

# Generic Meta-modelling with Concepts, Templates and Mixin Layers

Juan de Lara<sup>1</sup> and Esther Guerra<sup>2</sup>

<sup>1</sup> Universidad Autónoma de Madrid, Spain  
Juan.deLara@uam.es

<sup>2</sup> Universidad Carlos III de Madrid, Spain  
eguerro@inf.uc3m.es

**Abstract.** Meta-modelling is a key technique in Model Driven Engineering, where it is used for language engineering and domain modelling. However, mainstream approaches like the OMG's Meta-Object Facility provide little support for abstraction, modularity, reusability and extendibility of (meta-)models, behaviours and transformations.

In order to alleviate this weakness, we bring three elements of *generic programming* into meta-modelling: *concepts*, *templates* and  *mixin layers*. Concepts permit an additional typing for models, enabling the definition of behaviours and transformations independently of meta-models, making specifications reusable. Templates use concepts to express requirements on their generic parameters, and are applicable to models and meta-models. Finally, we define functional layers by means of *meta-model mixins* which can extend other meta-models.

As a proof of concept we also report on METADEPTH, a multi-level meta-modelling framework that implements these ideas.

## 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) [11], which proposes a linear, 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) [13]. However, even though meta-modelling is becoming increasingly used on an 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 transformations.

Generic programming [5,14] 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 promotes the abstraction of algorithms and types by lifting their details from concrete examples to their most abstract form [14]. The advantage is that any type that fulfils the algorithm's requirements can reuse such a *generic* algorithm. Hence, generic

programming shifts the emphasis from type-centric to requirements-centric programming [9], enhancing generality and reusability.

In this paper we bring into meta-modelling and language engineering 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 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 meta-models.

Furthermore, models often suffer from an early concretization of details which hinders their reusability. The use of model *templates* allows delaying some details on the model structure by defining model parameters. In this way, a model template could be instantiated with different parameters, allowing its reusability in different situations, and enhancing its compatibility and modularity. Model templates are also a mechanism to implement patterns for domain-specific languages and model component libraries.

Finally, *mixins layers* [12] allow defining meta-models with generic functional capabilities to be plugged into different meta-models. We found especially useful the definition of *semantic mixins layers* containing the necessary run-time infrastructure for the definition of semantics of meta-model families.

As a proof of concept, we have implemented these elements in a multi-level meta-modelling framework called METADEPTH [3]. However, our aim is not to describe an extension of METADEPTH's capabilities. We believe that *genericity* has a wide potential in meta-modelling, and hence what we describe here has immediate applicability to other frameworks, like the MOF. A prototype of the tool can be downloaded from <http://astreo.ii.uam.es/~jlara/metaDepth/>.

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. Section 4 presents generic concepts for meta-modelling, Section 5 presents model templates and Section 6 introduces semantic mixin layers. Section 7 discusses related research and Section 8 concludes.

## 2 Generic Programming

Genericity [5] is a programming paradigm found in 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 use in software construction. It involves expressing algorithms with minimal assumptions about data abstractions, as well as generalizing concrete algorithms without losing efficiency [5]. It 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.

In its basic form, generic programming involves passing type parameters to functions or data types which are then called *templates*. In this way, template functions or classes require the parameter types to fulfil a number of requirements for a correct instantiation of the template. This set of requirements is usually expressed using a *concept* [9]. Typical requirements include, e.g., 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. Rudimentary concepts exist e.g. in Java, 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.

As an example, Listing 1 shows a C++ template function *min* that returns the minimum of two elements of a parametric type *T*. The requirement for the type *T* is to define the “<” operator, specified by concept `LessThanComp`<sup>1</sup>.

```

1 template <typename T> requires LessThanComp<T>
2   T min(T x, T y) { return y < x ? y : x; }
3 concept LessThanComp <typename T> { bool 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 [12] 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 layers.

### 3 METADEPTH

METADEPTH [3] is a new multi-level meta-modelling framework with support for multiple meta-levels at the same time using *potency* [1]. 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 [3]. The framework uses a textual syntax and is integrated with the Epsilon family of languages<sup>2</sup>, so that EOL [7] can be used to express constraints and behaviours. EOL extends OCL with imperative constructs to manipulate models.

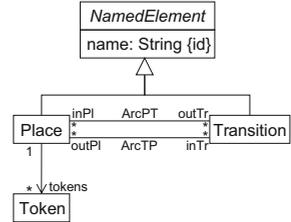
<sup>1</sup> Concepts have been post-poned from C++0x, the last revision of C++.

<sup>2</sup> <http://www.eclipse.org/gmt/epsilon/>

```

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

```



**Listing 2.** Meta-model for Petri nets, in METADEPTH’s syntax and UML.

Listing 2 shows a meta-model for Petri nets using METADEPTH’s syntax to the left and a UML representation to the right. Petri nets are a kind of automaton with two kinds of vertices: *Places* and *Transitions*. Places contain tokens, and can be connected with transitions through arcs (and the other way round).

The listing declares an abstract node `NamedElement` owning a field `name`. 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 association ends (`outTr`, `inTr` and `tokens`) with cardinality `0..*`. 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`. 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).

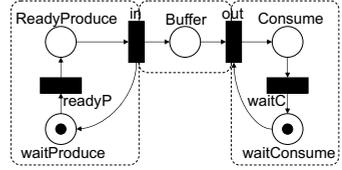
METADEPTH supports the definition of constraints and derived attributes either in Java or in EOL. Constraints can be declared in the context of `Models`, `Nodes` or `Edges`. Line 17 in the listing declares an EOL constraint named `minPlaces`, which demands `PetriNet` models to have at least one `Place`. As METADEPTH allows specifying multiplicities in the definition of `Nodes`, the same effect can be obtained by replacing line 5 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. The right of Listing 3 shows the system using the usual Petri nets visual notation, with places represented

```

1 PetriNet ProducerConsumer {
2   Place WP { name="waitProduce"; }
3   Place RP { name="ReadyProduce"; }
4   Transition ReadyP { name="readyP"; }
5   Transition Produce { name="in"; }
6   ArcPT (RP, Produce);
7   ...
8   Place Buffer { name="Buffer"; }
9   ...
10  Place C { name="Consume"; }
11  Place WC { name="waitConsume"; }
12  Transition Consume { name="out"; }
13  Transition ReadyC { name="waitC"; }
14  ...
15 }

```



**Listing 3.** A Petri net with the Producer-Consumer example.

as circles, transitions as black rectangles, and tokens as black dots inside places. The dotted rectangles delimit the different conceptual components of the system.

Listing 3 makes use of the normal instantiation capabilities found in most meta-modelling frameworks (like EMF [13]). However, one soon notices that the definition of our model could be improved concerning abstraction, modularity and reusability. 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. Section 5 will demonstrate how the use of templates allows performing this *at the model level*, without any need to modify the meta-model.

Finally, METADEPTH allows defining behaviour for models using either Java or EOL. 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 an example simulator written in EOL to execute Petri net models. The entry point for its execution is the operation *main*, which is annotated with the required meta-models (*PetriNet* in our case). The listing declares the auxiliary operations **enabled** and **fire** on the context of the *Transition* meta-class. These are invoked in the loop of the *main()* operation, firing the enabled transitions.

This simulator works well for instances of the Petri net meta-model. However, there are many languages whose behaviour can be defined in terms of Petri nets. Therefore, couldn't we abstract the essential elements of Petri net-like languages and define such behaviour in a generic way? Next section will show that concepts are a solution to this issue.

```

1 @metamodel(name=PetriNet,file=PetriNet.mdepth)
2 operation main() {
3   while (Transition.allInstances()->exists(t | t.enabled() and
4     t.fire())) {}
5 }
6 operation Transition enabled() : Boolean {
7   return self.inP1->forall(p | p.tokens.size(>0);
8 }
9 operation Transition fire() : Boolean {
10  for (p in self.outP1)
11    p.tokens.add(new Token);
12  for (p in self.inP1) {
13    var t : Token := p.tokens.random();
14    p.tokens.remove(t); delete t;
15  }
16  return true;
17 }

```

Listing 4. A simulator for Petri nets.

## 4 Concepts for Language Engineering

Now we are in the position to discuss how to bring elements of generic programming into meta-modelling. This section shows how to define *concepts*, and how to use them to define generic behaviours applicable to language families. The following two sections will discuss model and meta-model templates.

A *concept* in meta-modelling is a pattern specification that expresses requirements on models or meta-models. Concepts provide a dual typing in the context where they are used, which we use to define behaviour independently of specific meta-models. This is useful for reusability and composition of behaviours.

Let's start discussing an illustrative scenario. Assume one needs to describe the behaviour of two languages, one in the domain of Production Systems (where pieces are produced and consumed by machines) and the other for Communication Networks (where packets are produced and consumed by computers). Thus, one would define one program to simulate the first kind of models, and another different one to simulate the second kind of models. This situation is illustrated to the left of Fig. 1. In the figure we assume that behaviours are realized using EOL programs, but our approach is applicable to other means of specification of in-place model transformations, like e.g. graph transformation.

Analysing the semantics of these two languages reveals similarities between the two programs implementing them. Actually, 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 and define the behaviour for the common language only once. This situation is depicted to the right of Fig. 1, where `Model 1` is transformed into `Model 1'` and `Model 2` is transformed into `Model 2'`, being

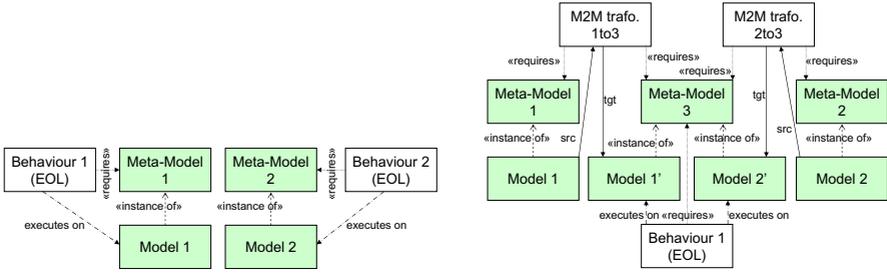


Fig. 1. Direct implementation of behaviours (left). Transformational approach (right).

both transformed models conformant to the same meta-model for which the behaviour is specified. However, 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 model according to the behaviour, this has to be translated back to the original language.

An improvement that avoids transforming models is to use extension or inheritance mechanisms for meta-models (see the left of Fig. 2). In this case, the meta-models 1 and 2 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 that all defined meta-models for which we want to define the semantics to inherit or extend the same meta-model. This may become unfeasible if more than one semantics (e.g. timed and untimed) is to be defined for the same language.

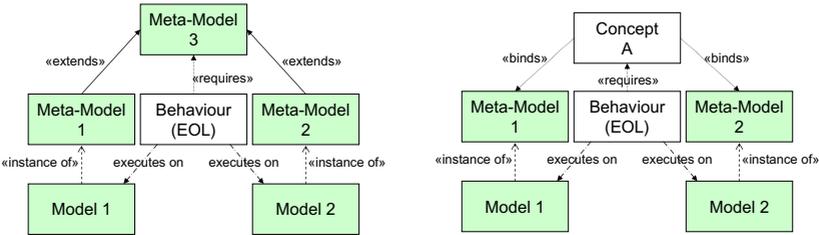


Fig. 2. Inheritance of behaviour (left). Approach based on concepts (right).

In this scenario, *concepts* can simplify the situation as they can express requirements on meta-models or models that some other elements (in this case the behaviour) needs. In our example, we can define a concept expressing the requirements that a simulator for Petri net-like languages needs. Such 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 to the right of Fig. 2. 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.

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 models and 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  $C$ . If  $C$  is defined at the meta-model level,  $L(C)$  contains a family of meta-models sharing similar characteristics. We use concepts to define generic behaviours 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, and thus they can impose a certain structure for their nodes, edges and fields, as well as define arbitrary constraints to restrict their applicability.

Listing 5 shows a concept, using METADEPTH’s syntax. The concept characterises languages with similar structural features, enabling their simulation using the same semantics, which we call **Token-Holder**. 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 an attribute of type  $\&T$ . Node  $\&P$  plays the role of a process or transition, and it must define two fields 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 an attribute `name:String`.

```

1 concept TokenHolder(&M, &H, &P,      6      Node &P {
1     &T, &tokens, &inHolders,      7         &inHolders : &H[*];
1     &outHolders) {                8         &outHolders: &H[*];
2     Model &M {                    9     }
3     Node &H {                     10    Node &T {}
4         &tokens : &T[*];          11    }
5     }                             12 }

```

**Listing 5.** A concept expressing the requirements for **Token-Holder** semantics.

We use this concept to characterize the family of meta-models sharing the **Token-Holder** semantics. For example, it can be bound to the **PetriNet** meta-model of Listing 2, where  $\&H$  is bound to **Place**,  $\&P$  to **Transition**, and so on.

Nonetheless, the concept can be bound to other meta-models as well. For instance, Listing 6 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 line 21: conveyors act like places, machines as transitions, and parts as tokens. We explicitly pass concrete meta-model elements to the pattern matcher in the `bind` command. The binding process matches association ends with a more general cardinality (e.g. `inHolders` with cardinality “\*”) to association ends with more restricted cardinality (e.g. `inC` with cardinality “1..\*”). This is so as all instances of the meta-model fulfil the cardinalities of the concept. The contrary is not allowed, as in that case some instances of the meta-model would not fulfil the cardinalities of the concept.

```

1 Model ProdSys {           15     owner           : Conveyor[0..1];
2   Node Machine {         16   }
3     ref : String;         17   Edge PH(Machine.outC,Conveyor.inM);
4     type: String;         18   Edge HP(Conveyor.outM,Machine.inC);
5     inC : Conveyor[1..*]; 19   Edge iP(Part.owner,Conveyor.parts);
6     outC: Conveyor[1..*]; 20 }
7   }
8   Node Conveyor {        21 bind TokenHolder(ProdSys
9     outM : Machine[*];    21           ,ProdSys::Conveyor
10    inM  : Machine[*];    21           ,ProdSys::Machine
11    parts: Part[*];       21           ,ProdSys::Part
12   }                       21           ,ProdSys::Conveyor::parts
13   Node Part {            21           ,ProdSys::Machine::inC
14     creationTime: int;    21           ,ProdSys::Machine::outC)

```

**Listing 6.** Binding the `Token-Holder` concept.

We can define generic behavioural specifications for the concepts, applicable to instances of any meta-model that satisfies the concept’s requirements. Listing 7 shows an excerpt of the EOL simulator for the `Token-Holder` concept. The program first states that it needs concept *TokenHolder* (line 1), and that 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, but nothing else. This program is actually an abstraction of that of Listing 4, because this one does not require concrete types. The working scheme is the same, but the operations *enabled* and *fire* are added to the class `&P` gets bound to. The simulator can be used to execute any instance of the `ProdSys` 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   while (&P.allInstances()->exists(t | t.enabled() and t.fire())) {}
4 }
5 operation &P.enabled() : Boolean {
6   return self.&inHolders->forAll(h | h.&tokens.size(>0);
7 }
8 operation &P.fire() : Boolean {...}

```

Listing 7. Token-Holder behaviour expressed over concept TokenHolder.

## 5 Model Templates

*Concepts* express requirements of models and meta-models. By using such abstraction mechanism, behaviours and transformations can be expressed in a type independent way, becoming more reusable. In this section we show how *model templates* can be used to define models in a generic way.

Templates use concepts to express requirements on the type 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 actually 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 and generic model components using templates, which we can later instantiate and combine. Consider again the Producer-Consumer Petri net model presented in Section 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 8 shows how to specify the templates with METADEPTH. The first template `Buff2` (lines 7-13) defines a generic buffer with one input and one output transitions. These two transitions (`&Tri`, `&Tro`), together with their owning models (`&PNi`, `&PNo`), are passed as parameters to the template. Then, the template simply imports both received models (line 9), declares one place (line 10) and connect it to the received transitions (lines 11-12). In addition, the template requires in line 8 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.

The second template, `TwoStateProc` (lines 15-22), defines a two-state process. In this case, the template has no parameters and acts like a pattern which 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 basic data types, which is left for future work.

```

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

```

Listing 8. Defining and using model templates.

Lines 24–26 instantiate the templates. The resulting model `ProducerConsumer` is equivalent to the one in Listing 3. However, the use of templates has risen the abstraction level of the model, which is now more concise, and we have reused the definition of the template `TwoStateProc`. Altogether, model templates help in defining component and pattern libraries for domain specific languages.

## 6 Meta-model Templates and Semantic Mixin Layers

Templates are not only useful to define generic models, but can be applied to meta-models to provide an extensible way of defining languages, similar to mixin layers [12]. In our context, a mixin layer is a set of functionalities added to a meta-model by extending the elements passed as parameters in the template that implements the functionalities. Here we explore *semantic mixin layers*, which are *meta-model templates* declaring functionality 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 `Collections` or `Maps` 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 to the right of Fig. 3. It shows a mixin layer template  $T$  that is used to extend the meta-model members of a semantic family, characterized by concept  $C$ .  $L(T)$  contains the meta-models in  $L(C)$  once they have been extended with the execution infrastructure. In this way,  $T$  can be seen as a function with domain  $L(C)$  which adds such infrastructure. Then, we can define a simulator for the mixin layer  $T$ , which will be applicable to instances of any meta-model in  $L(T)$ .

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. The simulator would necessitate 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 simulation. Hence, a separate mixin layer incorporates these elements into the language definition in a non-intrusive way.

Fig. 3 shows to the left the template implementing the mixin layer. It declares the necessary infrastructure to simulate instances of meta-models that satisfy the concept `TTokHold`, so that the template definition requires such a concept. This concept is similar to the concept `TokenHolder` in Listing 5, but in addition transitions (variable `&P`) are required to define a distinguished `time` field storing the firing delay. The template defines a family of meta-models which extend any meta-model `&M` satisfying concept `TTokHold` with the machinery needed for their simulation. For instance, assume we add an attribute `delay` to the `Machine` class in Listing 6. Then, the meta-model `ProdSys` is a valid binding for the concept `TTokHold`, and hence we can instantiate the mixin layer for the meta-model in order to incorporate it the simulation infrastructure. This is done by declaring `TimedSched<ProdSys, ProdSys::Conveyor, ...> SimProdSys`.

In particular, the template in Fig. 3 extends the received meta-model `&M` with a class `Event` to store the events, and a singleton class `FEvtList` to handle the event list. Moreover, the class with role `&P` (transition) is added a collection `evts` storing the scheduled events associated to the transition, and the class with role `&T` (token) is extended with the event in which the token is engaged, if any.

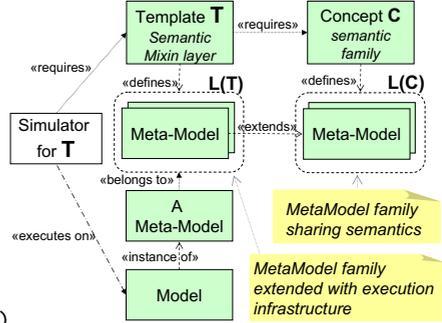
Behaviours associated to semantic mixin layers use the generic types of the template. Listing 9 shows an excerpt of the simulator associated to the `TimedSched` mixin layer. The simulator uses a `FEvtList` object (line 3) to keep the current simulation time and the list of scheduled events. The list of events is initialized with the set of active transitions (lines 5-6). The main simulation loop (lines 8-13) 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).

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 has been applied (i.e. any meta-model fulfilling the `TTokHold` concept), like `SimProdSys`, hence obtaining highly reusable specifications.

```

1 template <&M,&H,&P,&T,&tok,&inH,&outH,&time>
2   requires TTokHold(&M,&H,&P,&T,&tok,&inH,&outH,&time)
3 Model TimedSched extends &M {
4   Node &P { evts: Event[*]; }
5   Node &T { evts: Event[0..1]; }
6   Node FEvtList[1] {
7     first: Event[0..1];
8     time : double;
9   }
10  Node Event {
11    time: double;
12    next: Event[0..1];
13    proc: &P;
14    toks: &T[*];
15  }
16  Edge ProcTm(&P.evts, Event.proc)
17  { t:double; }
18  Edge TokTm(&T.evts, Event.toks)
19  { t:double; }
20 }

```



**Fig. 3.** Semantic mixin layer adding infrastructure to simulate TTokHold concepts (left). Working scheme of semantic mixin layers (right).

```

1 @template(name=TimedSched)
2 operation main() {
3   var FEL := new FEvtList;
4   FEL.time := 0;
5   var enab: Set(&P):= getEnabled();
6   FEL.schedule(enab);
7   var finish: Boolean := false;
8   while (not finish) {
9     FEL.time:= FEL.first.time;
10    var t: &P := FEL.first.proc;
11    t.fire();
12    ...
13  }
14 }

```

**Listing 9.** Excerpt of the simulator for the TimedSched mixin layer.

## 7 Related Work

The use of templates in modelling is not new, as they are already present in the UML 2.0 specification [10], as well as in previous approaches like Catalysis' model frameworks [4] and package templates, and the aspect-oriented meta-modelling approach of [2]. Interestingly, while all of them consider templates for meta-models (or class diagrams), none consider concepts or model templates.

Catalysis' model frameworks [4] are parameterized packages that can be instantiated by name substitution. Hence, they are similar to our meta-model templates. The package templates of [2] are based on those of Catalysis, and are used to define languages in a modular way. They are based on string substitution, as the parameters of the templates are strings that are substituted in the template definition. This approach is realized in the XMF tool [15]. Although

package templates were incorporated into the UML 2.0 specification, the MOF does not consider genericity at the meta-model or model level.

Kermeta ([kermeta.org](http://kermeta.org)) includes facilities for model typing [8], allowing establishing a subtyping relationship between meta-models. Hence, generic behaviours can be defined in a generic meta-model and applied to any subtype meta-model. This approach has been applied to generic refactorings [8].

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 5 (cf. Listing 8). Moreover, as our framework supports an arbitrary number of meta-models through potency [3], we can 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 behaviours, this is missing in other works [2,4,10].

Generic (meta-)modelling has fundamental differences with generic programming. The first 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, modelling concepts check structural properties of models.

Another set of related research are the (meta-)model modularization approaches, like *Reuseware* [6]. In that 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. Moreover, [6] does not consider behaviours and lacks abstraction mechanisms like concepts.

## 8 Conclusions and Future Work

In this paper we have shown the benefits of bringing concepts, templates and mixin layers into language engineering. Concepts allow expressing requirements of template parameters, and by themselves permit defining behaviours 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. We have seen that mixin layers, a kind of template meta-models, are especially useful to provide the necessary infrastructure to execute models. These elements have been realized in the METADEPTH tool (the examples and the tool are at <http://astreo.ii.uam.es/~jlara/metaDepth/Genericity.html>). However, the discussions are general and applicable to other contexts as well.

We are currently exploring the potential opened by genericity. We believe the semantics of modelling languages can be classified using concepts. Hence,

we will define concepts for transition-based semantics, communication semantics, discrete-event semantics, and so on. The combination of concepts and semantic mixin layers will provide support for the rapid prototyping of language semantics. We are also exploring the construction of pattern libraries for domain-specific languages through model templates. We are working in improving the METADEPTH support for genericity, adding extension relations between concepts, to allow the incremental construction of concepts and concept libraries. Genericity is also applicable to the definition of generic model-to-model transformations. Finally, we are also working on an algebraic formalization, and a proof of type safety of specifications using concepts.

**Acknowledgements.** Work sponsored by the Spanish Ministry of Science, project TIN2008-02081 and mobility grants JC2009-00015 and PR2009-0019, and by the R&D programme of the Community of Madrid, project S2009/TIC-1650.

## References

1. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.* 12(4), 290–321 (2002)
2. Clark, T., Evans, A., Kent, S.: Aspect-oriented metamodeling. *The Computer Journal* 46, 566–577 (2003)
3. de Lara, J., Guerra, E.: Deep meta-modelling with METADEPTH. In: Vitek, J. (ed.) *TOOLS 2010. LNCS*, vol. 6141, pp. 1–20. Springer, Heidelberg (2010)
4. D'Souza, D.F., Wills, A.C.: Objects, components, and frameworks with UML: the catalysis approach. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (1999)
5. García, R., Jarvi, J., Lumsdaine, A., Siek, J.G., Willcock, J.: A comparative study of language support for generic programming. *SIGPLAN Not.* 38(11), 115–134 (2003)
6. Heidenreich, F., Henriksson, J., Johannes, J., Zschaler, S.: On language-independent model modularisation. *T. Asp.-Oriented Soft. Dev.* VI 6, 39–82 (2009)
7. Kolovos, D.S., Paige, R.F., Polack, F.: The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (eds.) *ECMDA-FA 2006. LNCS*, vol. 4066, pp. 128–142. Springer, Heidelberg (2006)
8. Moha, N., Mahé, V., Barais, O., Jézéquel, J.-M.: Generic model refactorings. In: Schürr, A., Selic, B. (eds.) *MODELS 2009. LNCS*, vol. 5795, pp. 628–643. Springer, Heidelberg (2009)
9. Musser, D.R., Schupp, S., Loos, R.: Requirement oriented programming. In: Jazayeri, M., Musser, D.R., Loos, R.G.K. (eds.) *Dagstuhl Seminar 1998. LNCS*, vol. 1766, pp. 12–24. Springer, Heidelberg (1998)
10. OMG: UML 2.2 specification, <http://www.omg.org/spec/UML/2.2/>
11. OMG: MOF 2.0 (2009), <http://www.omg.org/spec/MOF/2.0/>
12. Smaragdakis, Y., Batory, D.: Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.* 11(2), 215–255 (2002)
13. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: *EMF: Eclipse Modeling Framework*, 2nd edn. Addison-Wesley Professional, Reading (2008)
14. Stepanov, A., McJones, P.: *Elements of Programming*. Addison Wesley, Reading (2009)
15. Tony Clark, J.W., Sammut, P.: *Applied Metamodeling, a Foundation for Language Driven Development*, 2nd edn., Ceteva (2008)