokens.remove(t);
34   delete t;
35  }
36 }
```

Listing 9 Generic simulator over the `TokenHolder` concept.

5.2 Generic code generators

In addition to simulators, we can also define generic code generators over the `TokenHolder` concept. Listing 10 shows an EGL template over the concept. Compared to Listing 5 it can be noted that line `9` iterates on all instances of the type `&H` gets bound to. The EGL template can only use the features defined in the concept, and so we cannot use *h.name* as holders are not required to have a *name* field. Instead, we concatenate the name of the type `&H` gets bound to (by means of *h.type()*) with an index, we store this in variable *name* in line `10`, and subsequently we use the variable in lines `12` and `14`.

```
1  [%
2  @concept(name=TokenHolder,file=TokenHolder.mdepth)
3  %]
4  <?xml version="1.0" encoding="iso-8859-1"?>
5  <pnml>
6  <net id="Net-One" type="P/T net">
7  [%
8  var i : Integer := 0;
9  for (h in &H.allInstances()) {
10  var name : String := h.type().name.toString()+i;
11 %]
12  <place id="[%=name%]">
13    <name>
14      <value>[%=name%]</value>
15    </name>
16    <initialMarking>
17      <value>[%=h.&tokens.size()%]</value>
18    </initialMarking>
19  </place>
20 [%
21  i := i+1;
22 }
23 %]
24 ...
```

Listing 10 Generic code generator over the `TokenHolder` concept (excerpt).

Altogether, we can apply the generic code generator to the instances of meta-models like the `ProductionSystem` and `PetriNet`, hence obtaining meta-model independence.

However, the presented binding of concepts to meta-models is somehow limited, as it requires an embedding of the structure of the concept in the concrete meta-model. A concept reflects a design decision, but other possibilities may exist as well. For example, the `TokenHolder` concept explicitly models tokens with a separate node, but some meta-models could have modelled tokens with an integer field in the holder node. Similarly, we have modelled the relation between holders and processes with references, but other more complex relations are possible as well, like using intermediate nodes. Therefore, a more flexible mechanism is desired which allows binding concepts to meta-models with certain structural heterogeneities. The next section proposes so-called hybrid concepts as one solution to this problem.

6 Hybrid Concepts, Static Binding and Concept Generalization

Structural concepts allow for the definition of generic model management operations by using the type variables of the concept. Then, the concept can be bound to several meta-models, and in this way the operation becomes reusable. However, the *binding* requires an embedding of the concept in the meta-model. One way to overcome this problem is through the definition of suitable interfaces that (partially) hide the specific structure of concepts behind appropriate operations. We call such an operation-based requirement specification a *hybrid concept*, as in addition to structural requirements it contains the necessary operations to be defined by certain meta-model elements.

For example, Listing 11 shows a hybrid concept that requires two nodes: `&H` representing the role of holders and `&P` representing the role of processes. However, instead of demanding certain structural relations between holders and processes, or between holders and tokens, their connectivity is modelled by operations. Hence, the concept requires three operations in holders: `tokens()` to query the number of tokens in the holder, and `addToken()` and `delToken()` to increase or decrease the number of tokens. For the process role the concept requires operations `inputHolders()` and `outputHolders()` returning the collections of input and output holders of a given process. These operations are not interpreted as variables, and thus their names are not preceded by &, but operations with same name and signature should be provided when performing the binding.

```
concept ProcessHolderB(&M, &H, &P) {
 Model &M {
  Node &H{
    operation tokens() : Integer;
    operation addToken();
    operation delToken();
  }
  Node &P{
    operation inputHolders() : Set(&H);
    operation outputHolders(): Set(&H);
  }
 }
}
```

Listing 11 Hybrid concept `ProcessHolderB`.

A generic simulator or code generator may then use the operations declared in the hybrid concept. Listing 12 shows the generic simulator using the hybrid concept of Listing 11, omitting the body of operation *main* as it does not change.

```
@concept(name=ProcessHolderB,file=ProcHolderB.mdepth)
operation main() {
   ...
 }

operation getEnabled() : Set(&P) {
 return &P.allInstances().select(t |
        t.enabled()).asSet();
}

operation &P enabled() : Boolean {
 return self.inputHolders()->forAll(p|p.tokens()>0);
}

operation &P fire() {
 for (p in self.outputHolders()) p.addToken();
 for (p in self.inputHolders()) p.delToken();
}
```

Listing 12 Generic simulator over hybrid concept `TokenHolderB`.

A hybrid concept may require structural elements, in addition to Nodes, just like structural concepts. However, its power comes from being able to hide accidental details required from specific meta-models. In this way, the hybrid concept for Token-Holders has a higher-level of abstraction than the structural one, as it imposes less structural requirements to the bound meta-models. As a drawback, the meta-models are required to implement the operations specified in the concept. In MetaDepth, these operations are defined in a separate file which is indicated when establishing the binding. As an example, Listing 13 shows the binding of our hybrid concept to the Petri nets meta-model shown in Listing 2. Below, Listing 14 shows the implementation of the operations for the bound meta-model.

```
bind ProcessHolderB(PetriNet,
                  PetriNet::Place,
                  PetriNet::Transition)
   requires "PetriNetOperations.eol"
```

Listing 13 Binding the `ProcessHolderB` hybrid concept to the Petri nets meta-model.

```
operation Place tokens() : Integer {
 return self.tokens.size();
}
operation Place addToken() {
 self.tokens.add(new Token);
}
operation Place delToken() {
 self.tokens.remove( self.tokens.random() );
}
operation Transition inputHolders() : Set(Place) {
 return self.inPl;
}
operation Transition outputHolders() : Set(Place) {
 return self.outPl;
}
```

Listing 14 Operations needed to bind the hybrid concept `ProcessHolderB` to the Petri nets meta-model.

In order to illustrate the versatility of our hybrid concept, assume we want to bind it to a different meta-model for Petri nets where tokens are modelled with an integer field `tokens` in node `Place`. In this case the only difference with respect to the previous example is that the operations implemented in `Place` would be different. These operations are shown in Listing 15.

```
operation Place tokens() : Integer {
 return self.tokens;
}
operation Place addToken() {
 self.tokens := self.tokens+1;
}
operation Place delToken() {
 self.tokens := self.tokens-1;
}
```

Listing 15 Some operations needed to bind the hybrid concept `ProcessHolderB` to a variation of the Petri net meta-model where tokens are modelled as an integer field.

Altogether, hybrid concepts abstract from accidental details by encapsulating them in suitable operations, therefore being applicable to a larger set of meta-models. However, they leave more burden to the reusers of the generic operations, as they need to provide an implementation for the operations (even though sometimes they are straightforward, like in Listings 14 and 15). In this sense, hybrid concepts act as *interfaces* (in the sense of e.g. Java interfaces) for meta-models.

This situation is illustrated in Fig. 10, where the upper part shows the elements defined by the developer of the generic model management operation, and the lower part the elements that the user of the generic operation defines. In particular, the user must implement the required operations for his specific meta-model. The next subsection will show how to facilitate the binding of hybrid concepts to arbitrary meta-models by providing alternative generic implementations for the required operations, so that they can be directly reused.

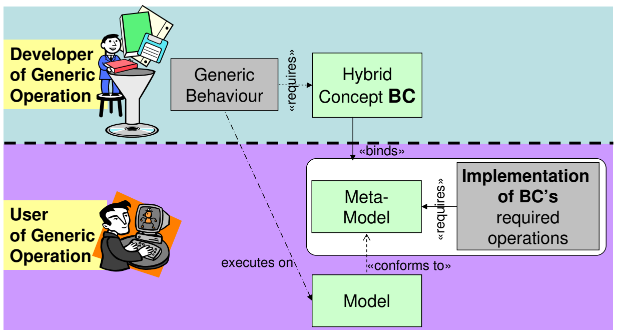

Fig. 10 Binding a hybrid concept.

6.1 Static bindings: Binding hybrid concepts to structural concepts

A hybrid concept can be bound to many structurally different meta-models. For instance, in our example, one meta-model may choose to represent tokens as an integer field, and a different meta-model as a reference to a node for the tokens. Each possibility will require the user of the concept to implement a different version of the concept's operations. In order to lighten this work, the developer of the concept may anticipate possible meta-model structures and implement the operations according to them. In particular, as Fig. 11 shows, he can build a structural concept for each foreseen meta-model, bind the hybrid concept to the structural one, and implement the operations using the types of the structural concept. We call such a binding from a hybrid to a structural concept a *static binding*. We also say that the structural concept and their operations *realize* the hybrid concept. From a technical perspective, this solution just makes use of another level of indirection to achieve its goals.

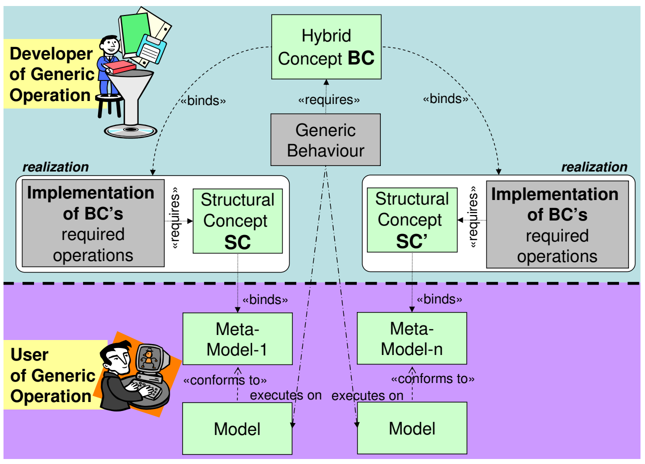

Fig. 11 *Realizations*: binding hybrid to structural concepts.

This approach has several advantages. First, users of the hybrid concept do not need to implement its operations, but they can select a suitable structural concept implementing them. If the user does not find a structural concept that fits his meta-model, then he will have to bind the hybrid concept and implement its operations. This implementation may be lifted to become a new structural concept realizing the hybrid one. Second, the generic model management operation is still defined only once, over the hybrid concept.

As an example, Listing 16 shows a structural concept for the Token-Holder semantics where tokens are represented as integers. The listing also includes the binding of the hybrid concept in Listing 11 to this new structural concept. Below, Listing 17 contains the generic implementation of the operations required by the hybrid concept in terms of the structural concept.

```
concept ProcessHolderInt(&M, &H, &P, &tokens,
                                &inHolders, &outHolders) {
  Model &M {
    Node &H {
      &tokens : int;
    }
    Node &P {
      &inHolders : &H[*];
      &outHolders: &H[*];
    }
  }
}

bind ProcessHolderB(ProcessHolderInt,
                    ProcessHolderInt::&H,
                    ProcessHolderInt::&P)
     requires "ProcessHolderIntOperations.eol"
```

Listing 16 Structural concept and static binding to hybrid concept.

```
operation &H tokens() : Integer {
  return self.&tokens;
}
operation &H addToken() {
  self.&tokens := self.&tokens+1;
}
operation &H delToken() {
  self.&tokens := self.&tokens-1;
}
operation &P inputHolders() : Set(&H) {
  return self.&inHolders;
}
operation &P outputHolders() : Set(&H) {
  return self.&outHolders;
}
```

Listing 17 Operations required by the hybrid concept expressed over the structural concept `ProcessHolderInt`.

The availability of several realizations for the same hybrid concept provides a kind of *overloading* mechanism for the required operations in the hybrid concept. In this sense, a generic operation defined over a hybrid concept is similar to the *template method* design pattern [23], as the required operations will be provided with an implementation either in realizations or when binding the concept to a specific meta-model. Altogether, realizations allow retaining the advantages of a more abstract and reusable concept (by using hybrid concepts), as well as an easy way to reuse the generic behaviour (by a suitable realization implementing the operations required by the hybrid concept).

6.2 Concept specialization

Similar to class or interface inheritance in object-oriented programming languages, we support *concept specialization* as a way to construct general/specific hierarchies of concepts, where the generic behaviour defined over a concept is applicable to all specializations of the concept as well. Concept specialization is also a means to construct concepts incrementally.

As an example, Listing 18 shows an extension of the hybrid concept `ProcessHolderB` defined in Listing 11. The purpose of the new concept is to characterize the Token-Holder semantics considering time, so that processes do not fire immediately but after a certain time elapse. The specialization `TimedProcessHolder` requires for this purpose an additional operation *getTime()*.

```
concept TimedProcessHolder(&M,&H,&P)
extends ProcessHolderB(&M,&H,&P){
  Model &M {
    Node &P{
      operation getTime() : Real;
    }
  }
}
```

Listing 18 Extending the hybrid concept `ProcessHolderB`.

In this way, we can apply the simulator for `ProcessHolderB` to instances of meta-models bound with concept `TimedProcessHolder`. This simulator makes an abstraction providing an untimed simulation of the model. One can define a simulator over the `TimedProcessHolder` concept as well. Actually, such a simulator needs to use events, event lists and further structures which are not required by the concept, but which are needed for the simulation. We will discuss in Section 8 the use of semantic mixin layers as a way to add such structure to meta-models consistent with `TimedProcessHolder`.

7 Model Templates

Concepts express requirements of models and meta-models. By using such an abstraction mechanism, behaviours and transformations can be expressed in a type independent way, becoming more reusable. However, genericity can be applied not only to behaviours, but to models and meta-models as well.

As already noticed in the example of Listing 3, it is desirable to have a means to define libraries of model fragments with well defined interfaces, so that users can be more productive when building models. With such a mechanism, models could be built by selecting, instantiating and interconnecting model fragments. In this section we show how *model templates* realize this idea. We use model templates to define reusable models, where their interface requirements are specified by means of concepts. Hence, compositionality is obtained without the need to modify the meta-models.

Up to now we have used concepts to express requirements for meta-models, but we can use them to define requirements for models as well. This is illustrated in Fig. 12. A concept C at the model level expresses a number of requirements that are fulfilled by a (possibly infinite) set of models $L(C)$. We say that any model in $L(C)$ can be bound to the concept C. The concept itself uses types from some meta-model (depicted as relation *"typed on"*), of which the models in $L(C)$ are instances. Whereas the relation *"conforms to"* indicates that the models use the types of the meta-model and satisfy all its integrity constraints, relation *"typed on"* indicates that the concept uses the types of the meta-model but it is not required to satisfy its integrity constraints. Thus, a concept for models is typed on a given meta-model but might violate, e.g., some of the minimum cardinality constraints imposed by the meta-model, hence not being conformant to it.

Model templates use concepts to express requirements on the parameters they receive. They declare a number of variables which can be checked against concepts. In this way, when the templates are instantiated, an implicit binding process checks whether the actual parameters satisfy the concepts. A template T requiring concept C defines a language $L(T)$ of all its possible instantiations using as parameters any element of $L(C)$, the language defined by the concept. In this way, a template can be seen as a function $L(C) \xrightarrow{T} L(T)$.

The possibility of *instantiating* templates is very interesting for modelling, because we can express patterns

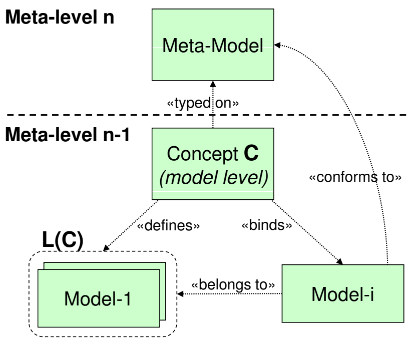

Fig. 12 Scheme of concepts for models.

and generic model components using templates, which we can later instantiate and combine. Such model templates define a generic interface through appropriate concepts expressing the requirements for a correct interconnection. This observation is depicted in Fig. 13. The picture shows a model template T, which is designed to be composable by importing another model, and possibly performing some connections to elements of that other model. Instead of specifying the concrete model to be imported, template T imports any model that satisfies the concept C. This concept expresses the requirements for composability of these two models. Genericity is obtained because we are able to compose template T with any model in $L(C)$. Instantiating T means choosing one model $M \in L(C)$ (i.e. a model bound to C) for the composition. The template definition and its instances are therefore at the same meta-level. In this way, concepts provide a non-intrusive means to express interface requirements, because there is no need to modify the meta-models. It is also possible that a model template uses several concepts for expressing requirements for several interfaces.

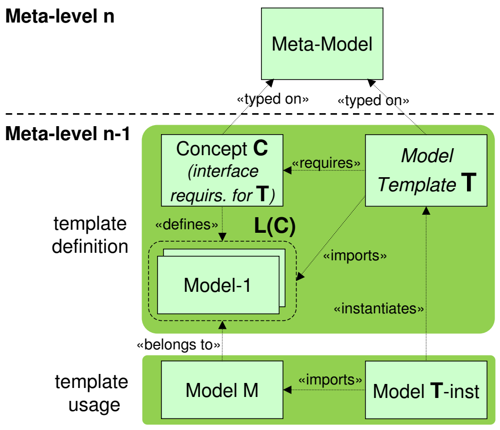

Fig. 13 Scheme of model templates.

Please note that instantiating a model template might yield a model that is not conformant to the meta-model. This is so as the template model typically imports some other variable models received as parameters, and the result of this import may violate some integrity constraint. While in MetaDepth one can rely on a verification procedure after the import operation takes place, alternatively, the concepts could also encode the conditions needed from the model parameters to yield a correct instantiation.

Consider again the Producer-Consumer Petri net model presented in Fig. 3. The model would benefit from a higher-level representation enabling the definition of processes (for the producer and the consumer) as well as of buffers. For this purpose we can define two model templates, acting like model components or modelling patterns that the modellers can use to construct their models.

Listing 19 shows how to specify these templates with MetaDepth. The first template `Buff2` (lines `7-14`) defines a generic buffer with one input and one output transition. These two transitions (`&Tri`, `&Tro`), together with their owning models (`&PNi`, `&PNo`), are parameters of the template. The template imports both received models (line `10`), declares one place (line `11`) and connects it to the received transitions (lines `12-13`). In addition, the template requires in lines `8-9` that the input parameters satisfy the concept `SimpleTrans`. The concept, defined in lines `1-5`, requires the transition to have one input and one output place, checked by the EOL constraint in line `5`.

```
 1 concept SimpleTrans(&M, &T) {
 2   PetriNet &M {
 3     Transition &T {}
 4   }
 5 } where $&T.inPl.size()=1 and &T.outPl.size()=1$
 6 // ---------------------------
 7 template<&PNi,&Tri,&PNo,&Tro>
 8 requires SimpleTrans(&PNi,&Tri),
 9          SimpleTrans(&PNo,&Tro)
10 PetriNet Buff2 imports &PNi,&PNo{
11   Place Buffer {}
12   ArcPT (Buffer, &Tro);
13   ArcTP (&Tri, Buffer);
14 }
15 // ---------------------------
16 template<>
17 PetriNet TwoStateProc {
18   Place p1 {}
19   Place p2 {}
20   Transition t12 {}
21   Transition t21 {}
22   ...
23 }
24 // ---------------------------
25 TwoStateProc<> Producer;
26 TwoStateProc<> Consumer;
27 Buff2<Producer,Producer::t12,Consumer,Consumer::t12>
28      ProducerConsumer;
```

Listing 19 Defining and using model templates.

The second template, `TwoStateProc` in lines `16-23`, defines a two-state process. In this case, the template has no parameters and acts like a pattern that can be instantiated by the modeller. In a realistic scenario, we may like to pass as parameters the names of the places, but currently MetaDepth does not support template parameters of primitive data types, which is left for future work.

Lines 25-28 instantiate the templates. The resulting model ProducerConsumer is equivalent to the one in Listing 3. However, the use of templates has raised the abstraction level of the model, which is now more concise, and we have reused the definition of the template TwoStateProc. Altogether, model templates enable defining component and pattern libraries for domain specific languages. Hence, a component designer would identify or design generic, useful model components that software engineers would be able to reuse and connect to build their models.

7.1 Generic model templates

Model templates are a way to capitalize on knowledge about useful domain-specific primitives, which are captured in terms of model fragments. They make use of concepts at the model level to express requirements on their interconnection interfaces. However, there is still room for further abstraction, as model templates still make use of types from specific meta-models. Hence, we can define *generic model templates*, which are templates that use the type variables from concepts. These templates express common patterns applicable to families of meta-models. The scheme of this approach is shown in Fig. 14.

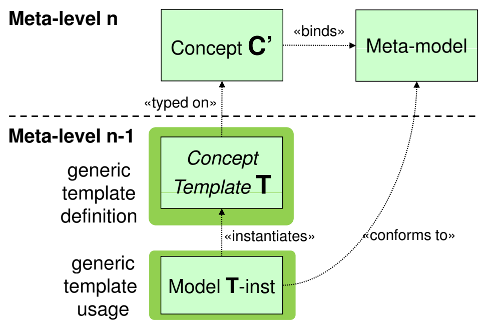

Fig. 14 Scheme of generic model templates.

As a difference to Fig. 13, in Fig. 14 the generic model template T is typed on a concept C', instead of on a specific meta-model. In this way, once we bind the concept C' to a specific meta-model, we can instantiate the generic model template T. Moreover, generic model templates are normally defined over structural concepts, in order to characterize as tightly as possible the represented family of meta-models.

Listing 20 shows the generalization of the two-state process template shown in Listing 19. This time the template is defined over the TokenHolder concept and hence can be applied to the ProductionSystem meta-model as well. This example shows that it is feasible to define patterns for domain-specific languages in a meta-model independent way.

```
template<>
requires TokenHolder(&M, &H, &P, &T, &tokens,
                          &inHolders, &outHolders)
&M TwoStateProc {
  &H p1 {}
  &H p2 {}
  &P t12 {}
  &P t21 {}
  ...
}
// ---------------------------
bind TokenHolder(ProductionSystem,
                 ProductionSystem::Conveyor,
                 ProductionSystem::Machine,
                 ProductionSystem::Part,
                 ProductionSystem::Conveyor::parts,
                 ProductionSystem::Machine::inConveyors,
                 ProductionSystem::Machine::outConveyors)
TwoStateProc<> Producer;
TwoStateProc<> Consumer;
Buff2<Producer,Producer::t12,Consumer,Consumer::t12>
      ProducerConsumer;
```

Listing 20 Defining and using generic model templates.

8 Meta-Model Templates and Semantic Mixin Layers

Templates are not only useful to define generic models, but can also be applied to meta-models in order to provide an extensible way of defining languages, similar to mixin layers [44]. In our context, a mixin layer is a meta-model containing a set of auxiliary elements, which are needed to implement some functionality. These mixin elements are added to a given meta-model by extending the elements passed as parameters of the mixin. Here we explore *semantic mixin layers*, which are *meta-model templates* declaring elements needed to express the behaviour of meta-models. These templates are complemented with behavioural specifications, defined over the generic types of the mixin.

In order to define the semantics of a language, it is often the case that its meta-model has to be extended with auxiliary classes and elements needed for the simulation. For example, when simulating an automaton, we need a pointer to the current state and the sequence of symbols to be parsed. When simulating an event system, we need a list of the scheduled events ordered by their simulation time. These extra elements are not part of the language, but of the simulation infrastructure. If the language for specifying the semantics is powerful enough, we can use it to create the required simulation infrastructure. For instance, EOL provides data structures like Collection or Map that can be used for that purpose. However, some specification languages lack this expressivity (e.g. graph transformation), so that in general, a simulation infrastructure needs to be modelled and rendered.

The working scheme of semantic mixins is shown in Fig. 15. It shows a mixin layer template T that is used to extend the meta-models of a semantic family characterized by concept C. Hence $L(C)$ contains all meta-models satisfying the concept requirements. The semantic mixin

T extends any such meta-models with appropriate extra features by using an extension mechanism similar to package merge [11,18]. However, instead of extending a particular meta-model in $L(C)$, the mixin extends any meta-model in $L(C)$, chosen when instantiating the mixin. Please notice that a mixin and its instantiation for a particular meta-model belong to the same meta-level. The mixin T has an associated behaviour, defined on the generic types of the mixin and its associated concept. This behaviour becomes available for any meta-model to which the mixin is applied.

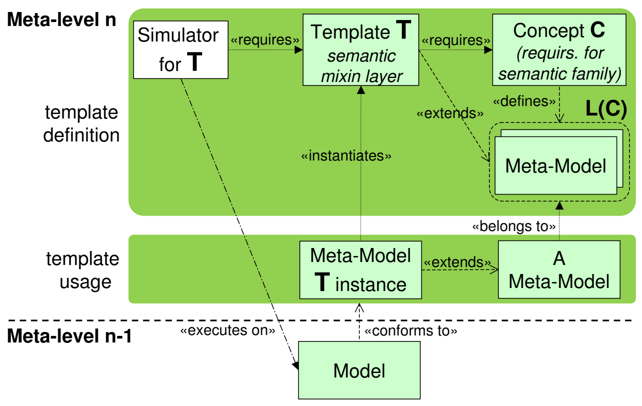

Fig. 15 Working scheme of semantic mixin layers.

As an example, assume we want to define a simulator for timed token-holder languages. These languages follow a Token-Holder semantics, but transitions fire after a given delay. Hence, we can characterize these languages with the `TimedProcessHolder` hybrid concept of Listing 18. The simulator needs storing a list of the firings that are currently scheduled, together with the transition and tokens involved in each firing. These extra elements are not part of the timed token-holder language, but devices needed only for the simulation. Hence, a separate mixin layer can incorporate these elements into the language definition in a non-intrusive way.

Lines `1-19` in Listing 21 show the template implementing the mixin layer. It declares the necessary infrastructure to simulate instances of meta-models that satisfy the concept `TimedProcessHolder`, therefore the template definition requires this concept. The template defines a family of meta-models which extend any meta-model `&M` satisfying concept `TimedProcessHolder` with the machinery needed for simulation. In particular, the template extends the received meta-model `&M` with a node `Event` to store the events, and a singleton node `FEvtList` to handle the event list (this is indicated with the cardinality interval [1] in line `7`). Moreover, the node with role `&P` (process) is added a collection `evts` storing the scheduled events associated to the transition.

```
 1 template <&M,&H,&P>
 2 requires TimedProcessHolder(&M,&H,&P)
 3 Model TimedSched extends &M {
 4   Node &P {
 5     evts: Event[*];
 6   }
 7   Node FEvtList[1] {
 8     first: Event[0..1];
 9     time : double;
10   }
11   Node Event {
12     time: double;
13     next: Event[0..1];
14     proc: &P;
15   }
16   Edge ProcTm(&P.evts, Event.proc) {
17     t : double;
18   }
19 }
20 // ----------------------------
21 TimedSched<ProductionSystem,
22            ProductionSystem::Conveyor,
23            ProductionSystem::Machine> SimProdSys
24 requires "ProdSystemsOps.eol"
```

Listing 21 Semantic mixin layer adding infrastructure to simulate concept `TimedProcessHolder`.

Now, assume we add a field `delay` to the `Machine` node in Listing 8, and define the operations required by the `TimedProcessHolder` concept. Then, the meta-model `ProductionSystem` (together with such operations) is a valid binding for the concept `TimedProcessHolder`, and hence we can instantiate the mixin layer for the meta-model in order to extend it with the simulation infrastructure. The instantiation is declared in lines `21-24` of Listing 21.

Behaviours associated to semantic mixin layers use the generic types of the template. Listing 22 shows an excerpt of the simulator associated to the `TimedSched` mixin layer. The simulator uses a `FEvtList` object (line `3`) to keep the current simulation time and the list of scheduled events. The list of events is initialized with the set of active transitions (line `5`). The main simulation loop (lines `7-11`) advances the simulation time to the time of the first event in the list, fires the transition associated to the event, and schedules new events (this latter is not shown in the listing).

```
 1 @template(name=TimedSched)
 2 operation main() {
 3   var FEL := new FEvtList;
 4   FEL.time := 0;
 5   FEL.schedule( getEnabled() );
 6   var finish: Boolean := false;
 7   while (not finish) {
 8     FEL.time:= FEL.first.time;
 9     FEL.first.proc.fire();
10     ...
11   }
12 }
```

Listing 22 Simulator for the `TimedSched` mixin layer (excerpt).

Associating the simulator to the mixin layer has the advantage that the simulator can be reused with any meta-model to which this mixin layer is applied (i.e. any meta-model fulfilling the `TimedProcessHolder` concept), like `SimProdSys` in Listing 21, hence obtaining highly reusable simulator specifications.

9 Further Examples

Next we provide further examples to illustrate the presented techniques and demonstrate their applicability.

9.1 Automata model templates

If we are interested in working with deterministic finite state automata (DFSA), we can define them as shown in Listing 23. We have opted for defining two separate meta-models, one with the definition of the input alphabet (containing the definition of symbols or events), and the other one with the automaton itself. The `id` annotation on the `value` field of `Symbol` (line `3`) ensures that each symbol has a different value. The `DFSA` meta-model imports the `Input` meta-model in line `7`, and includes two global constraints in lines `25–28` to ensure a unique initial state and one or more final states. The `State` node contains two local constraints (`nonRepSymb` and `allSymb`) in lines `14–18` to ensure determinism.

```
 1 Model Input {
 2  Node Symbol {
 3   value : String{id};
 4  }
 5 }
 6
 7 Model DFSA imports Input {
 8  Node State {
 9   name   : String {id};
10   ins    : State[*];
11   outs   : State[*];
12   initial: boolean = false;
13   final  : boolean = false;
14   noRepSymb: $self.Transitionouts.collect(t |
15              t.symbol).asSet().size()=
16              self.Transitionouts.size()$
17   allSymb: $Symbol.allInstances().size =
18              self.Transitionouts.size()$
19  }
20
21  Edge Transition(State.ins, State.outs) {
22   symbol : Symbol;
23  }
24
25  oneInitial : $State.allInstances().one(s |
26                s.initial=true)$
27  someFinal : $State.allInstances().exists(s |
28                s.final=true)$
29 }
```

Listing 23 Deterministic automata meta-model.

We can instantiate the DFSA meta-model to define automata. For instance, Listing 24 shows an automaton over the binary alphabet accepting binary words ending in 1.

```
 1 Input Binary {
 2  Symbol Zero { value = "0"; }
 3  Symbol One { value = "1"; }
 4 }
 5
 6 DFSA Accept2nd imports Binary {
 7  State Initial { name = "i"; initial = true; }
 8  State Final { name = "f"; final = true; }
 9
10  Transition t1(Initial,Initial) { symbol = Zero; }
11  Transition t2(Initial,Final) { symbol = One; }
12  Transition t3(Final,Initial) { symbol = Zero; }
13  Transition t4(Final,Final) { symbol = One; }
14 }
```

Listing 24 An automaton model.

However, one soon realizes that the `Accept2nd` model is unnecessarily concrete, in the sense that the same automaton would work with any alphabet with two symbols. Thus, we decide to convert the automaton into a template model that requires the input alphabet to have exactly two symbols, which is specified by means of a concept. Listing 25 shows the realization of this idea. Lines `1–6` define a concept requiring an alphabet with exactly two symbols, and lines `8–18` define a template model over the concept. The template is a generalization of the model in Listing 24. Then, the user of the generic automaton can easily instantiate the template using different alphabets, as lines `25–35` show. This approach provides additional flexibility, as it is easy to exchange the use of the different symbols in the alphabet. For example, the instantiation in lines `33–35` creates an automaton that accepts the zero-terminated binary words. Please notice that the meta-model remains unchanged, so that this template-based technique is non-intrusive.

```
 1 concept alpha2symb (&I, &S1, &S2) {
 2  Input &I {
 3   Symbol &S1{}
 4   Symbol &S2{}
 5  }
 6 } where $Symbol.allInstances().size()=2$
 7
 8 template<&I, &s1, &s2>
 9 requires alpha2symb(&I,&s1,&s2)
10 DFSA Accept2nd imports &I {
11  State Initial { name = "i"; initial = true; }
12  State Final { name = "f"; final = true; }
13
14  Transition t1(Initial,Initial) { symbol = &s1; }
15  Transition t2(Initial,Final) { symbol = &s2; }
16  Transition t3(Final,Initial) { symbol = &s1; }
17  Transition t4(Final,Final) { symbol = &s2; }
18 }
19
20 Input AlphaBeta {
21  Symbol Alpha { value = "a"; }
22  Symbol Beta { value = "b"; }
23 }
24
25 Accept2nd<AlphaBeta,
26        AlphaBeta::Alpha,
27        AlphaBeta::Beta> ab;
28
29 Accept2nd<Binary,
30        Binary::Zero,
31        Binary::One> zeroOne;
32
33 Accept2nd<Binary,
34        Binary::One,
35        Binary::Zero> oneZero;
```

Listing 25 Turning the automaton into a template.

Altogether, this example shows the usefulness of concepts in order to define model templates. Such model templates enable the construction of libraries of reusable model fragments.

9.2 Questionnaires and timed automata

In this section we demonstrate the incremental construction of meta-models, as well as the rapid definition of their semantics and associated behaviour through concepts and genericity.

Assume we want to build questionnaires made of questions with a number of answers, some of them being correct and redirecting to a new question until the end of the questionnaire is reached. A meta-model to describe such questionnaires is shown in Listing 26. This meta-model shares certain characteristics with the one for automata presented in previous section, as `Questions` can be interpreted as states, and `Answers` as transitions.

```
Model Questionnaire {
  Node Quiz {
    title : String;
    start : Question;
  }

  Node Question {
    text  : String;
    options : Answer[*];
  }

  Node Answer {
    ident : String;
    text  : String;
    correct : boolean = false;
    target : Question;
  }
}
```

Listing 26 Meta-model for questionnaires.

As previously stated, concepts allow the definition of generic simulators and code generators, applicable to instances of all meta-models to which we can bind the concepts. This can be used for the rapid definition of semantics for domain-specific languages, by defining suitable concepts and associated behaviour for semantic families. Up to now, we have defined concepts for Token-Holder semantics, automata semantics, timed automata, queueing networks and event-scheduling semantics. For example, Listing 27 shows a hybrid concept for automata-like languages for which we have built a simulator. The concept demands the existence of two kinds of node: states and events. The former must be equipped with operations to check whether states are initial or final, and to obtain its name and the target state given an event. Events must define an operation to obtain their identity, and another one to check whether two events are the same.

```
concept StateTransition(&M, &State, &Event) {
  Model &M {
    Node &State{
      operation isInitial() : boolean;
      operation isFinal() : boolean;
      operation getName() : String;
      operation getNext(e : &Event) : &State;
    }
    Node &Event{
      operation getId() : String;
      operation equivs(e: &Event) : boolean;
    }
  }
}
```

Listing 27 Concept for automata semantics.

The previous concept can be easily bound to the meta-models of Listings 23 and 26, once the concept operations are defined for the meta-models. For the latter meta-model, `Question` plays the role of `State` and `Answer` of `Event`.

Many times, timed semantics can be expressed as an extension of untimed semantics. For example, Listing 28 presents a concept expressing the commonalities of languages behaving like simple timed automata, which we call automata with timeout. These are an extension of normal automata with special transitions having a timeout. The automaton is forced to take a timeout transition when it stays in the source state a number of time steps equal to the specified timeout. Moreover, transition labels can be input or output, enabling the synchronization of concurrent automata through the same transitions. These automata are a simplification of the classical timed automata [1]. In our case, we have built the `TimedStateTransition` concept incrementally, by extending concept `StateTransition`.

```
load "StateTransition"

concept TimedStateTransition(&M, &State, &Event)
extends StateTransition(&M, &State, &Event)
{
  Model &M {
    Node &State{
      operation getMaxDelay() : Integer;
      operation getTimeoutState() : &State;
    }
    Node &Event{
      operation isInput() : boolean;
    }
  }
}
```

Listing 28 Concept for timed automata semantics.

In a similar way, we can define a meta-model template like the one in Listing 29 to increment meta-models conforming to the `StateTransition` concept with the necessary timing elements. The template extends `State` with a maximum time and a timeout state. For events, it adds a flag indicating whether they are input or output events, so that we can build networks of models synchronized through events. Lines `15-20` show two instantiations of the template with the `Questionnaire` and `DFSA` meta-models. This permits designing questionnaires with a maximum time to respond each question. If such time is consumed, the user is redirected conveniently to a different question. Moreover, we can use the extended DFSA meta-model to describe *user models* responding to the questionnaires. In particular, users will produce answers (output events) which will get synchronized with the answers that the questionnaire expects (input events).

```
1  template <&M,&State,&Event>
2  requires StateTransition(&M, &State, &Event)
3
4  Model TimeExt extends &M {
5    Node &State {
6      maxTime : int = 0;
7      timeOut : &State[0..1];
8    }
9
10   Node &Event {
11     isInput : boolean = false;
12   }
13 }
14
```

```
TimeExt <Questionnaire,
         Questionnaire::Question,
         Questionnaire::Answer> TimedQuiz;
TimeExt <DFSA,
         DFSA::State,
         DFSA::Symbol> TimeoutAutomata;
```

Listing 29 Meta-model template for timeout automata.

We must implement the operations of the `TimedStateTransition` concept in order to bind it with the *TimedQuiz* and *TimeoutAutomata* meta-models in the previous listing. These operations can be defined over the template, obtaining a static binding as explained in Section 6.1. In this way, we can simulate both meta-models by using our generic simulator. Alternatively, we can use a generic code generator defined over the concept in order to produce code for analysis tools, like UPPAAL [5]. This is a tool enabling the visual simulation of timed automata, as well as its analysis using temporal logic. Listing 30 shows an excerpt of the generic code generator which is in charge of generating appropriate tags to position the nodes of the automata. Being this generator defined over the `TimedStateTransition` concept, we can synthesize code from instances of both *TimedQuiz* and *TimeoutAutomata*.

```
[%
@concept(name=TimedStateTransition,file=TST.mdepth)
%]
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE nta PUBLIC '-//Uppaal Team//...'
                'http://www.it.uu.se/...flat-1_1.dtd'>
<nta>
 ...
 [% var ident : Integer := 0;
    var yPos : Integer := -166;
    var initIdent : Integer := 0;
    for (s in &State.allInstances()) {
      yPos := yPos+ident*60; %]
   <location id="id[%=ident%]" x="-224" y="-136">
     <name x="-234" y="[%=yPos%]">Q[%=ident%]</name>
   </location>
 [%   s.~identif := ident; s.~yPos := yPos;
      if (s.isInitial()) { initIdent := ident; }
      ident := ident+1;
    } %]
 ...
</nta>
```

Listing 30 Generic code generator for concept `TimedStateTransition`.

Timed automata have compositional semantics as we can build networks of automata that synchronize by sending and receiving events. Our `TimedStateTransition` concept permits composition without being tied to the specific meta-model used for modelling. Hence, we can synthesize code from a network of models built using the *TimeQuiz* meta-model for the questionnaires and the *TimeoutAutomata* meta-model for the user behaviour. Afterwards, we can use our generic code generator to synthesize code for UPPAAL and perform analysis. Fig. 16 shows a screenshot of UPPAAL being used to analyse a questionnaire with a particular user model. We have used the analysis capabilities of UPPAAL to check, for example, whether a particular questionnaire is solvable with a given user strategy.

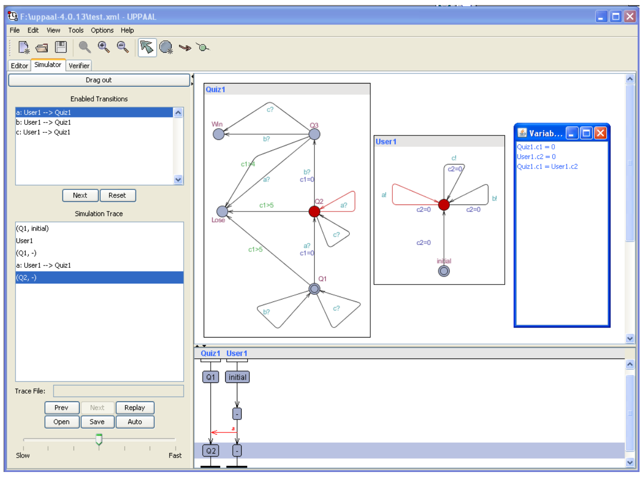

Fig. 16 Analysis of a Quiz+User model using UPPAAL.

Altogether, this example shows how to incrementally augment meta-models with additional elements, e.g. incorporating them additional structure to express more sophisticated semantics. In particular, we have seen that timing semantics can be incorporated in this way to untimed formalisms. We have also seen that concepts can be extended in a similar way (e.g. the `TimedStateTransition` concept extends `StateTransition`). The definition of generic behaviours and code generators allow their application to families of meta-models, obtaining interoperability as well.

10 Related Work

The use of templates in modelling is not new. They are already present in the UML 2.0 specification [39], as well as in approaches like Catalysis' model frameworks [19] and package templates, and the aspect-oriented meta-modelling approach of [11]. Interestingly, while all of them consider templates for meta-models or class diagrams, none consider concepts to express the requirements of type parameters.

The UML 2.0 proposes classifier (e.g. class, component), package, collaboration and operation templates which are provided with a list of formal parameters representing classifiers, values or features (i.e. properties and operations). A template binding specifies the substitution of actual parameters for the formal parameters of the template, and has the same semantics as if the contents of the template were copied into the bound element, substituting the formal template parameters by the corresponding actual parameters in the binding [39]. Hence, UML lacks support to express requirements for the formal parameters in a non-intrusive way, as supported by the notion of *concept* we have presented here.

In the context of UML 2, concepts would be a valuable means to express the requirements that parameter instantiations should fulfil in order for a template binding to be correct. Currently, this can be achieved only by requiring that some formal parameter conforms to a specific class, in a similar way as in Java, where a parameter may be required to implement a certain interface. However, if the template has several parameters, it is often not sufficient to demand requirements for each one of them in isolation, but for the set of parameters as a whole. Notice also that the genericity provided by UML is mainly directed to *generic models*, but our approach also allows the definition of *generic behaviours*. Although package templates were incorporated into the UML 2.0 specification, the MOF [40] does not consider genericity at the meta-model or model level.

Catalysis' model frameworks [19] are parameterized packages that can be instantiated by name substitution. Hence, they are similar to our meta-model templates. The package templates of [11] are based on those of Catalysis, and are used to define languages in a modular way. They are based on string substitution, as the template parameters are strings that are substituted in the template definition. This approach is realized in the XMF tool [12].

Our work extends the mentioned approaches in several ways. First, we can apply templates not only to meta-models, but also to models, as seen in Section 7 (cf. Listing 19). Actually, as our framework supports an arbitrary number of meta-models through potency [14], we could apply templates at any meta-level. Second, our approach is based on *concepts*, which helps in expressing requirements on template parameters. In addition, we can define behaviour for concepts and templates (in particular with semantic mixin layers), independently of meta-models. Third, our approach provides a stronger support for templates, as our template parameters are model elements whose requirements can be expressed by *concepts*. This permits type checking at the template level. Finally, whereas we consider the definition of generic behaviour, this is missing in other works [11,19,39]. This paper also extends our own previous work on adding genericity to a model management framework [22,41], where we explored the use of structural concepts to define model transformation operations, but we did not explore more flexible forms of concepts like hybrid concepts or genericity for models and meta-models.

Even though our work is strongly influenced by the generic programming community, generic (meta-)modelling has fundamental differences with generic programming [25,29]. The first one refers to the level of granularity, as generic programming deals with generic classes or functions, whereas we consider generic (meta-)models which include several modelling elements, more similar to mixin layers. Second, while the purpose of programming concepts is to identify whether a class defines certain operations, structural concepts check structural properties of models. Our hybrid concepts are similar to declarations of Java interfaces, but where operations are defined for a collection of classes.

With respect to the binding, generic programming proposes either an automatic binding of concepts, or a manual one through concept maps [29]. The latter allow an explicit mapping between concept and class operations, and can include code for the required concept operations. In [25], the authors propose concept-based overloading by defining operations with same signature in different concepts. Then, it is possible to define a family of overloaded templates, each requiring a different concept. In this way, the most specific implementation of the required operation will be selected depending on the input type provided. Instead, we propose *realizations* of hybrid concepts by several structural ones. In this way, the user of the generic behaviour will select the most appropriate structural concept fitting his meta-model. Finally, the generic programming community has proposed the specification of *axioms* defining properties for the concept operations, like commutativity or associativity. Compilers can use these properties for several purposes, like testing or optimization [4]. We believe this idea could also be brought to our hybrid concepts by using a constraint language such as OCL to express the properties.

Another non-intrusive way to reuse model management operations is by structural subtyping mechanisms [10,45]. Structural subtyping permits defining generic behaviours over an arbitrary meta-model, and applying them to any meta-model that is found to be a subtype. The subtype-of relation between the meta-models does not need to be declared as in nominal subtyping, but it is automatically inferred like in the Kermeta system [30]. This approach has been applied for example to generic refactoring [35]. In contrast, our approach requires providing an explicit binding between concepts (supertype) and meta-models (subtypes). This enables a fine control of the part of the meta-model to be bound to a concept, as several bindings may be possible. Hence, an explicit binding is preferred for small concepts that might be bound in many different ways to a meta-model (e.g. a concept used to calculate the transitive closure of a relation would only contain one class and one relation). Instead, Kermeta's structural subtyping frees the developer from specifying bindings, but at the price of less control. This approach is more adequate when the supertypes are "big" and there are few ways in which a meta-model can be a subtype of the supertype. In general, we believe that our notion of *concept* and *binding* could be easily adapted for this approach. Thus, we will explore partial bindings and (semi-)automatic completion mechanisms. We could also use Kermeta's approach to meta-model pruning [43] to automatically derive structural concepts given a concrete operation that we want to make generic. Nonetheless, note that concepts can be thought as representatives of (meta-)model types. Such explicit representation enables their use for expressing requirements of mixins and model template parameters, which is not possible with structural subtyping.

Another set of related research are the (meta-)model modularization approaches, like *Reuseware* [26]. In this approach, the authors develop a language-independent composition language, which can be used to define composition interfaces for models, in an intrusive way. While *Reuseware* solves the modularization of models, our templates provide in addition an instantiation mechanism, suitable to construct patterns and component libraries. In addition, [26] does not consider generic behaviours and lacks abstraction mechanisms like concepts.

Parameterized modules were proposed in algebraic specification in the eighties [21]. A parameterized module is usually represented with a morphism $par\colon P \to M$ from the formal parameters to the module. This approach was updated in [50] in order to define parameterized MOF-based meta-models. While we will take as inspiration these previous approaches to build a formalization of our approach, in this paper we propose using *concepts* to restrict how the formal parameters can be bound to the actual parameters in both mixins and model templates.

Our approach also has some similarities with Aspect Oriented Modelling (AOM) [31]. AOM focuses on modularizing and composing crosscutting concerns within software models. These concerns are expressed using template models, frequently class or sequence diagrams, which have some type parameters. Some aspects have *pointcuts* expressing conditions that enable the application of the aspect (the *advice*). Hence, one could interpret our model templates as aspects, where the template parameters and the concept act like pointcuts, and the body of the template acts like an advice. The application of our approach for AOM is left for future work.

Finally, there are several tools supporting multiple meta-levels, like DeepJava [34] and the approach of [2]. Forms of multi-level meta-modelling can be traced back to knowledge-based systems like Telos [38] and deductive object base managers like ConceptBase [28]. However, none of those systems consider genericity explicitly.

11 Conclusions and Future Work

In this paper we have shown the benefits of bringing concepts, templates and mixin layers into MDE. Concepts allow expressing requirements of template parameters, and by themselves permit defining behaviour independently of meta-models, hence becoming more reusable. Templates can be applied to models or meta-models and promote extendibility, modularity and reusability. At the model level, they are useful to define patterns and model component libraries. At the meta-model level, mixin layers are especially useful to provide the necessary infrastructure to simulate and execute models. We have shown how the MetaDepth tool [14] provides support for all these elements, however the discussions in this paper are general and applicable to other contexts and tools as well.

We believe the semantics of many modelling languages can be classified using concepts. Hence, we plan to continue defining concepts for other kinds of semantics, like communication semantics or process-interaction semantics. Moreover, the combination of concepts and semantic mixin layers will provide support for the rapid prototyping of language semantics.

We are currently exploring the potential opened by genericity, for instance to build pattern libraries for domain-specific languages through model templates, or to define any kind of model management operation like model-to-model transformations. We are also working on a formalization of our approach to investigate bindings enabling different degrees of type-safety for given operations, potential issues with non-injective bindings, the conditions under which template instantiation yields conformance, as well as the use of genericity elements with formal transformation languages like graph transformation. With such a formalization, we aim at deriving proof mechanisms for type safety similar to those in [45], as well as the correctness of mixin and model template instantiation and composition. We will also explore the implications and usage patterns of genericity in a multi-level meta-modelling setting with more than two meta-levels, as well as the usefulness of our approach for AOM.

Regarding tool support, we are currently improving the MetaDepth support for genericity, in particular to allow primitive types as template parameters, the definition of axioms for hybrid concepts, the extension of several meta-models by a mixin, and more flexible bindings including partial bindings and their (semi-)automated completion.

Acknowledgements. We thank the referees for their useful comments. This work has been sponsored by the Spanish Ministry of Science and Innovation with projects METEORIC (TIN2008-02081) and *Go Lite* (TIN2011-24139), and by the R&D program of the Community of Madrid with project “e-Madrid” (S2009/TIC-1650).

References

1. R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
2. C. Atkinson, M. Gutheil, and B. Kennel. A flexible infrastructure for multilevel language engineering. *IEEE Trans. Soft. Eng.*, 35(6):742–755, 2009.
3. C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.
4. A. H. Bagge and M. Haveraaen. Axiom-based transformations: Optimisation and testing. *Electr. Notes Theor. Comput. Sci.*, 238(5):17–33, 2009.
5. G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0.

Quantitative Evaluation of Systems, International Conference on, 0:125–126, 2006. See also `http://www.uppaal.org/`.

6. P. Bonet, C. Llado, R. Puijaner, and W. Knottenbelt. PIPE v2.5: A Petri net tool for performance modelling. In *CLEI'07*, 2007. `http://pipe2.sourceforge.net/`.
7. Boost. `http://www.boost.org/`.
8. P. Bottoni, E. Guerra, and J. de Lara. Enforced generative patterns for the specification of the syntax and semantics of visual languages. *J. Vis. Lang. Comput.*, 19(4):429–455, 2008.
9. BPMN. `http://www.bpmn.org/`.
10. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17:471–523, December 1985.
11. T. Clark, A. Evans, and S. Kent. Aspect-oriented metamodelling. *The Computer Journal*, 46:566–577, 2003.
12. T. Clark, P. Sammut, and J. Willans. *Applied Metamodelling, a Foundation for Language Driven Development, 2nd edition.* Ceteva, 2008.
13. CPNTools. `http://wiki.daimi.au.dk/cpntools`.
14. J. de Lara and E. Guerra. Deep meta-modelling with MetaDepth. In *TOOLS'10*, volume 6141 of *LNCS*, pages 1–20. Springer, 2010. See also `http://astreo.ii.uam.es/~jlara/metaDepth/`.
15. J. de Lara and E. Guerra. Generic meta-modelling with concepts, templates and mixin layers. In *MoDELS (1)*, volume 6394 of *LNCS*, pages 16–30. Springer, 2010.
16. J. de Lara, E. Guerra, and P. Bottoni. Triple patterns: Compact specifications for the generation of operational triple graph grammar rules. In *GT-VMT'07*, volume 6 of *Electronic Communications of the EASST*, 2007.
17. J. de Lara and H. Vangheluwe. Automating the transformation-based analysis of visual languages. *Formal Aspects of Computing*, 22:297–326, 2010.
18. J. Dingel, Z. Diskin, and A. Zito. Understanding and improving UML package merge. *SoSyM*, 7:443–467, 2008.
19. D. F. D'Souza and A. C. Wills. *Objects, components, and frameworks with UML: the catalysis approach.* Addison-Wesley Longman Publishing Co., Inc., 1999.
20. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of algebraic graph transformation.* Springer-Verlag, 2006.
21. H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints.* Springer, Monographs in Theor. Comp. Sci., 1990.
22. Epsilon. `http://www.eclipse.org/gmt/epsilon/`, 2009.
23. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.
24. R. García, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. *SIGPLAN Not.*, 38(11):115–134, 2003.
25. D. Gregor, J. Järvi, J. G. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA*, pages 291–310. ACM, 2006.
26. F. Heidenreich, J. Henriksson, J. Johannes, and S. Zschaler. On language-independent model modularisation. *T. Asp.-Oriented Soft. Dev. VI*, 6:39–82, 2009.
27. L. Hillah, E. Kindler, F. Kordon, L. Petrucci, and N. Tréves. A primer on the petri net markup language and iso/iec 15909-2. *Petri Net Newsletter*, 76:9–28, October 2009. See also `http://www.pnml.org`.
28. M. Jarke, R. Gallersdörfer, M. A. Jeusfeld, and M. Staudt. Conceptbase - a deductive object base for meta data management. *J. Intell. Inf. Syst.*, 4(2):167–192, 1995.
29. J. Järvi, M. Marcus, and J. N. Smith. Programming with C++ concepts. *Sci. Comput. Program.*, 75(7):596–614, 2010.
30. Kermeta. `http://www.kermeta.org/`.
31. J. Kienzle, W. A. Abed, F. Fleurey, J.-M. Jézéquel, and J. Klein. Aspect-oriented design with reusable aspect models. *T. Aspect-Oriented Software Development 7*, 6210:272–320, 2010.
32. D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Object Language (EOL). In *ECMDA-FA'06*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.
33. T. Kühne. An observer-based notion of model inheritance. In *MoDELS'10*, volume 6394 - Part I of *LNCS*, pages 31–45. Springer, 2010.
34. T. Kühne and D. Schreiber. Can programming be liberated from the two-level style? – Multi-level programming with DeepJava. In *OOPSLA'07*, pages 229–244. ACM, 2007.
35. N. Moha, V. Mahé, O. Barais, and J.-M. Jézéquel. Generic model refactorings. In *MoDELS'09*, volume 5795 of *LNCS*, pages 628–643. Springer, 2009.
36. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
37. D. R. Musser, S. Schupp, and R. Loos. Requirement oriented programming. In *Generic Programming*, volume 1766 of *LNCS*, pages 12–24. Springer, 1998.
38. J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM Trans. Inf. Syst.*, 8(4):325–362, 1990.
39. OMG. UML 2.2 specification. `http://www.omg.org/spec/UML/2.2/`.
40. OMG. MOF 2.0. `http://www.omg.org/spec/MOF/2.0/`, 2009.
41. L. Rose, E. Guerra, J. de Lara, A. Etien, D. S. Kolovos, and R. F. Paige. Genericity for model management operations. *SoSyM*, In press, 2011.
42. L. M. Rose, R. F. Paige, D. S. Kolovos, and F. Polack. The Epsilon Generation Language. In *ECMDA-FA'08*, volume 5095 of *LNCS*, pages 1–16. Springer, 2008.
43. S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Metamodel pruning. In *MoDELS*, volume 5795 of *LNCS*, pages 32–46, 2009.
44. Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.
45. J. Steel and J.-M. Jézéquel. On model typing. *SoSyM*, 6(4):401–413, 2007.
46. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework, 2nd Edition.* Addison-Wesley Professional, 2008.
47. A. Stepanov and M. Lee. The standard template library. Technical report, HP Labs, 1994.

48. A. Stepanov and P. McJones. *Elements of Programming*. Addison Wesley, 2009.
49. B. Stroustrup. The C++0x remove concepts decision. *Dr.Dobbs*, 2009. `http://www.ddj.com/cpp/218600111`.
50. I. Weisemöller and A. Schürr. Formal definition of MOF 2.0 metamodel components and composition. In *MoDELS'08*, volume 5301 of *LNCS*, pages 386–400. Springer, 2008.