# Abstracting Modelling Languages: A Reutilization Approach

Juan de Lara[1], Esther Guerra[1], and Jesús Sánchez Cuadrado[1]

Universidad Autónoma de Madrid (Spain)
{Juan.deLara, Esther.Guerra, Jesus.Sanchez.Cuadrado}@uam.es

**Abstract.** Model-Driven Engineering automates the development of information systems. This approach is based on the use of Domain-Specific Modelling Languages (DSMLs) for the description of the relevant aspects of the systems to be built. The increasing complexity of the target systems has raised the need for abstraction techniques able to produce simpler versions of the models, but retaining certain properties of interest. However, developing such abstractions for each DSML from scratch is a time and resource consuming activity.
Our solution to this situation is a number of techniques to build reusable abstractions that are defined once and can be reused over families of modelling languages sharing certain requirements. As a proof of concept, we present a catalogue of reusable abstractions, together with an implementation in the MetaDepth multi-level meta-modelling tool.

**Keywords:** Model-Driven Engineering, Domain-Specific Modelling Languages, Abstraction, Genericity

## 1 Introduction

In Model-Driven Engineering (MDE), models are actively used to conduct the different phases of the project, being employed to specify, simulate, optimize, test and produce code for the final systems. Models are sometimes specified using general purpose modelling languages, like the UML, but in order to unfold the full potential of MDE, Domain-Specific Modelling Languages (DSMLs) are frequently used, specially tailored to specific areas of concern.

As the application domain becomes complex, models tend to become complex as well. This fact hinders the development, understanding and analysis of models. To tackle this issue, researchers have developed abstraction techniques to reduce model complexity for particular notations and purposes [1–4]. In our setting, an abstraction is a model operation that produces a simpler model that retains certain properties of interest from the original model.

Frequently, different notations can be abstracted following similar patterns. For example, a common abstraction consists on aggregating a linear sequence of connected elements, which can be applied to a class diagram to flatten an inheritance hierarchy, or to a process model to aggregate a sequence of activities. Additionally, some abstractions are specific to particular domains. For example,

the abstractions for Petri nets [1] produce nets easier to analyse, but which preserve liveness, safeness and boundedness. Similarly, there are abstractions specific to process modelling [2, 3] which produce more comprehensible models or facilitate their verification [4]. These abstractions are applicable (in theory) to a family of notations sharing semantics, like BPMN and Activity Diagrams.

In MDE, the abstract syntax of DSMLs is defined through a meta-model and the needed abstraction operations are defined over its elements. This enables the application of the abstractions to the instances of a specific meta-model, but not to the instances of other meta-models even if they share characteristics with the former one. Therefore, even though one can develop abstractions for the meta-model of a particular DSML, like the OMG's BPMN 2.0, these cannot be used for other related DSMLs, like YAWL [4], Activity Diagrams or other BPMN variants. A catalog of abstractions and mechanisms for their reuse accross meta-models would save significant effort in defining advanced DSML environments.

In previous works [5], we developed a technique for the reutilization of model management operations. The basic idea is to introduce an extra level of indirection, so that operations are not defined over concrete meta-models, but over so-called *concepts* which gather the requirements that a meta-model should fulfil so that the operation becomes applicable. In our approach *concepts* have the form of meta-models as well, and operations defined over them become reusable, as concepts can be bound to families of meta-models satisfying their requirements.

In this work we present reusable abstraction techniques for modelling languages in the context of MDE. For this purpose, we first introduce a classification of abstractions for modelling languages, and then present a catalogue of generic, reusable abstractions, applicable to sets of meta-models. We consider four abstraction types: aggregation, merge, deletion and view generation. Orthogonally, each abstraction can be either *horizontal* if it is applicable to meta-models of different domains, or *domain-specific* if it is applicable to families of meta-models sharing semantics (e.g. languages similar to Petri nets or workflow languages). Our abstractions are: (a) reusable, as they can be applied to several modelling languages; (b) customizable, as some aspects can be configured, like similarity criteria or attribute aggregation operations; and (c) adaptable, as they provide extension mechanisms for meta-models when a language lacks support for encapsulation or aggregation. We have validated these ideas by an implementation in the MetaDepth tool [5], and several case studies (BPMN, Statecharts, DSMLs).
**Paper organization**. Section 2 introduces a categorization of abstractions for modelling languages. Section 3 presents some techniques to define generic abstractions, which we use to define a catalogue of reusable abstractions in Section 4. Section 5 shows an implementation using MetaDepth. Section 6 compares with related research and Section 7 concludes.

## 2 Classifying Abstractions for Modelling Languages

In this section we present a categorization of abstractions for modelling languages. This classification has been built after a thorough analysis of the ab-

stractions provided by or developed for languages of extended use like BPMN [3], as well as from our own experience on the construction of DSMLs.

In our setting, a model abstraction is an operation that reduces the complexity of some aspect of a model. In this way, the purpose of an abstraction can be to increase the comprehensibility of a large model, or to reduce the size of a model to ease its verification while retaining certain properties of interest, among others. An abstraction may imply the deletion of existing model elements, as well as the addition of new ones – like aggregate objects or hierarchical constructs – that encapsulate existing elements which share certain features. The addition of new elements may influence the way in which a model is visualized, e.g. enabling to zoom-into or to hide the aggregated objects, but the focus of this work is not on model visualization, which we touch only briefly in Section 5.

We distinguish two types of abstractions according to their applicability:

- *Horizontal abstractions*, applicable to modelling languages of (perhaps very) different domains. For example, an abstraction that encapsulates a sequence of model elements can be used to flatten a linear inheritance hierarchy in a class diagram, or to simplify a business process model by creating a subprocess that groups a sequence of consecutive activities.
- *Domain-specific abstractions*, specific to a family of DSMLs sharing semantics. Examples of this kind of abstractions include the reduction techniques for Petri nets [1] and the ones for workflows [4]. Being domain-specific, they can take into account the semantics of the languages in the domain and ensure the preservation of properties, which permits their use for verification purposes. For example, the reduction techniques in [1] result in simpler models that preserve liveness, safeness and boundedness.

Orthogonally, we identify four abstraction types according to their behaviour:

- *Merge*. In this kind of abstraction, a set of model elements considered similar is replaced by one element of the same type which collects the properties of the merged elements. A frequent use of this abstraction is for verification. E.g. the reduction rules for Petri nets [1] merge groups of places or transitions into a unique place or transition collecting the arcs of the merged elements. In some cases, the element replacing the group is assigned property values that result from a calculation using the properties of the merged elements.
- *Aggregation*. In this case, a set of similar model elements is grouped hierarchically under a higher-level element, normally of a different type, which serves as an aggregate. An example is the encapsulation of a linear sequence of activities in a process model into a subprocess, obtaining a more comprehensible, hierarchical model. Properties of the aggregate object may be calculated using properties of the aggregated objects.
- *Deletion*. This kind of abstraction deletes elements that are not considered relevant or that do not modify some observed property of a model. An example is the elimination of self-loops in Petri nets [1] and workflow nets [4].
- *Views*. In this case, the abstraction produces a new model (called *view*) which may be expressed using the same language as the original model or a

different one. The view discards those features of the original model which are irrelevant for the aim of the abstraction. This is the most general type of abstraction as it includes the previous ones, whenever the languages of the original model and the view are the same. There are many examples of this type of abstraction for verification, where the view uses a language with a rich body of theoretical results enabling the analysis of the source model. Transforming BPMN models into Petri nets is an example of this abstraction

Once we have seen the different model abstraction types, next section discusses a possible approach to make abstractions reusable for several DSMLs.

## 3 Making Abstractions Generic

MDE is a type-centric approach because model manipulations use the types of a concrete meta-model, becoming hard to reuse for other meta-models. In [5] we introduced some mechanisms to define reusable model management operations, which we review and apply to define generic model abstractions in the following.

Assume we need an operation to simplify a business process model by creating an aggregate object abstracting the flow elements between two complementary gateways (e.g. a parallel fork and join). Currently, there is a plethora of different business process modelling notations like Activity diagrams, YAWL, the OMG's BPMN 2.0, and different variations of BPMN, like the Intalio's BPM modeller[1]. Hence, one needs to select a particular meta-model and implement the operation for it, so that the operation is not applicable for the other meta-models anymore.

To overcome this limitation, we propose building a *generic* model abstraction operation that can be applied to a family of meta-models. For this purpose, we do not build the operation over the specific types of a meta-model, but over the variable types defined in a so-called *structural concept* which gathers the requirements that a meta-model should fulfil to qualify for the operation. Concepts have the form of a meta-model, but their elements (classes, attributes and references) are interpreted as variables. A generic operation is used by *binding* its associated concept to a meta-model, as shown in Fig. 1. The binding defines a 1-to-1 mapping from each class, attribute and reference in the concept to a class, attribute and reference in the meta-model, respectively. It needs to follow some well-formedness rules (see [5]) to ensure a correct application of the generic operation. For example, if a class $c$ in the concept is bound to a class $c'$ in the meta-model, the attributes and references in $c$ must be bound to features defined in $c'$ or in some superclass of $c'$. We can map two classes $c$ and $d$ in the concept to a unique class in the meta-model provided it contains all features demanded by $c$ and $d$. Once defined, the binding induces a re-typing of the generic operation, which becomes applicable to the instances of the bound meta-models, obtaining reusability (the *same* operation becomes applicable to *several* meta-models).

As an example, Fig. 1 defines the generic abstraction `abstractBlock` using the types of the concept *Workflow*. The operation creates an aggregate object abstracting the flow elements between two complementary gateways. The concept

[1] `http://www.intalio.com/bpms/designer`

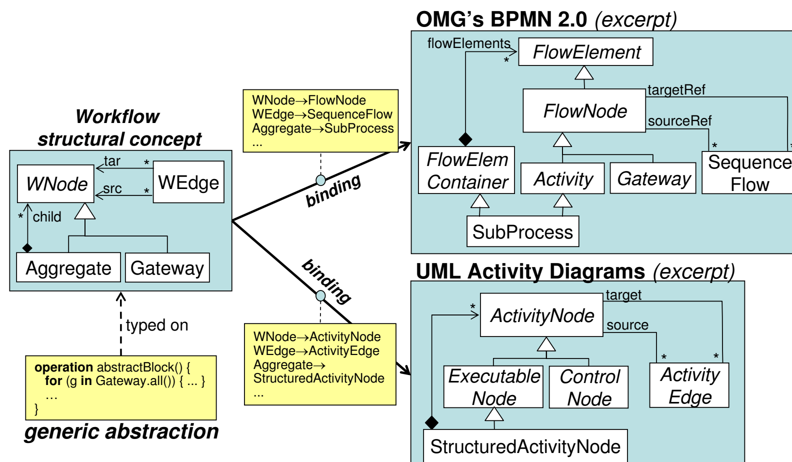


**Fig. 1.** Generic model abstraction operation through a structural concept.

gathers the variable types used by the operation, like `Gateway` and `Aggregate`, but note that it does not include a class representing a task (even though workflow languages usually include such a class) because the operation does not need to distinguish any flow node other than gateways. In this way, the concept and its bindings are kept as simple as possible. Then, we can bind the concept to different meta-models, like those of the OMG's BPMN 2.0 and the UML 2.0 Activity Diagrams, enabling the application of the abstraction operation to the model instances of these notations. The figure shows an excerpt of the binding for both meta-models. For BPMN, `WNode` is bound to `FlowNode` and `WEdge` to `SequenceFlow`. The bindings permit certain heterogeneity in the subtyping, as `Aggregate` is a direct subtype of `WNode` in the concept, but `SubProcess` is an indirect subtype of `FlowNode` in the BPMN meta-model.

### 3.1 Configuration and adaptation to the modelling language

A *structural concept* has the form of a meta-model and reflects a design decision that some meta-models could implement differently. For example, the Intalio's BPMN meta-model represents all kinds of flow nodes through a unique class `Activity` with an attribute to discriminate the type. As a consequence, we cannot bind the previous structural concept *Workflow* to this meta-model. Our solution to widen the range of boundable meta-models for a generic operation is to use so-called *hybrid concepts* [5], which abstract away the accidental details introduced by the specific choice of structure in the concept behind a suitable interface. Thus, *hybrid concepts* are like structural ones but require a number of operations from the classes they define. The binding is then obtained by mapping the elements and implementing the operations declared in the concept.

Fig. 2 shows the definition of the hybrid concept *Workflow-2*, which is a more flexible version of the structural concept presented in Fig. 1, as it imposes less

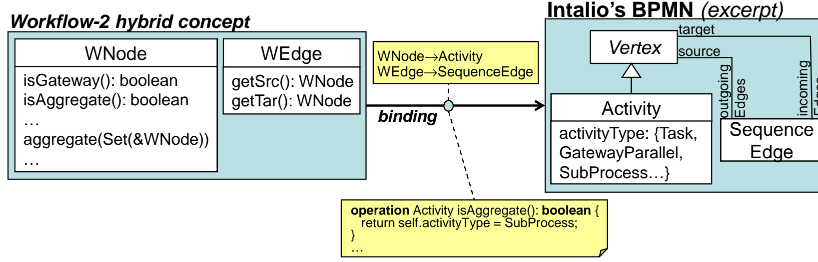


**Fig. 2.** Binding to Intalio's BPMN meta-model with a hybrid concept.

structural requirements. This concept can be bound to the Intalio's BPMN meta-model, which requires implementing the operations required by the concept. For instance, the figure shows the implementation of operation `isAggregate`, which in this case returns true whenever the attribute `activityType` of an activity takes the value `SubProcess`.

Finally, sometimes an abstraction needs to be configured with similarity criteria or with a function over attribute values. These configuration points can be expressed through *hook* operations in a hybrid concept as well. Hook operations provide a default implementation, and only need to be implemented during the binding if a special behaviour is required. For instance, in Fig. 2, operation `aggregate` in class `WNode` is a hook to customize an aggregation operation on attributes (e.g. adding up a time attribute in all aggregated nodes).

### 3.2 Extending the modelling language

Some abstraction operations may create an aggregate object grouping similar model elements. However, some notations were not designed with hierarchical or encapsulation elements, and therefore we cannot apply these abstractions to them. To overcome this limitation, our solution is to define a so-called mixin layer, which is a parameterized meta-model that contains those auxiliary elements needed by an operation [5]. Then, we use concepts to express the requirements that a meta-model should fulfil to become extendible by the mixin.

As an example, the mixin in Fig. 3 adds an aggregate class to any meta-model boundable to the *Workflow-3* concept, which in this case does not demand the existence of such a class. The generic operation is defined over the gluing of the mixin and the concept (label 2 in the figure) through the *extension points* or parameters of the mixin (label 1). In the figure, class `WNode` is the only parameter of the mixin. The rest of elements of the mixin will be added to any meta-model to which we can bind the concept. In this way, we can bind the concept to meta-models like Petri nets, or to DSMLs to represent plant factories like the one shown in the figure (label 3). Applying the mixin to the DSML (label 4) adds the aggregate class to its meta-model, and hence the abstraction becomes applicable to their instances. Moreover, this kind of mixin preserves the compatibility of models conformant to the old meta-model with the extended one.

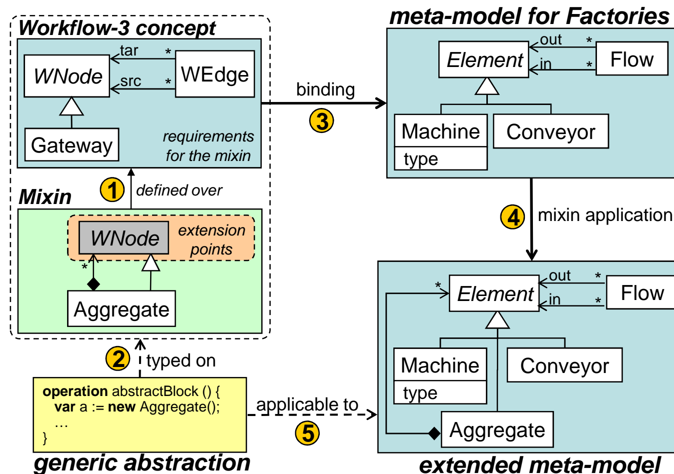


**Fig. 3.** Generic model abstraction operations through a mixin.

Now that we have seen the different ways to make abstractions generic, customizable and adaptable, we provide a catalogue of reusable abstractions, defined using the presented techniques.

## 4 A Catalogue of Generic Abstractions

One of the aims of the present work is to make easier the incorporation of abstraction capabilities and operations to existing or new modelling languages, so that these operations do not need to be developed from scratch. For this purpose, based on the techniques presented in the previous section, we have built a catalogue of generic abstractions that a designer can select and customise for his particular modelling language. Our abstractions are generalizations of the domain-specific ones we have found in the literature, and can easily be customised for a particular modelling language by identifying the element types participating in the abstraction. Additionally, some of them permit configuring additional details through the use of hook methods (e.g. the similarity criteria used to discriminate which elements of a model should be abstracted), or present variations in their semantics (e.g. whether a set of elements should be either encapsulated in an aggregate object or merged).

Technically, our generic abstractions are operations defined over suitable concepts to be bound to the meta-models of specific modelling languages. To increase the reuse opportunities of each abstraction type, we provide different binding possibilities for them: (i) from a structural concept, which is the simplest approach when the structures of the concept and the meta-model are similar, (ii) from a hybrid concept, which gives freedom regarding the particular structure of the bound meta-model, and (iii) from a concept associated to a mixin in case the

bound meta-model has no support for abstractions, e.g. it lacks a class to represent object aggregations, in which case the mixin extends the meta-model with such a class, enabling the abstraction application. Internally, for each abstraction type we have defined a binding from its hybrid to its structural concept, which permits encoding the abstraction operation just once over the hybrid concept and reusing it for the structural one. In any case, this is transparent to the reuser of the abstraction. Finally, we provide two implementations for each abstraction operation: (i) one performing the abstraction on the input model in-place, and (ii) another one generating a view (i.e. a different model) with the result.

In the remaining of this section we present our catalogue of abstractions, classified depending on their applicability: horizontal (i.e. general) and domain-specific. We do not claim that this catalogue is complete, as we expect to add new abstractions in the future. Nonetheless, we will show some examples illustrating that our current catalogue can be used to customize meaningful abstractions for well-known notations.

### 4.1 Horizontal abstractions

**Target parallel.** It abstracts a maximal set of objects that refer to the same target objects. There are two variants of this abstraction: *aggregate* and *merge*. The *aggregate* variant creates an aggregate that encapsulates all objects with same target, whereas the *merge* variant replaces the parallel objects by another one of the same type. In both cases, all original references are substituted by non-duplicate references from the created abstraction.
To use this abstraction, a binding has to be provided from the associated concept (either structural, hybrid or mixin-based if the target meta-model does not define an aggregate class) to a meta-model. Fig. 4 shows to the left the structural concept for the *aggregate* variant of this abstraction. Hence, the abstraction can be reused for a particular notation by specifying the type of the objects to abstract (class `Item`), the referenced target (class `Context` and reference `target`), and the aggregation elements (`Aggregate` and `child`). The concept includes a hook method `canAggregate` to configure extra conditions that the abstracted elements need to fulfill. The right of the same figure shows the working scheme of this abstraction, where three objects are aggregated as all refer to the same target objects (which in this case is only one) and their references to the target object are substituted by a unique reference from the created aggregate. Later, a particular tool may decide whether showing or not the objects inside the aggregate.
Fig. 5 shows an application of this abstraction over UML 2.0 Statecharts, to encapsulate sets of states that reach a same state through a same trigger (i.e. it performs an unflattening of statecharts). The left of the figure shows an excerpt of the UML Statecharts meta-model (simplified as it does not include *Regions*). We have depicted the binding through annotations. Hence, `Item` and `Context` in the concept are both bound to `Vertex` in the meta-model as we want to detect states connected to the same states. We have parameterized the abstraction by overriding operation `canAggregate`

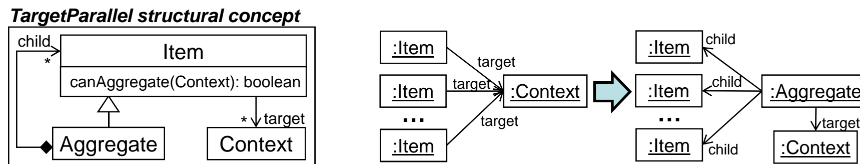


**Fig. 4.** Target parallel aggregation abstraction: concept (left) and behaviour (right).

to include only target states with the same `Trigger`. In fact, we have used the hybrid version of the concept for this abstraction, as states are not connected to states through a reference (as demanded by the structural concept) but through an intermediate class `Transition`. The right of Fig. 5 shows the in-place application of this unflattening to a Statechart, which abstracts all states with a transition triggered by "e" to *S4*.

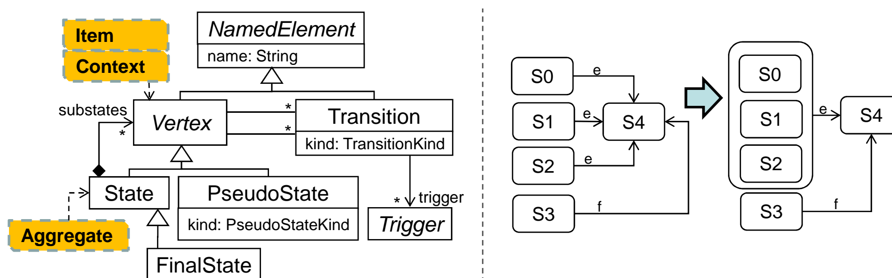


**Fig. 5.** Target parallel aggregation abstraction: binding to Statecharts (left) and application (right).

**Source parallel.** It abstracts a maximal set of objects which is referenced from the same source objects. Thus, this abstraction is like the previous one, but considers input parallel objects instead of target ones. The concept to bind in this case is therefore similar to the target parallel one, but there is a reference `source` from class `Context` to `Item` instead of the reference `target`.

**Parallel.** It abstracts a maximal set of objects with the same source and target elements. The concept for this abstraction is shown to the left of Fig. 6 in its variant *merge* (i.e. the abstracted items are substituted by another item). There are also structural and hybrid versions of this concept for the *aggregate* variant, which include a class `Aggregate`. The middle of the figure shows an application of this abstraction to the meta-model of (non-deterministic) automata. The aim is simplifying automata by merging sets of equivalent states, which are those that can reach, and can be reached from, the same states through transitions with the same symbols. In such a case, we can merge the equivalent states and obtain a simpler automaton which preserves behaviour, as the figure illustrates to the right. For this purpose, we bind all

classes in the concept to class `State`, and override the `canAggregateS` and `canAggregateT` hook methods to check that transitions to/from the same states have the same symbol. As in the previous example, we have used the hybrid version of the concept because states are inter-connected through the intermediate class `Transition`, not directly through a reference.

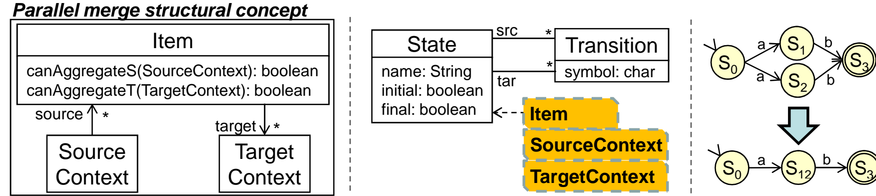


**Fig. 6.** Parallel merge abstraction: concept (left), binding to non-deterministic automata (middle), and application (right).

**Sequential.** It abstracts a maximal sequence of linearly connected objects, and admits both variants *merge* and *aggregate*. The left of Fig. 7 shows the structural concept for the sequential *aggregate* variant, and the center of the figure illustrates how the abstraction works: it creates an aggregate for a sequence of items, and copies the connections of the first and last item of the sequence to the aggregate. The figure shows to the right an application to BPMN 2.0, where we have mapped `Item` to `FlowNode` and `Aggregate` to `SubProcess` (see Fig. 1). Moreover, we have customized the `canAggregate` hook method to forbid having gateways as first or last element in the abstracted sequence.

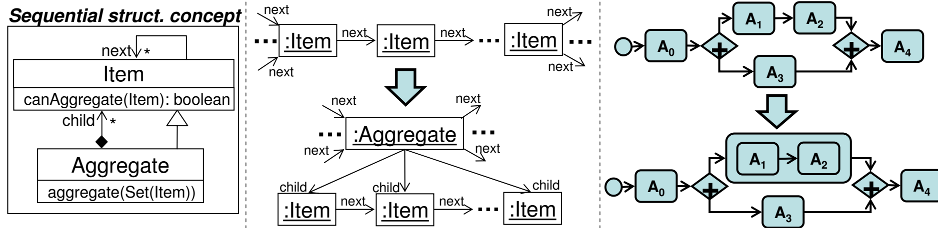


**Fig. 7.** Sequential aggregation abstraction: concept (left), behaviour (middle) and application to BPMN (right).

**Similarity.** It abstracts a maximal set of objects considered similar by a custom-defined operation. The left of Fig. 8 shows the structural concept for the *aggregate* variant of this abstraction. The right of the figure shows an example application to BPMN 2.0 models. As in the abstraction heuristic presented in [3], tasks are considered similar if they produce or consume the same

sets of `Data Objects` and use the same resources (in the model example, they are performed by the same `HumanPerformer`). Thus, we configure this abstraction by binding class `Item` in the concept to `FlowNode`, `Aggregate` to `SubProcess`, and implementing the `isSimilarTo` hook method.

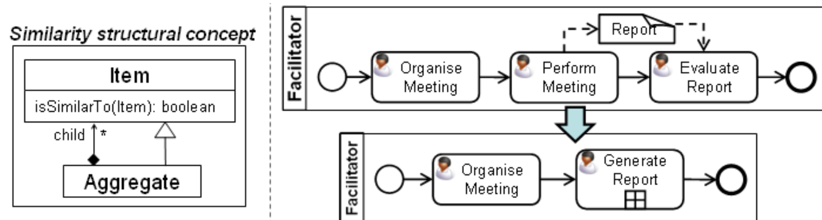


**Fig. 8.** Similarity aggregation abstraction: concept (left), application to BPMN (right).

**Loop removal.** It removes loops between two types of objects. The associated concept is shown in Fig. 9(a). We consider two variants: the first removes one of the references (`source` or `target`) to break the loop, whereas the other one removes the object with type `Item` and its references. The second variant is illustrated in Fig. 9(b). Fig. 9(c) shows an application of this abstraction that simplifies Petri nets, while preserving their behaviour, through the *elimination of self-loop places* [1]. Thus, a place is removed if it has at least one token and has no further input or output arcs. Fig. 9(d) shows an application to YAWL nets [4] that eliminates self-loop tasks. A different binding into YAWL would enable the elimination of self-loop conditions [4].

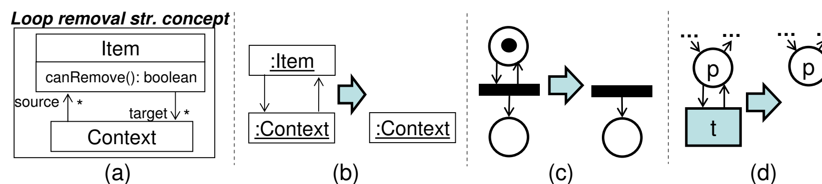


**Fig. 9.** Loop removal abstraction: concept (a), behaviour (b), application to Petri nets (c) and application to YAWL nets (d).

### 4.2 Domain-specific abstractions

Oftentimes, abstractions are specifically designed for a particular domain, such as Petri nets [1] or process models [3, 4]. Defining them over concepts makes them meta-model independent, and therefore reusable for other languages with similar semantics. For example, Fig. 1 shows the concept *Workflow* for process

modelling languages, which expresses the requirements for the *block* abstraction operation. This abstraction aggregates into a subprocess a block delimited by two complementary gateways. An example application of this abstraction to BPMN is shown in Fig. 10.

If an abstraction for a particular domain is complex, we can certainly build it by reusing and customizing horizontal abstractions (like the ones in our catalogue). However, then it is sensible to specialize it as a domain-specific abstraction over a concept that reflects the particularities of the domain and can be bound to languages with close semantics without much configuration effort. For instance, once we have built the similarity abstraction that aggregates activities sharing data and resources (cf. Fig. 8), we can specialize it for their reuse for any workflow language. For this purpose, we would redefine the abstraction operation over a concept like the one in Fig. 1 but extended with tasks, resources and data objects. This concept (and abstraction) becomes easily reusable for families of process languages, like BPMN and Activity Diagrams.

## 5 Tool Support

The approach and abstractions presented in this paper have been implemented and validated using MetaDepth, a meta-modelling framework which supports multi-level meta-modelling and textual modelling [5]. It is integrated with the Epsilon family of languages[2], which permits defining in-place model manipulations, model-to-model transformations and code generators. All these operations can be made generic by their definition over (structural or hybrid) concepts or mixins. Moreover, we have extended MetaDepth's genericity support with the possibility of defining hook methods with default implementations in concepts.

Concepts, mixins and bindings are specified textually in MetaDepth. As an example, Listing 1 shows the definition of the structural concept shown in Fig. 1. The definition of a concept is similar to a meta-model definition, but its elements are considered variables and their definition is preceded by "&". The concept has a list of parameters (lines 2–3) in order to ease the binding to meta-models, as we will see later.

```
concept Workflow
  (&M, &WNode, &WEdge, &Aggregate,
   &Gateway, &child, &src, &tar) {
   Model &M {
      abstract Node &WNode {}
      Node &WEdge {
         &src : &WNode;
         &tar : &WNode;
      }
      Node &Aggregate : &WNode {
         &child : &WNode[*];
      }
      Node &Gateway : &WNode {}
   }
}
```

**Listing 1.** Structural concept.

```
operation blockAbstraction() : Boolean {
  var comp : &WNode = null;
  for (gw in &WNode.allInstances())
    if (gw.isSplit()) {
      comp := getJoin(gw);
      if (comp<>null) {
        var sq : Sequence(&WNode);
        sq.addAll(getAllInBetween(gw, comp));
        createAggregateFor(sq, gw, comp);
        return true;
      }
    }
  return false;
}
...
```

**Listing 2.** Block abstraction operation.

[2] http://eclipse.org/gmt/epsilon/

Once the concept is defined, we can build operations that use the variable types defined in the concept. In MetaDepth, the abstraction operations that modify the models in-place are defined using the Epsilon Object Language (EOL), whereas the *view*-generating abstraction operations are defined with the Epsilon Transformation Language (ETL). Listing 2 shows an excerpt of the *block* abstraction operation using EOL (auxiliary operations and methods are not shown), which uses the types of the `Workflow` concept of Listing 1.

Finally, in order to apply the generic operation, a binding from the concept to a specific meta-model needs to be specified. Listing 3 shows the binding of the concept in Listing 1 to the meta-model of BPMN2.0. In this case, the `SubProcess` type of BPMN acts as aggregate (fourth parameter of the binding, corresponding to the `&Aggregate` variable of the concept).

```
bind Workflow ( BPMN, BPMN::FlowNode, BPMN::SequenceFlow,
   BPMN::SubProcess, BPMN::Gateway, BPMN::FlowElementsContainer::flowElements,
   BPMN::SequenceFlow::sourceRef, BPMN::SequenceFlow::targetRef )
```

**Listing 3.** Binding to meta-model.

The user of a generic abstraction only has to define the binding from the concept to his modelling language. The binding in Listing 3 allows for the reuse of the *block* abstraction operation, which consists of 200 lines of EOL code. Thus, the user has to specify fewer lines of code (3) to benefit from a proven abstraction operation. Actually, the real definition of our *Workflow* concept is slightly larger and has associated further domain-specific abstractions, so that the reuse opportunities are larger. Moreover, the user of the abstraction is not confronted with understanding the code being reused and a subsequent manual adaptation of the operation for a particular meta-model, which is error-prone. Instead, he only deals with the operation "interface" (the concept), and so the different elements in the concept become similar to roles in design patterns.

So far we have considered abstractions at the abstract syntax level. However, modelling languages have a concrete syntax, typically a graphical one. When an abstraction is applied, the visualization has to be updated accordingly. As a proof of concept, we have built a visualization engine for MetaDepth models with support for grouping together elements into aggregate objects. We have created a meta-model to define graphical concrete syntaxes which includes the notion of grouping. Hence, in order to define the visualization of a language, a simple mapping from its meta-model to our graphical syntax meta-model must be given. For those elements added by a mixin to a meta-model (and which therefore are not defined in the meta-model) we use a generic visualization. Then, our engine interprets concrete syntax models and renders their visualization with the jGraph library. Fig. 10 shows the application of the *block* abstraction to a BPMN model.

## 6 Related work

Abstraction has been recognized as one of the key techniques for the model-based engineering of software and systems in the field of multi-paradigm modelling [6].

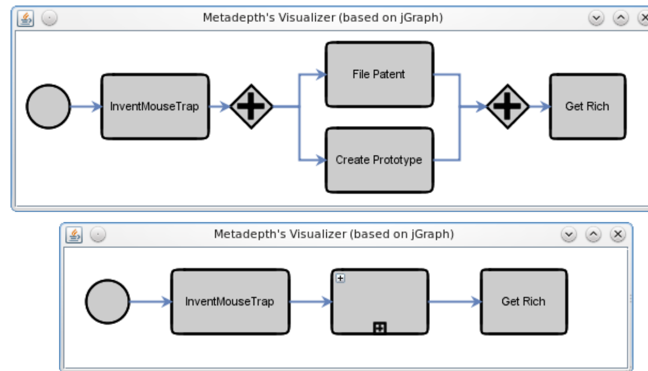


**Fig. 10.** Example of a BPMN model, before and after applying an abstraction.

However, to our knowledge, there are no works aimed at systematizing and providing a framework to reuse abstractions for DSMLs in MDE.

Two kinds of models are distinguished in [7]: type and token models. The latter capture singular elements of a system (e.g. a road map is a token model), and the former capture universal aspects of it through classification (e.g. a meta-model is a type model). While a type model can be seen as an abstraction of the set of its valid instance models, there is a lack of proposals – like the one we give here – for systematic abstraction of token models.

Model slicing has been proposed as a model comprehension technique inspired by program slicing. It involves extracting a subset of a model, called a slice, which retains some properties of interest. Slicers are typically bound to concrete meta-models, for instance UML [10]. This technique can be seen as a particular case of our view abstraction, when the obtained view conforms to the original meta-model. In [11], a language to define and generate model slicers is proposed, but the obtained slicers are not reusable for different meta-models in MDE.

There are also works that aim at the description of generic model refactorings [12]. Although they do not explicitly deal with abstractions, they could be used to abstract models. However, they lack constructs like mixins or hybrid concepts which we use to broaden the applicability of abstractions. Similarly, [13] describes a comprehensive set of *change patterns* for process models for the purpose of comparing the change frameworks provided by process-support technologies. Some of the described patterns (e.g. Extract Sub Process) can be interpreted as abstractions. Our work is especially directed to abstractions, and is not tied to process modelling languages, being generic. While the goal of [13] is to provide a systematic comparison framework, our goal is to offer automatic support for abstracting DSMLs in MDE. This is achieved through a catalogue of abstractions, defined using concepts, which are reusable by means of bindings.

Most abstraction techniques are specifically designed for particular notations. For example, there are techniques tailored for abstracting process models in BPMN [3]. Abstractions are also heavily used in verification [1] to obtain simpler

models. However, their implementation is frequently tied to the specificities of a language, and is hardly reusable even for similar notations.

Abstraction has also been studied theoretically. Hobbs [14] suggests that, in the course of reasoning, we conceptualize the world at different levels of granularity. He defines the notion of indistinguishability that allows us to map a complex theory of the world to a simpler theory for a particular context. This work has been used as a foundation to build theories about abstraction. For example, in [15], an abstraction is defined as a surjective, computable function between two first-order languages that preserves semantics. *Granularity* abstractions are defined as those non-injective mappings collapsing several symbols into a unique, abstracted one. Abstraction has also been used in many areas of Artificial Intelligence and Logic [16], e.g. to ease automated deduction. Keet [17] uses abstraction to help the comprehension of ontologies. In [18], granularity abstraction is applied to natural language processing. Natural language is represented as logical forms that are mapped to coarse-grained forms to enable sentence analysis. Kascheck [19] develops a theory of abstraction for information systems introducing *cohesion* predicates (m-ary relations) and abstractions of these (consistent n-ary relations, with $n < m$).

Henderson-Sellers and González-Pérez have explored these theories of abstraction and granularity for conceptual modelling [8, 9]. For example, in [8], they consider granularity for whole/part, generalization and instantiation relations, and develop best-practices when adopting a meta-model for method engineering.

The field of information and diagram visualization also makes use of abstraction techniques. For example, in [20], the authors develop an ad-hoc semantic-zooming technique to ease the navigation in complex UML diagrams, and some visual language editors like DiaGen enable the definition of abstractions [21]. However, the only purpose of these abstractions is visualization (i.e. they do not change the underlying model), they have to be manually programmed and are not reusable across different languages.

Altogether, even though abstraction has been studied in many different fields, our work is the first one proposing mechanisms for the development of reusable abstractions for modelling languages, in the context of MDE.

## 7 Conclusions and future work

In this paper, we have used generic techniques to define reusable abstractions, applicable to several modelling languages in a meta-model independent way. We have presented a catalogue of horizontal abstractions and domain-specific ones for process modelling languages. We have implemented the approach in the MetaDepth tool, which provides support for visualization of aggregate objects.

As future work, we plan to improve MetaDepth's support for genericity, e.g. making the binding more flexible. We also plan to explore the use of these techniques to define generic model slicers, to extend our catalogue of abstractions, and to provide built-in support for similarity techniques (e.g., Formal Concept Analysis) and for detecting non-confluent abstraction applications.

**Acknowledgements.** Work funded by the Spanish Ministry of Economy and Competitivity (TIN2011-24139), and the R&D programme of Madrid Region (S2009/TIC-1650). We thank the referees for their valuable comments.

## References

1. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
2. A. Polyvyanyy, S. Smirnov, and M. Weske. The triconnected abstraction of process models. In *BPM*, volume 5701 of *LNCS*, pages 229–244, 2009.
3. S. Smirnov, H.A. Reijers, and M. Weske. A semantic approach for business process model abstraction. In *CAiSE'11*, volume 6741 of *LNCS*, pages 497–511, 2011.
4. M.T. Wynn, H.M.W. Verbeek, W.M.P. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Reduction rules for YAWL workflows with cancellation regions and or-joins. *Inf. & Softw. Techn.*, 51(6):1010–1020, 2009.
5. J. de Lara and E. Guerra. From types to type requirements: Genericity for model-driven engineering. *SoSyM*, page (in press), 2011.
6. P.J. Mosterman and H. Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *Simulation*, 80(9):433–450, 2004.
7. T. Kühne. Matters of (meta-)modeling. *SoSyM*, 5(4):369–385, 2006.
8. B. Henderson-Sellers and C. González-Pérez. Granularity in conceptual modelling: application to metamodels. In *ER*, volume 6412 of *LNCS*, pages 219–232. Springer, 2010.
9. B. Henderson-Sellers. Random thoughts on multi-level conceptual modelling. volume 6520 of *LNCS*, pages 93–116. Springer, 2011.
10. K. Lano and S. Kolahdouz. Slicing techniques for UML models. *Journal of Object Technology*, 10:11:1–49, 2011.
11. A. Blouin, B. Combemale, B Baudry, and O. Beaudoux. Modeling model slicers. In *MoDELS'11*, volume 6981 of *LNCS*, pages 62–76. Springer, 2011.
12. 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.
13. B. Weber, S. Rinderle, and M. Reichert. Change patterns and change support features in process-aware information systems. In *CAiSE'07*, volume 4495 of *LNCS*, pages 574–588. Springer, 2007.
14. J.R. Hobbs. Granularity. In *IJCAI'85*, pages 432–435. M. Kaufmann, 1985.
15. Chiara Ghidini and Fausto Giunchiglia. A semantics for abstraction. In *ECAI*, pages 343–347. IOS Press, 2004.
16. Fausto Giunchiglia and Toby Walsh. A theory of abstraction. *Artif. Intell.*, 57(2-3):323–389, 1992.
17. C.M. Keet. Enhancing comprehension of ontologies and conceptual models through abstractions. In *AI*IA*, volume 4733 of *LNCS*, pages 813–821. Springer, 2007.
18. I. Mani. A theory of granularity and its application to problems of polysemy and underspecification of meaning. In *KR'98*, pages 245–255. M. Kaufmann, 1998.
19. R. Kaschek. A little theory of abstraction. In *Modellierung*, volume 45 of *LNI*, pages 75–92. GI, 2004.
20. M. Frisch, R. Dachselt, and T. Brückmann. Towards seamless semantic zooming techniques for UML diagrams. In *SOFTVIS*, pages 207–208. ACM, 2008.
21. O. Köth and M. Minas. Structure, abstraction, and direct manipulation in diagram editors. In *Diagrams*, volume 2317 of *LNCS*, pages 290–304. Springer, 2002.