Reusable Abstractions for Modelling Languages

Juan de Lara[a], Esther Guerra[a], Jesús Sánchez Cuadrado[a]

[a]*Computer Science Department, Universidad Autónoma de Madrid (Spain)*

Abstract

Model-Driven Engineering proposes the use of models to describe the relevant aspects of the system to be built and synthesize the final application from them. Models are normally described using Domain-Specific Modelling Languages (DSMLs), which provide primitives and constructs of the domain. Still, the increasing complexity of systems has raised the need for abstraction techniques able to produce simpler versions of the models while retaining some properties of interest. The problem is that developing such abstractions for each DSML from scratch is time and resource consuming.

In this paper, our goal is reducing the effort to provide modelling languages with abstraction mechanisms. For this purpose, we have devised some techniques, based on generic programming and domain-specific meta-modelling, to define generic abstraction operations that can be reused over families of modelling languages sharing certain characteristics. Abstractions can make use of clustering algorithms as similarity criteria for model elements. These algorithms can be made generic as well, and customised for particular languages by means of annotation models.

As a result, we have developed a catalogue of reusable abstractions using the proposed techniques, together with a working implementation in the MetaDepth multi-level meta-modelling tool. Our techniques and prototypes demonstrate that it is feasible to build reusable and adaptable abstractions, so that similar abstractions need not be developed from scratch, and their integration in new or existing modelling languages is less costly.

Keywords: Model-Driven Engineering, Domain-Specific Modelling Languages, Abstraction, Genericity, Domain-Specific Meta-Modelling,MetaDepth.

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 [56]. 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) tailored to specific areas of concern are frequently used [33]. Hence, a common activity in MDE is the development of environments for DSMLs, supporting the different usages of models during the project.

Email addresses: Juan.deLara@uam.es (Juan de Lara), Esther.Guerra@uam.es (Esther Guerra), Jesus.Sanchez.Cuadrado@uam.es (Jesús Sánchez Cuadrado)

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 directed to reducing model complexity for particular notations and purposes [45, 48, 50, 52, 53, 58]. In our setting, an abstraction is an 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 notations in different domains: e.g. to a class diagram to flatten an inheritance hierarchy, or to a process model to encapsulate a sequence of activities. Additionally, some abstractions are specific to particular domains. For example, the abstractions for Petri nets [45] produce smaller nets preserving liveness, safeness and boundedness, being easier to analyse. Similarly, there are techniques to produce abstracted transition systems from finite-state programs for the purpose of model-checking [4], or abstractions specific to process modelling [48, 52, 53] which produce more comprehensible models or facilitate their verification [58]. The latter abstractions are indeed applicable to any notation with a workflow-like semantics, like BPMN [46] and Activity Diagrams [47].

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 essential 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 [46], these cannot be used for other related DSMLs, like YAWL [58], Activity Diagrams or different BPMN variants. As a result, the same abstraction operations need to be encoded repeatedly for different variations of similar meta-models. Hence, a catalogue of generic abstractions that can be reused across meta-models would save significant effort in defining advanced environments for DSMLs.

In previous works [7], we developed a technique for the reutilization of model management operations based on the idea of having an extra level of indirection. Thus, operations are not defined over concrete meta-models, but over so-called *concepts* which gather the requirements that a meta-model should fulfil to make the operation applicable to the meta-model instances. In our approach, concepts have the form of meta-models, and the operations defined over a concept can be reused by binding the concept to any meta-model satisfying the concept requirements.

Another way to enable reuse is multi-level meta-modelling [2, 6, 38], a modelling paradigm that allows working with an arbitrary number of meta-levels, and not just two (i.e. model/meta-model). It can be used to build domain-specific meta-modelling languages [8] for a particular domain (e.g. the process modelling domain), which are then used to build DSMLs for different applications within the domain (e.g. software process modelling or educational process models). This approach promotes reuse because we can define operations over the domain-specific meta-modelling language, and the operations become applicable to any DSML within the domain.

Additionally, abstraction operations often use similarity criteria to decide whether a set of elements can be abstracted together. While there are many works in clustering techniques [5, 27, 40, 49], their implementation in a way that allows their application to arbitrary meta-models remains a challenge in MDE.

In this work, our goal is to facilitate the development of abstractions for modelling languages in the context of MDE, for which purpose we rely on the above mentioned techniques. In particular, 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 languages lacking support for encapsulation or aggregation. We have validated these ideas by an implementation in the MetaDepth tool [7] and their application to several case studies.

This paper is an extended version of [9], where we have expanded our catalogue of abstractions and the set of examples, we have improved our tool support, we have evaluated the generality of our approach, and we apply two additional techniques to the definition of reusable abstractions: *domain-specific meta-modelling* as a basis to build abstractions that are applicable to families of modelling languages within a domain, and *annotation models* as a configuration mechanism for those auxiliary clustering algorithms used by generic abstractions. We demonstrate the use of annotation models through the definition and reuse of similarity abstractions making use of formal concept analysis [5], relational concept analysis [27] and k-means clustering [40].

The rest of the paper is organized as follows. 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. Afterwards, Section 5 presents annotation models to configure abstractions that make use of clustering techniques. Section 6 shows an implementation of our proposal using MetaDepth. Section 7 evaluates its generality to implement existing abstraction proposals. Section 8 compares with related research, and Section 9 draws some conclusions and lines for future work.

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 abstractions provided by or developed for languages of extended use like BPMN [48, 52, 53], as well as from our own experience on the construction of DSMLs [10, 19, 20, 21].

In our setting, a model abstraction is an operation that reduces the complexity of some aspect of a model. Its purpose 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 that are considered irrelevant for the abstraction operation, 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 6. In contrast to refactorings [13], abstractions do not necessarily need to preserve the observed behaviour in the abstracted model, although some abstractions do preserve some behavioural properties, for instance when the purpose of the abstracted model is to facilitate analysis [58].

We classify abstractions according to two factors: applicability and behaviour. According to their applicability, we distinguish two types of abstractions:

- *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 [45] and the ones for workflows [58]. Being domain-specific, they can take into account the semantics of the languages in the domain and ensure the preservation of some properties, which permits their use for verification purposes. For example, the reduction techniques in [45] result in simpler models that preserve liveness, safeness and boundedness.

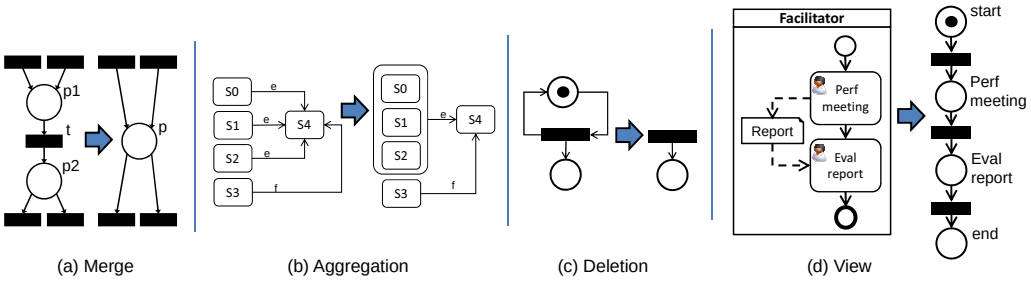

Figure 1: Classification of abstractions according to their behaviour.

Orthogonally, we identify four abstraction types according to their behaviour. The four types, illustrated through examples in Figure 1, are the following:

- *Merge*. In this kind of abstraction, a set of model elements is replaced by one element of the same type which collects the properties of the merged elements. In this way, the resulting model becomes more coarse-grained. This abstraction is widely used for verification. For example, the reduction rules for Petri nets [45] merge groups of places or transitions into a unique place or transition collecting the arcs of the merged elements. A schema of the *"Fusion of Series Places"* reduction rule [45] is shown in Figure 1(a), where the places $p1$ and $p2$ and the transition t have been merged into a new place p. The resulting net preserves liveness and boundedness, being easier to analyse as it is smaller.

  In some cases, the element replacing the group is assigned property values that result from a calculation using the properties of the merged elements. For example, some reduction rules for delay time Petri nets [30] merge transitions, and the timing of the new replacement transition is calculated using the timing of the fused transitions.

- *Aggregation*. In this case, a set of model elements is grouped hierarchically under a higher-level element, perhaps 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. As a difference with *merge*, here the aggregated elements are not deleted, but they are just placed at a different abstraction level as children of an aggregate object, yielding a more structured model. In

contrast, *merge* does not produce more structured models with elements at different levels of abstraction. Nonetheless, *aggregation* and *merge* are similar in a sense, because if we disregard the aggregated elements in the *aggregation* and only consider the introduced aggregate object, then we also obtain a more coarse-grained model.

Figure 1(b) shows an *aggregation* example, where three state machine states that have the same output transition events are aggregated under a common state, and a unique output transition is created for the introduced state.

- *Deletion*. This 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 [45] (see Figure 1(c)) and workflow nets [58]. Other examples are the deletion of the products in a process model, so that only the flow of activities (and not the produced/consumed artefacts) is retained, or the elimination of all process paths with non-maximal cost in a business process [53].

- *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 [21], 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 kind, which we illustrate in Figure 1(d). The generated Petri net reflects the task flow, but abstracts away the information regarding the performers of the activity and the produced/consumed artefacts.

Once we have seen the different model abstraction types, the next section discusses our approach to build them in a generic way, so that they can be reused for several DSMLs.

3. Building Generic Abstractions

MDE is a mostly type-centric approach because model manipulations need to be defined over the types of a concrete meta-model, becoming hard to reuse for other meta-models. The following subsections review some approaches to genericity that we will use in the rest of the paper to make abstractions reusable for different meta-models. While genericity through concepts and mixins was initially proposed in [7], genericity of model operations defined over domain-specific meta-modelling languages and annotation models (see Section 5) are novel contributions in this work.

3.1. Concepts

Assume we need an operation to simplify business process models by creating an aggregate object that abstracts the flow elements between two complementary gateways (e.g. a parallel fork and a join). Currently, there is a plethora of different business process modelling notations like Activity Diagrams, Event-driven Process Chains [55], 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

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

particular meta-model and implement the operation for it, but then this operation is not applicable for the other meta-models.

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 Figure 2. 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 fulfil some well-formedness rules 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 also 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. Moreover, the binding must preserve any subtyping relation defined in the concept, so that if two classes are related though inheritance in the concept, so must be the bound meta-model classes. And conversely, if two classes are not related through inheritance in the concept, the bound meta-model classes cannot be related through inheritance either. In [7], there is a detailed description of all well-formedness rules for bindings.

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).

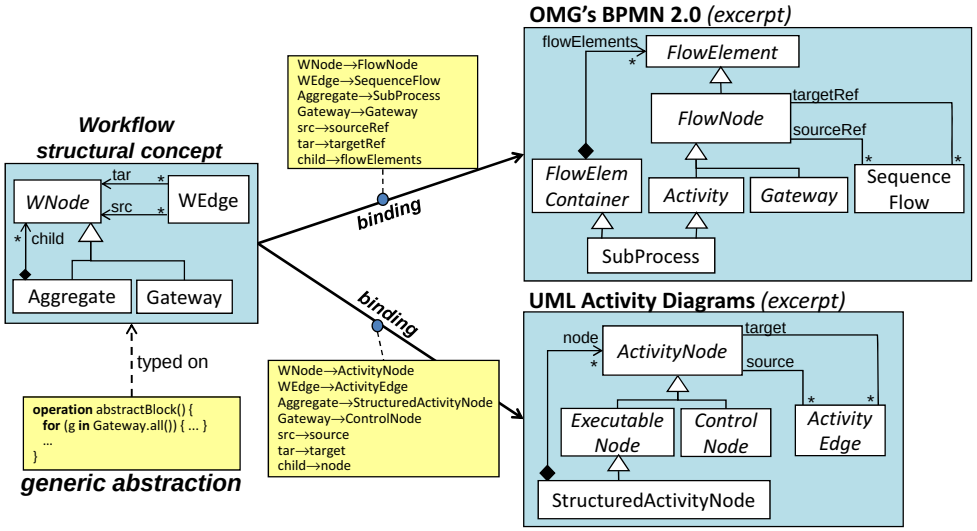

Figure 2: Generic model abstraction operation defined over a structural concept.

As an example, Figure 2 defines the generic abstraction operation `abstractBlock` using the types of the concept *Workflow*. The operation creates an aggregate object abstracting the flow elements between two complementary gateways. The 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. Once defined, 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 the binding for both meta-models. For BPMN, `WNode` is bound to `FlowNode` and `WEdge` to `SequenceFlow`. The binding permits 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.

The definition of the binding in the example, which consists of 7 mappings, allows reusing an abstraction operation of several hundred lines of code (this latter not shown in the figure). Thus, the gain of reuse is high, and further gains are possible if several abstraction operations are defined over the same concept. Moreover, the implementation details of the operation do not need to be understood by the reuser, who only needs to bind the concept elements to the concrete meta-model elements, that is, to establish the role played by the concrete meta-model elements in the abstraction operation.

3.2. 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* in Figure 2 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* [7], which abstract away the accidental details introduced by the specific choice of structure in the concept behind a suitable interface. This interface consists of a number of operations associated to the classes of the concept. Thus, *hybrid concepts* are like structural ones but require a number of operations from the elements they contain. The binding is then obtained by mapping the elements and implementing the operations declared in the concept.

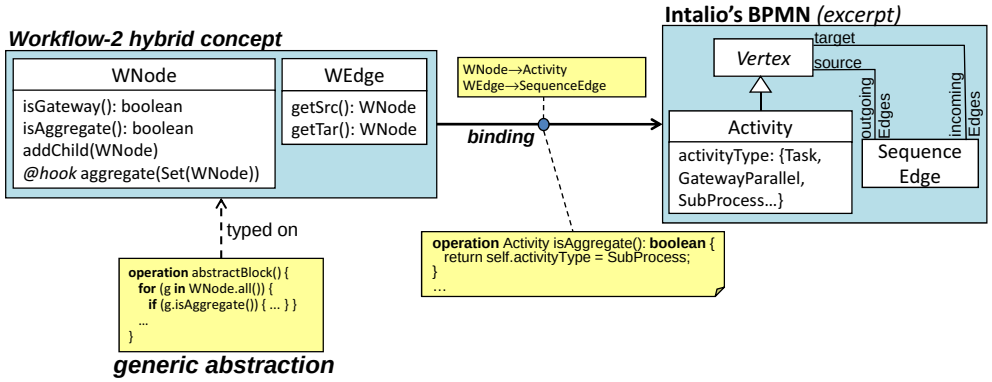

Figure 3: Binding to Intalio's BPMN meta-model with a hybrid concept.

Figure 3 shows the definition of the hybrid concept *Workflow-2*, which is a more flexible version of the structural concept presented in Figure 2, as it imposes less structural requirements.

In order to bind this concept to the Intalio's BPMN meta-model we need to implement 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 to perform some computation over the attribute values of the abstracted elements. To express these configurations we rely on the well-known notion of *hook*, typically used in the Template Method design pattern [15] to leave the variable parts of an algorithm open. In the case of our abstractions, hook operations provide a default implementation, and only need to be implemented during the binding if a special behaviour is required. They are available in both structural and hybrid concepts. For instance, in Figure 3, 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).

3.3. Extending the modelling language

Some abstraction operations create an aggregate object grouping a set of similar or related model elements. However, some notations are not designed with hierarchical or encapsulation elements, and therefore we cannot apply these abstractions to them, as they lack a class acting as aggregate. 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 [7]. Then, we use concepts to express the requirements that a meta-model should fulfil to become extendible by the mixin.

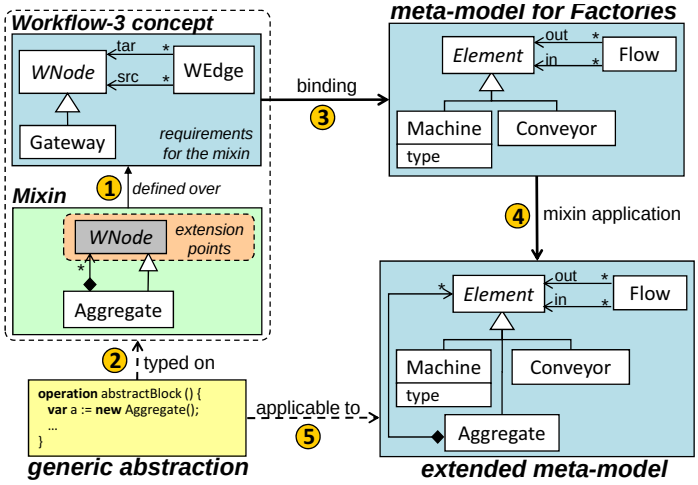

Figure 4: Generic model abstraction operation defined over a mixin.

As an example, the mixin in Figure 4 adds an aggregate class to any meta-model boundable to the *Workflow-3* concept. As a difference from the previous examples, the *Workflow-3* concept does not demand the existence of an aggregate class in the bound meta-models, but this class will be added by the mixin. The generic operation is defined over the glueing 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 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 the meta-model instances. Moreover, this kind of mixin preserves the compatibility of models conformant to the original meta-model with the extended one.

3.4. Considering families of modelling languages

Some abstractions are horizontal, applicable to a large range of unrelated DSMLs in different domains, whereas some others are applicable to families of modelling languages within the same domain. A way to construct such families of DSMLs is through the use of appropriate domain-specific meta-modelling (DSMM) languages which provide primitives of the specific domain [8]. Standard modelling languages are defined through a meta-model and are instantiated at the meta-level below. DSMM languages are defined through a meta-model as well, but can be instantiated at the next two meta-levels below, i.e., they span three meta-levels.

Figure 5 shows an example. The upper part corresponds to a simple DSMM language for process modelling which makes available meta-modelling primitives such as `Task`, `Artifact` or `Performer`. These domain-specific primitives make process modelling more natural than using all-purpose primitives provided by general-purpose meta-modelling languages, like `Class` or `Association` [18, 28]. In this way, we can use the DSMM language to build DSMLs for particular applications within the process modelling domain, like software processes (middle left) or educational processes (middle right), which in their turn can be instantiated to define particular software and educational models (below).

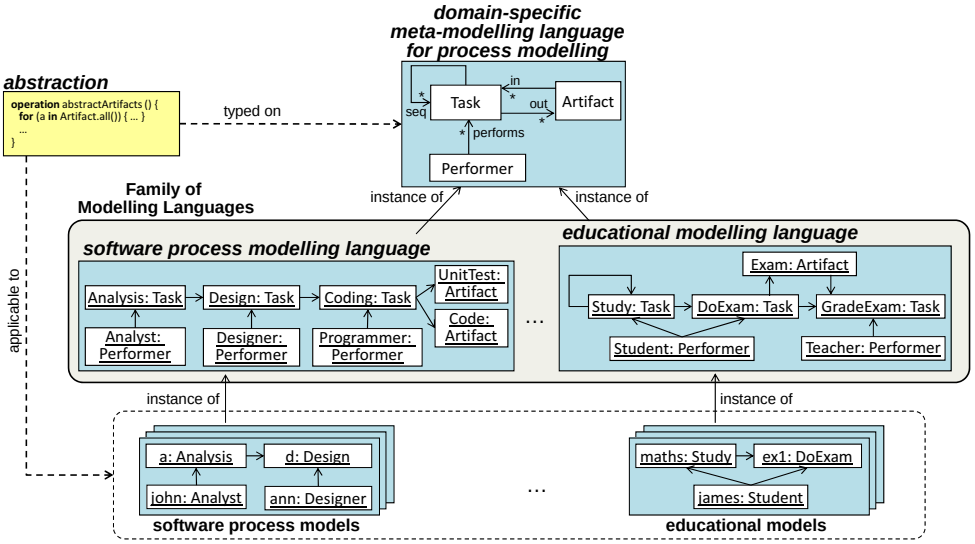

Figure 5: Model abstraction operation defined over a domain-specific meta-modelling language.

In this context, we can define abstraction operations over the primitives provided by the DSMM language (upper level), and the operations become applicable to all models of the DSMLs built with the DSMM language (lower level). This is possible as the objects in the lower level

are indirect instances of the top-level ones. For instance, `john` is an instance of `Analyst`, and `Analyst` is an instance of `Performer`; hence, `john` is (indirectly) a `Performer`. In this way, all abstraction operations defined over the DSMM language are directly reusable by the family of languages built with it.

Now that we have presented the techniques we propose to make abstractions generic, customizable and adaptable, in the next section we provide a catalogue of reusable abstractions defined with them.

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 a 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.

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 in the catalogue, we provide three different binding possibilities for each one: (i) from a structural concept, which is the simplest approach when the structure of the concept and the structure of 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. 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

Source parallel. This operation abstracts a maximal set of objects which is referenced from the same source objects. There are two variants of this abstraction: *aggregation* and *merge*. The *aggregation* variant creates an aggregate that encapsulates all objects with the same source, whereas the *merge* variant replaces the parallel objects by another one of the same type (or a subtype). In both cases, all references from/to the abstracted objects are replicated in the created abstraction.

The structural concept that corresponds to the *merge* variant is shown to the left of Figure 6. The abstraction can be reused for a particular notation by specifying the type of the objects to abstract (class `Item`), the type of the source objects that reference them (class `Context`

and reference `target`), and the aggregate type to create (`Aggregate`)[2]. In case of the *aggregation* variant, the `Aggregate` class typically has also a `child` reference to the `Item` class (see the left of Figure 8 as an example for a different abstraction operation). Finally, the concept includes two hook methods: `canAggregate` to configure extra conditions that the objects to be abstracted need to fulfil, and `aggregate` to compute possible attribute values in the created aggregate.

The right of Figure 6 shows the working schema of the *merge* variant, where an aggregate object is created in place of all items having a common source. The operation folds the `target` references, copies the incoming and outgoing references from the original items to the created aggregate, and deletes the abstracted items.

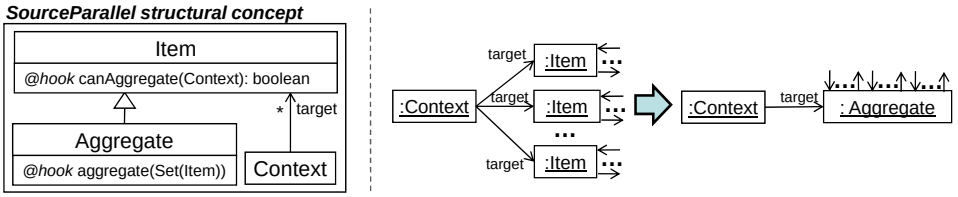

Figure 6: Source parallel *merge* abstraction: concept (left) and behaviour (right).

Figure 7 shows an application of this abstraction to a DSML for computing systems. The language contains two types of computing units (clusters and processors) characterized by a clock rate, and which can be connected to storage units. We have depicted the binding from the concept to the meta-model as tags. Thus, `Cluster` plays the role of `Aggregate`, while `ComputingUnit` plays the role of both `Item` and `Context`. Moreover, the hook method `aggregate` is configured to assign the introduced aggregate the minimum clock rate of all aggregated computing units (this is done in OCL by collecting the clock rate of each computing unit and picking the minimum) and the number of merged units. The right of the figure shows two consecutive applications of this abstraction to a model, where processors are graphically represented as chips, and clusters as blobs. The abstraction operation automatically copies the referenced memory storages from the merged computing units to the created cluster.

Target parallel. It abstracts a maximal set of objects that refer to the same target objects. Thus, this abstraction is like the previous one, but considers target parallel objects instead of input ones. The structural concept for its *aggregation* variant is shown to the left of Figure 8, in which the only difference with respect to the concept of Figure 6 is the direction of the `target` reference and that it requires a `child` reference to store the items in the created aggregate. 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.

[2]Please note that the class representing the type into which the objects will be abstracted is called `Aggregate` in our concepts, regardless we use the variant *merge* or *aggregation*.

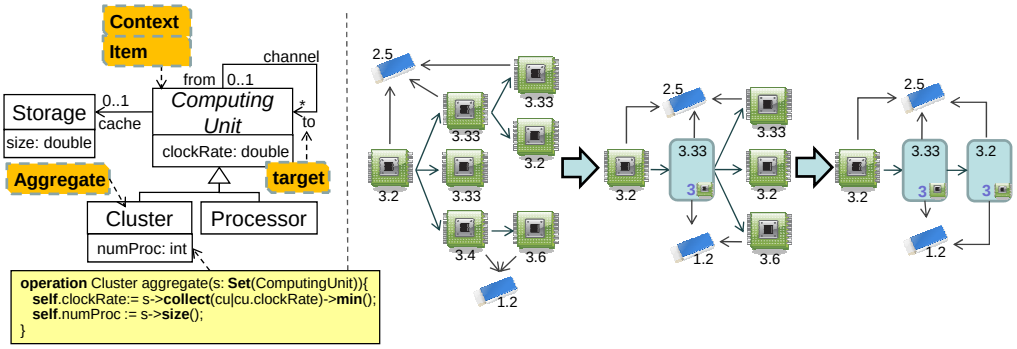

Figure 7: Source parallel *merge* abstraction: binding (left) and application (right).

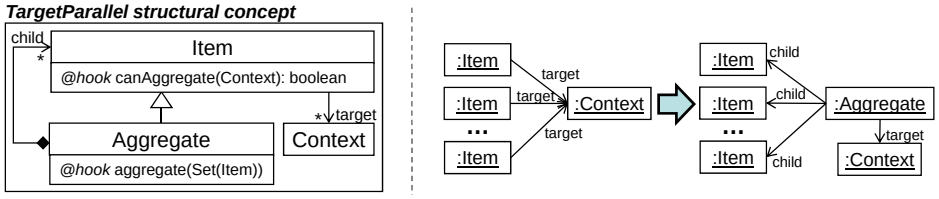

Figure 8: Target parallel *aggregation* abstraction: concept (left) and behaviour (right).

Figure 9 shows an application of this abstraction over UML 2.0 Statecharts, to encapsulate sets of states that reach the same state through the 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*. The defined binding maps both `Item` and `Context` in the concept to `Vertex` in the meta-model, as we want to detect states connected to the same states. We have customized the abstraction by overriding the hook operation `canAggregate` to include only target states with the same `Trigger`, and `aggregate` to attach an appropriate `Trigger` to the created transition. Implementation-wise, we have to use 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 Figure 9 shows the in-place application of this unflattening to a Statechart, which abstracts all states with a transition triggered by "e" to *S4*.

Parallel. It abstracts a maximal set of objects with the same source and target elements. The structural concept for this abstraction is shown to the left of Figure 10 in its variant *merge* (i.e. the abstracted items are deleted and substituted by a single new item of type `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 tran-

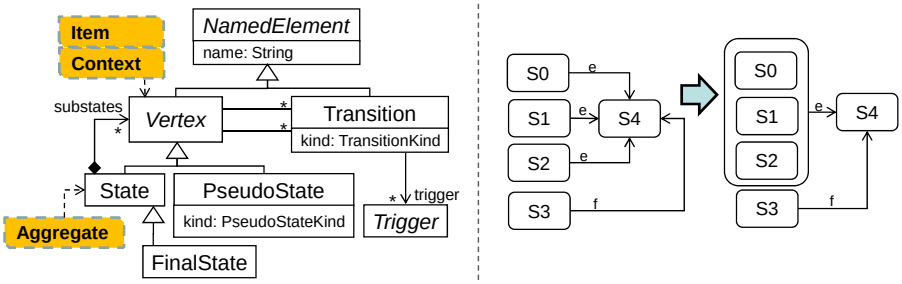

Figure 9: Target parallel *aggregation* abstraction: binding to Statecharts (left) and application (right).

sitions 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`, and not directly through a reference as demanded by the structural concept.

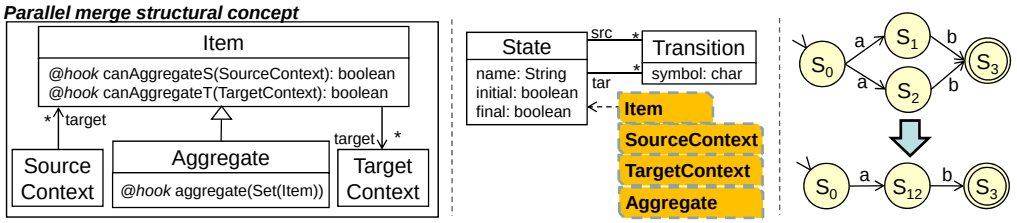

Figure 10: Parallel *merge* abstraction: concept (left), binding to automata (middle), and application (right).

Sequential. It abstracts a maximal sequence of linearly connected objects, and admits both variants *merge* and *aggregation*. The left of Figure 11 shows the structural concept for the *aggregation* variant. The right of the figure illustrates the working schema of this abstraction: 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.

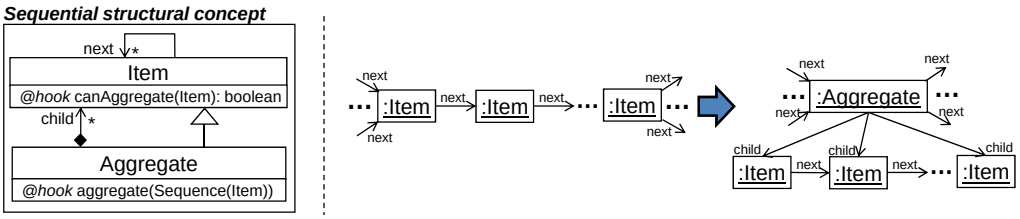

Figure 11: Sequential *aggregation* abstraction: concept (left) and behaviour (right).

Figure 12 shows to the left a binding of the concept to the BPMN 2.0 meta-model, where we have mapped `Item` to `FlowElement` and `Aggregate` to `SubProcess`. Reference `child` in the concept could be directly mapped to reference `flowElements`, however we use the hybrid version of the concept because we have to navigate between `FlowNodes`

through `SequenceFlows`, and not directly with a reference. In this case, we have customized the `canAggregate` hook method to forbid having gateways as first or last element in the abstracted sequence, and the `aggregate` method to create the start and end events of the subprocess. The right of the same figure shows an example application.

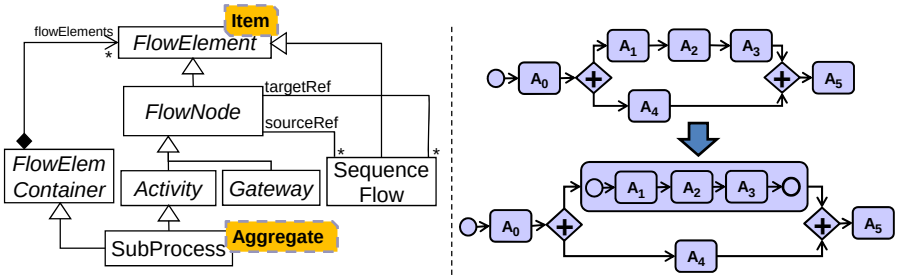

Figure 12: Sequential *aggregation* abstraction: binding to BPMN (left) and application (right).

Block. It abstracts a connected set of elements between two given objects, and supports both *merge* and *aggregation* variants. Moreover, it is also possible to configure whether the two objects delimiting the block should be abstracted or not. As a difference from the previous cases, here the location where the abstraction will be applied is not detected automatically, but the two objects delimiting the block must be given as parameters to the abstraction operation. Figure 13 shows to the left the hybrid concept for this abstraction. `InitialContext` and `FinalContext` correspond to the type of the objects delimiting the block (the concrete objects will be passed as parameters), and `Item` corresponds to the type of the abstracted elements. As the concept is hybrid, the binding needs to provide operations `getNext` to obtain all adjacent items of a given one, and `addChild` to add the items in the block to the aggregate. The right of Figure 13 shows the working scheme for the *aggregation* variant of this abstraction and including the delimiters of the block in the aggregate.

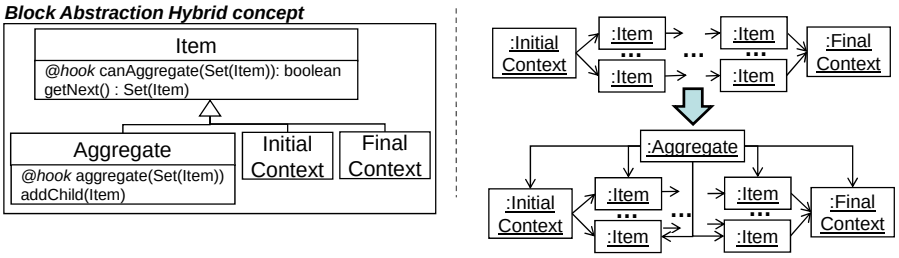

Figure 13: Block *aggregation* abstraction: hybrid concept (left) and working scheme (right).

Figure 14 shows a binding to BPMN and an example application. For the case of BPMN, both block delimiters need to be gateways, typically of complementary type. Role `Item` is bound to `FlowElement` so that both `FlowNodes` and `SequenceFlows` are aggregated. While this abstraction needs to be provided with the initial and final context objects, we have designed a specialization of this abstraction for BPMN which automatically detects abstraction opportunities (see next subsection).

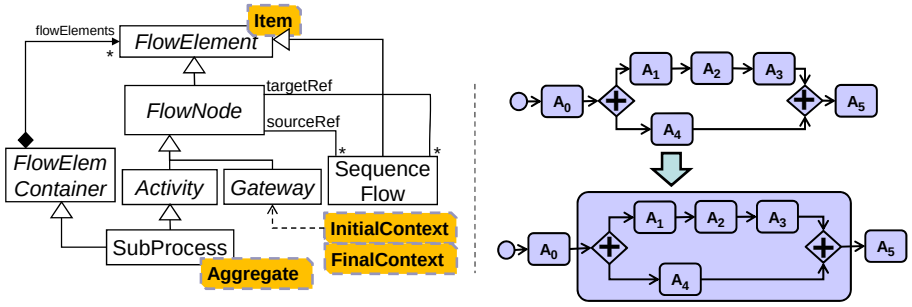

Figure 14: Block *aggregation* abstraction: binding to BPMN (left) and example (right).

Similarity. It abstracts a maximal set of objects considered similar by a custom-defined operation. The left of Figure 15 shows the structural concept for the *aggregation* variant of this abstraction. The right of the figure shows an application example to BPMN 2.0 models. As in the abstraction heuristic presented in [52], we consider similar those tasks that produce or consume the same sets of `Data Objects` and use the same resources (in the model example, if they are performed by the same `HumanPerformer`). Thus, we configure this abstraction by binding class `Item` in the concept to `FlowElement`, `Aggregate` to `SubProcess`, and implementing the `isSimilarTo` hook method. Please note that we do not need to bind BPMN `Data Objects`, but the `isSimilarTo` operation, when implemented for BPMN, is in charge of comparing the data objects attached to `FlowElements`.

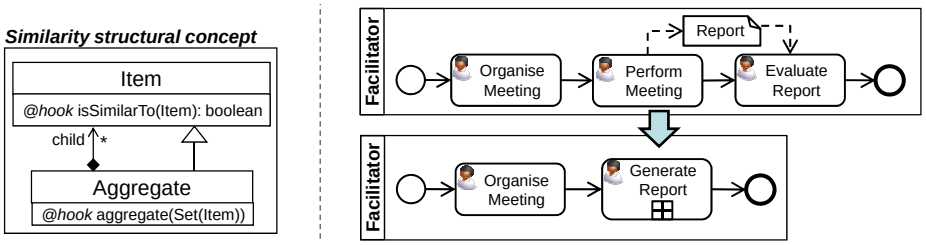

Figure 15: Similarity *aggregation* abstraction: concept (left) and application to BPMN (right).

Loop removal. It removes loops between two types of objects. The associated concept is shown in Figure 16(a). We consider two variants: the first one removes one of the references to break the loop, whereas the second one removes the object with type `Item` and its references. The second variant is illustrated in Figure 16(b). Figure 16(c) shows an application of this abstraction that simplifies Petri nets, while preserving their behaviour, according to the *elimination of self-loop places* reduction rule [45]. Thus, a place is removed if it has at least one token and has no further input or output arcs, which is checked in the implementation of the hook method `canRemove`. Figure 16(d) shows an application to YAWL nets [58] that eliminates self-loop tasks. A different binding into YAWL would enable the elimination of self-loop conditions as well [58].

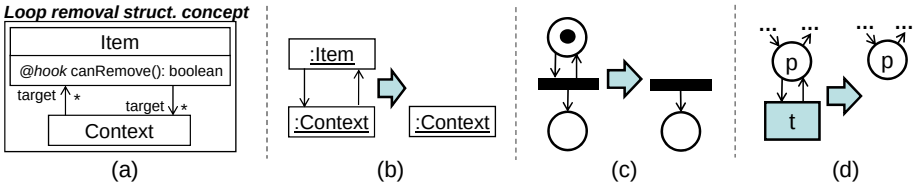

Figure 16: Loop removal abstraction: concept (a), behaviour (b), and applications to Petri nets (c) and YAWL nets (d).

4.2. Domain-specific abstractions

Oftentimes, abstractions are specifically designed for a particular domain, such as Petri nets [45] or process models [52, 58]. Their definition over concepts makes them meta-model independent, and therefore reusable for other languages with similar semantics. If an abstraction for a particular domain is especially complex, it can still be built by reusing and customizing horizontal abstractions (like the ones in our catalogue). However, in such cases, we can benefit from specializing 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 with minimal configuration effort. In the following, we briefly describe some abstractions for different domains.

Petri nets. The left of Figure 17 shows on top a hybrid concept named *ProcessHolder* capturing the essential elements of Petri net-like languages. The `Holder` role is played by place-like entities, while `Process`es are the active elements, played by transition-like elements. This concept allows querying the model through the operations `tokens`, `inputs` and `outputs`, which hide the specific structure used by different boundable meta-models. The figure shows in addition a typical organization of concepts, where concepts needed for specific operations, like abstraction and simulation, extend a base concept. While the concept for simulation needs the implementation of an additional mutator operation `addToken`, the one for abstractions in the bottom left can be configured through a number of hooks to perform additional actions when deleting a holder or a process, when aggregating two holders, or when merging their inputs.

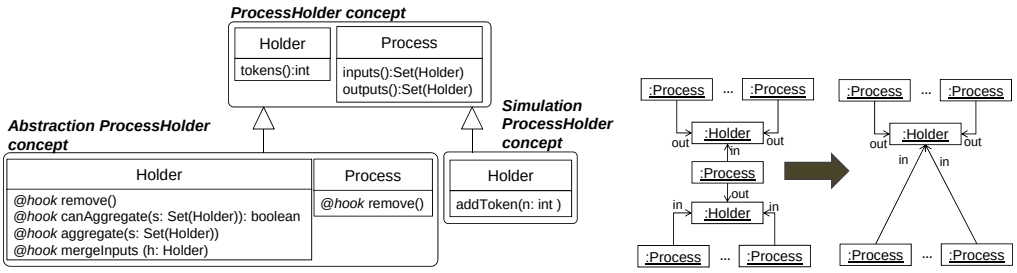

Figure 17: Concepts for Petri net-like languages (left). Fusion of serial places abstraction (right).

As an example, the right of the same figure shows the working scheme of the abstraction operation *Fusion of Serial Places* (FSP), which is defined over the concept *Abstraction ProcessHolder*. This abstraction merges two holders in sequence. The abstraction is taken from the standard catalogue of Petri-net reduction rules, which preserve liveness, safeness

and boundedness [45]. The rationale is to obtain a simpler model through abstraction, which becomes more efficient to analyse, as the resulting state space is smaller.

As we have defined the abstraction operation over a concept, we can apply the operation to languages with Petri net-like semantics, to which we can bind the *Abstraction ProcessHolder* concept. For instance, Figure 18 shows the binding of the concept to a DSML for factories. This DSML is made of three types of machines: generators, assemblers and terminators. Generators model processes that add parts in the factory, terminators take parts out of the factory and assemblers transform parts. Parts are transferred between machines through conveyors. The binding maps `Holder` and `Process` in the concept to the classes `Conveyor` and `Machine` in the meta-model, respectively, so that the abstraction will be used to fusion serial conveyors. The binding needs to implement the required query operations `tokens`, `inputs` and `outputs` (we only show the implementation of the first one for brevity). In addition, the binding implements two hook operations: `mergeInputs`, which redirects the references from input machines to the merged conveyor, and `aggregate`, which calculates the number of parts to be placed in the merged conveyor. The right of the figure shows the application of the FSP abstraction to a factory model, leading to the merging of two adjacent conveyors.

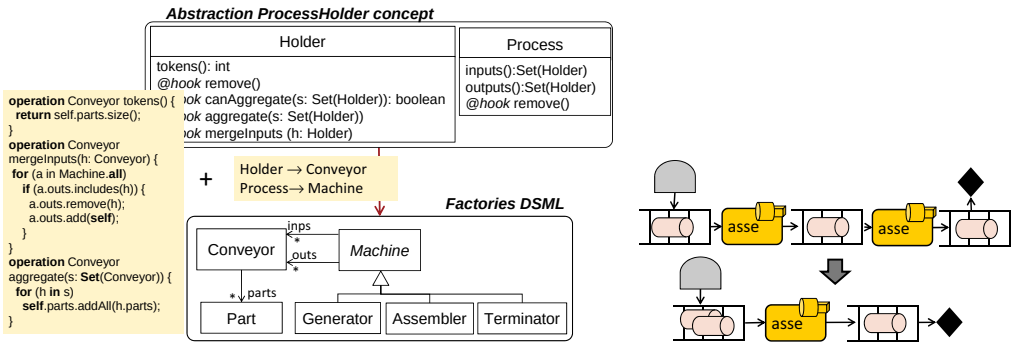

Figure 18: Fusion of serial places abstraction: binding to a DSML for factories (left) and application (right).

Workflows. Another domain for which we have defined specific abstraction operations over suitable concepts is process modelling. For example, Figure 2 showed the concept *Workflow* for process modelling languages, which gathers 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 Figure 24. This abstraction is a specialization of the `block` horizontal abstraction for workflow languages. Building this specialization has two advantages: first, it is easier to identify the elements to bind for languages of the workflow domain, and second, it can use the workflow semantics to detect where to apply the abstraction automatically (using the horizontal abstraction directly would require giving the initial and final context objects delimiting the block as parameters). Other abstractions we have built (which we took from [53]) include sequential merge, loop abstraction and elimination of dead-end paths.

Domain-specific process modelling. Another way to define abstractions for particular domains is by the use of domain-specific meta-modelling, as explained in Section 3.4. In this ap-

proach, a domain-specific meta-modelling language is used to define a family of languages for a certain domain (see Figure 5). The abstractions defined over types of the domain-specific meta-modelling language are applicable to any DSML built with it.

Up to now, we have defined several abstractions for domain-specific process modelling languages. For example, Figure 19 shows the working schema of a view abstraction that transforms process models into Petri-nets. The upper-left of the figure corresponds to a domain-specific meta-modelling language for process modelling (it is a refinement of the one shown in Figure 5). Below, it is instantiated to define a modelling language for software processes. The bottom-left shows a particular software process model. The view abstraction is defined as a model-to-model transformation, using the types of the meta-modelling language. Our multi-level framework is able to deal with *indirect* types [8], so that the transformation becomes applicable to the bottom-most model. Hence, the rule `Task2Place` defined over `Task` is applicable to objects `a`, `d` and `c`.

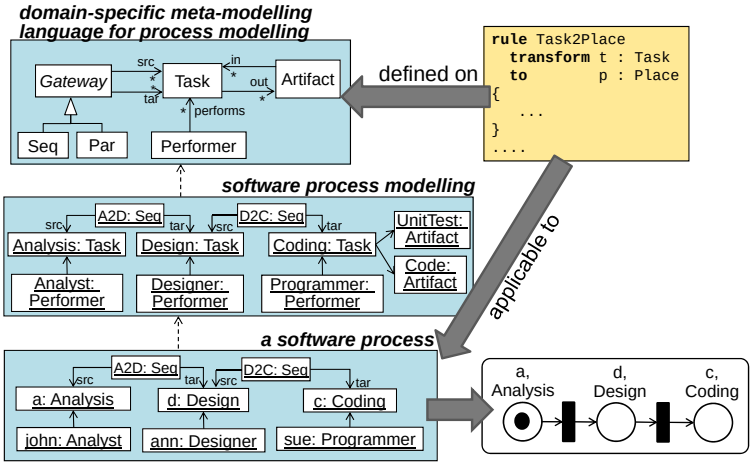

Figure 19: A view abstraction for domain-specific process modelling.

5. Configuring the abstraction operations through annotation models

The abstractions presented so far include hook methods, like `isSimilarTo` or `canAggregate`, to customise the object similarity criteria used by the abstractions. If the similarity criteria are simple, it is straightforward to implement the hook methods. However, some abstractions may rely on complex clustering techniques – like those of Formal Concept Analysis [5] (FCA), or k-means [40] – to estimate the similarity of objects. All these techniques have in common that they compare objects according to a subset of their attributes, and in some cases, they partition the attribute values in equivalence classes. As a result of this comparison, objects which are more similar to each other are classified in the same cluster. Hence, the challenges are: (i) how to build clustering algorithms reusable for different meta-models, (ii) how to configure the objects to be compared and the attributes used in the comparison, and (iii) how to use the resulting clusters of objects as similarity criteria in our abstraction operations.

A scheme of our solution is shown in Figure 20. As before, the abstraction operation is attached a concept, to be bound to a particular meta-model. In addition, the operation can use

generic implementations of clustering algorithms through the `isSimilarTo` and `canAggregate` hooks. The clustering algorithms are not tied to a domain meta-model, but they are typed on a predefined annotation meta-model. The configuration of the clustering algorithm for a domain meta-model is done through an *annotation model* that identifies the meta-model classes and attributes involved in the comparison, as well as the possible equivalence classes for the attributes. The clustering algorithm remains generic as it is typed on the annotation meta-model, and only uses the configuration information provided by the annotation model.

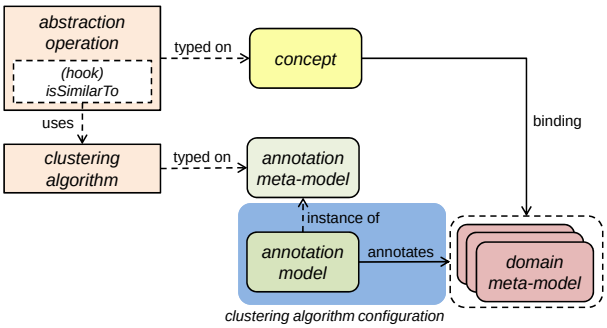

Figure 20: Scheme of the configuration of generic abstraction operations through annotation models.

Figure 21 shows the meta-model of our annotation models. It allows defining the type of the objects to be clustered by the algorithm, the features used for their comparison, and partitions for attribute values by means of equivalence classes. Classes `Object`, `Relation` and `Property` are pointers to the objects, references and fields to be used by the clustering algorithm. There should be at least one `Object` with its attribute `classifiable` set to *true*, which identifies the element type to cluster, taking into account the annotated features. The meta-model also supports the annotation of complex navigation expressions, e.g. involving the traversal of several relations, by using the class `IndirectRelation`. Finally, the children of class `Equivalence` allow defining equivalence classes for attribute values. For attributes of type *real*, we can indicate the range of values for the equivalence class (attributes `min` and `max` to specify a bounded range, or `noMin` and `noMax` if it is unbounded).

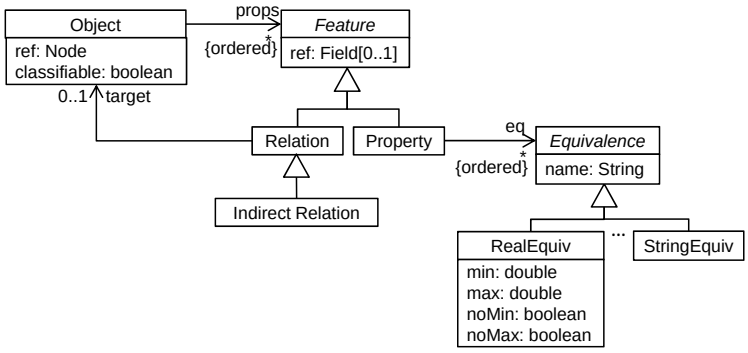

Figure 21: Annotation meta-model for clustering.

Next, we describe two generic clustering algorithms that we have defined over this meta-

model, and show some examples of abstraction operations that use them.

***k*-means based similarity.** The Machine Learning community has proposed many clustering techniques. In particular, the *k*-means technique [40] classifies a set of objects in *k* clusters. The objects are encoded as numerical vectors describing their features, and then they are grouped according to their distance to the mean of each cluster. In this way, the algorithm minimizes the expression

$$argmin \sum_{i=1}^{k} \sum_{x_j \in S_i} \|x_j - \mu_i\|^2$$

which is the distance of each object x_j to the mean value μ_i of its cluster S_i. We have implemented this algorithm using the types of our annotation meta-model (i.e. object, relation, etc.), so that it can be reused for different abstractions and modelling languages, as we illustrate below.

Some works reported in the literature use *k*-means to find sets of similar objects, which are then abstracted. For example, [52] uses this algorithm to cluster activities in process models based on their input/output data objects and on the resources they use. The algorithm needs a fixed value for *k*, and the authors suggest values from 5 to 7 in process modelling [52].

To implement this abstraction for BPMN using our approach, we first need to annotate its meta-model to indicate the elements used to measure the similarity of activities. The annotations will be used by our *k*-means algorithm to automatically build clusters of similar activities. Then, we can use a *similarity* horizontal abstraction to aggregate the activities in each cluster. The advantage is that we do not need to hand-code the algorithm to identify the sets of similar activities, but the configuration is an instantiation of the annotation meta-model.

Figure 22 shows the annotation model used to configure the similarity criteria for this abstraction. The annotation model contains an instance of `Object` which points to `Activity` in the BPMN meta-model, as this is the element to compare and classify. Two similarity criteria are defined for activities: used resources and input/output data objects. The first one corresponds to the relation `resources` stemming from `Activity`, therefore it is enough to annotate this relation through an instance of `Relation`, as shown in the figure. For the second criterion, we need to navigate from `Activity` objects to the data objects they access, which is done by the expression attached to the `IndirectRelation` annotation[3].

Additionally, to use the *similarity* horizontal abstraction, we need to bind the BPMN meta-model to the concept of Figure 15. In this case, `Activity` plays the role of `Item` and `SubProcess` will be used as `Aggregate`. The abstraction would use the result of the *k*-means algorithm to aggregate the similar activities.

FCA/RCA-based similarity. Formal Concept Analysis (FCA) [5] is a clustering technique based on the calculation of so-called *FCA concepts*[4], which are clusters of objects sharing maximal sets of properties. As a difference with *k*-means, the number of clusters does not

[3]The expression is simplified, as the navigation goes through intermediate `InputOutputSpecification` objects.

[4]Coincidentally, FCA concepts have the same name as generic concepts, but both terms are unrelated.

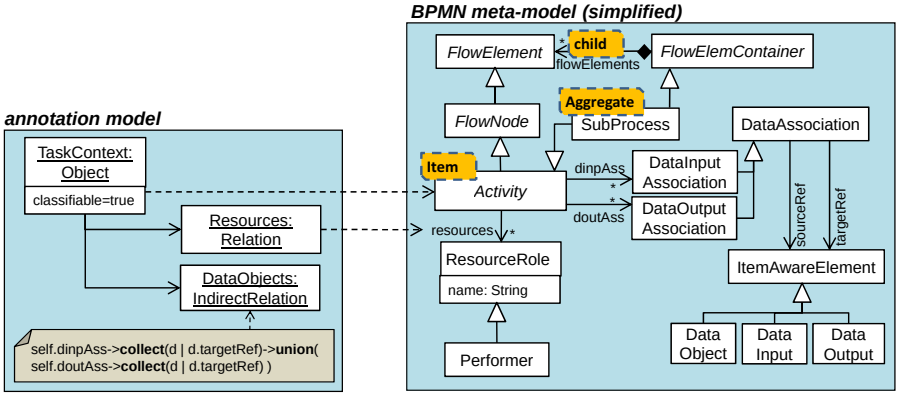

Figure 22: Configuring k-means for BPMN through an annotation model, and binding for similarity abstraction.

need to be fixed a priori. Moreover, FCA requires the definition of equivalence classes for all attributes involved in the comparison, which is called data scaling [27]. Some simple comparison criteria can also be supplied for references, which by default is reference equality.

Relational Concept Analysis (RCA) [27] extends FCA with a richer classification for objects with references, as in this case, the algorithm builds additional concept lattices for the target of references (i.e. it classifies the target objects in clusters as well).

We have implemented generic algorithms for FCA and RCA, using the types from our annotation meta-model. As an application example, Figure 23 shows the meta-model of a DSML for computing systems (upper right), which we already used for the source parallel abstraction. The left of the figure shows the annotation model we need to provide to our RCA algorithm to classify computing units according to their clock rate and the size of their storage cache. The classifiable object in the meta-model is `ComputingUnit`, as we want to classify both processors and clusters. Two properties are defined as comparison criteria: the attribute `clockRate`, for which three equivalence classes are provided (low, medium and high), and the attribute `size` of the computing unit's storage, accessed through the reference `cache`, with two possible categories (small and big).

We can use the results of the clustering algorithm in different abstraction operations. For instance, in order to abstract similar computing units into a single `Cluster` object, we can bind the DSML to the concept for the *similarity merge* abstraction as follows: `ComputingUnit` is bound to `Item`, and `Cluster` to `Aggregate`. The bottom-right of Figure 23 shows the result of applying this abstraction to the same model used in Figure 7.

6. Tool Support

The approach and abstractions presented in this paper have been implemented and validated using MetaDepth[5], a meta-modelling framework which supports multi-level meta-modelling and textual modelling [7]. It is integrated with the Epsilon family of languages[6], which permits

[5]The tool and the abstractions are available for download at `http://astreo.ii.uam.es/~jlara/metaDepth/`
[6]`http://eclipse.org/gmt/epsilon/`

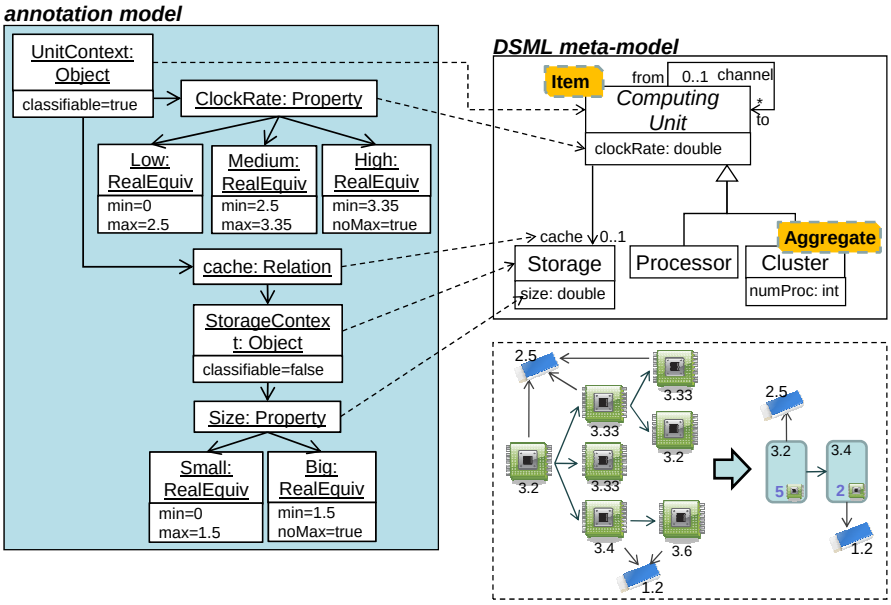

Figure 23: Configuring RCA for a DSML through an annotation model, binding for similarity abstraction and application.

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.

6.1. Concept-based reusability

Concepts, mixins and bindings are specified textually in MetaDepth. As an example, Listing 1 shows the definition of the structural concept shown in Figure 2. 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.

```
1  concept Workflow
2    (&M, &WNode, &WEdge, &Aggregate,
3     &Gateway, &child, &src, &tar) {
4    Model &M {
5      abstract Node &WNode {}
6      Node &WEdge {
7        &src : &WNode;
8        &tar : &WNode;
9      }
10     Node &Aggregate : &WNode {
11       &child : &WNode[*];
12     }
13     Node &Gateway : &WNode {}
14   }
15 }
```

Listing 1: Structural concept.

```
1  operation blockAbstraction() : Boolean {
2    var comp : &WNode = null;
3    for (gw in &Gateway.allInstances())
4      if (gw.isSplit()) {
5        comp := getJoin(gw);
6        if (comp<>null) {
7          var sq : Sequence(&WNode);
8          sq.addAll(getAllInBetween(gw, comp));
9          createAggregateFor(sq, gw, comp);
10         return true;
11       }
12     }
13   return false;
14 }
15 operation &Gateway isSplit() : Boolean{...}
16 ...
```

Listing 2: Block abstraction operation.

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) [34], whereas the *view*-generating abstraction operations are defined with the Epsilon Transformation Language (ETL) [35]. Listing 2 shows an excerpt of the `block` abstraction operation using EOL (most auxiliary operations and methods are not shown), which uses the types of the *Workflow* concept of Listing 1.

To reuse the generic operation with a given modelling language, we need to provide a binding from the concept to the meta-model of the language. 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).

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

Listing 3: Binding to meta-model.

The users of a generic abstraction only have to define the binding from the concept to their modelling languages. The binding in Listing 3 allows reusing the `block` abstraction operation, which consists of several hundred 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, hence 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" (i.e. the concept), and so the different elements in the concept become similar to roles in design patterns.

6.2. Multi-level based reusability

MetaDepth supports modelling using an arbitrary number of meta-levels [6]. The number of levels in which an element can be instantiated is controlled by its *potency* [2]. This is a natural number attached to classes, edges and fields, which gets automatically decreased at each lower meta-level. When it reaches zero, it is not possible to instantiate the element in lower meta-levels. The interested reader can consult [6, 8] for further details.

We can use MetaDepth's multi-level capabilities to define domain-specific meta-modelling languages and abstractions for them. As an example, Listing 4 shows an excerpt of the definition of the domain-specific meta-modelling language to define process modelling languages, similar to the one shown in Figure 19. Line 1 declares the meta-modelling language with potency 2 (indicated after the symbol '@'), which means that it can be instantiated at two meta-levels. Lines 2–7 declare meta-class `Task`. Since it does not specify a potency on its own, it takes the potency of the enclosing model (i.e. 2). It declares an attribute `name` with potency 1, which can receive a value at the next meta-level, a boolean flag `initial` with potency 2 (taken from `Task`) which can receive a value two meta-levels below, and two references to input and output artefacts. Similarly, classes `Performer` and `Artifact` are declared in lines 8–11 and 12, respectively. Finally, an abstract class `Gateway` is defined in lines 14–17 with references to source and target tasks, together with two subclasses `Seq` and `Par` in lines 18 and 19.

```
1  Model ProcessModel@2 {
2    Node Task {
3      name@1 : String[0..1];
4      initial : boolean;
5      ins : Artifact[*];
6      outs : Artifact[*];
7    }
8    Node Performer {
9      name : String;
10     performs : Task[*];
11   }
12   Node Artifact {}
13
14   abstract Node Gateway {
15     src : Task[*];
16     tar : Task[*];
17   }
18   Node Seq : Gateway {}
19   Node Par : Gateway {}
20 }
```

Listing 4: Process meta-modelling language.

```
1  ProcessModel SoftwareProcess {
2    Task Analysis { name = "requirements, analysis"; }
3    Task Design { name = "high−leve design, low−level design"; }
4    Task Coding { name = "coding, unit testing"; }
5
6    Performer Analyst { perf : Analysis{performs}; }
7    Performer Designer { perf : Design{performs}; }
8    Performer Programmer { perf : Coding{performs}; }
9
10   Seq A2D {
11     from: Analysis{src};
12     to : Design{tar};
13   }
14   Seq D2C {
15     from: Design{src};
16     to : Coding{tar};
17   }
18 }
```

Listing 5: Software process modelling language.

Listing 5 shows a simple software process modelling language, built using the previous domain-specific meta-modelling language (i.e. it is an instance of the meta-model in Listing 4). The language defines three tasks types (`Analysis`, `Design` and `Coding`) and three types of `Performer` (`Analyst`, `Designer` and `Programmer`). Each task type can assign a value to `name` because this attribute was defined with potency 1, and each performer declares the type of task he can get involved in by explicitly instantiating the `performs` reference. Finally, the instances of `Seq` in lines 10 and 14 allow navigating from `Analysis` to `Design` tasks, and from `Design` to `Coding` tasks.

We can use this language to define software process models, like the one shown in Listing 6. In line 2, we can assign a value to `initial`, as this attribute was given potency 2 in Listing 4.

```
1  SoftwareProcess aProcess {
2    Analysis a { initial = true; }
3    Design d {}
4    Coding c { }
5
6    Analyst john{ name = "John"; perf = a; }
7    Designer ann{ name = "Ann"; perf = d; }
8    Programmer sue{ name = "Sue"; perf = c; }
9
10   A2D a2d { from = a; to = d; }
11   D2C d2c { from = d; to = c; }
12 }
```

Listing 6: Software process model.

```
1  @lazy
2  rule Task2Place
3    transform task : Source!Task
4    to place : Target!Place
5  {
6      place.name := task.name;
7      if (task.initial) place.tokens := 1;
8  }
9
10 rule Gateway2Transition
11   transform gateway : Source!Gateway
12   to transition : Target!Transition
13 {
14     for ( ref in gateway.references("tar") )
15       for ( task in gateway.value(ref) )
16         new Target!ArcTP(transition, task.equivalent());
17
18     for ( ref in gateway.references("src") )
19       for ( task in gateway.value(ref) )
20         new Target!ArcPT(task.equivalent(), transition);
21 }
```

Listing 7: View abstraction for business process models.

The abstraction operations built over a domain-specific meta-modelling language are applicable to the instances of any modelling language built with it. As an example, Listing 7 shows a model-to-model transformation that creates a Petri net from a process model (i.e. a view abstraction of the process model). The transformation is implemented using the Epsilon Transformation Language (ETL) [35], which is integrated in MetaDepth. The transformation uses the types of

the domain-specific meta-modelling language in Listing 4, and is applicable to models two meta-levels below, like the one in Listing 6. Rule `Gateway2Transition` transforms any gateway into a transition. If we apply this rule to the model in Listing 6, two transitions would be created: one corresponding to `a2d` and another for `d2c`, as both are indirect instances of `Gateway` (they are instances of `A2D` and `D2C` respectively, which are instances of `Seq`, and this is a subtype of `Gateway`). The rule iterates on all tasks which are output to the gateway (lines 14–15), creating a new Petri net arc which connects the created transition to the place created from the output task (line 16). This place is actually obtained through the standard ETL method `equivalent`. An invocation to this method implicitly executes the lazy rule `Task2Place` if the task has not been transformed yet. Rule `Task2Place` creates a place for each task, and adds one token to the place if the task is initial.

Both concepts and multi-level modelling can be used to build reusable operations. Both techniques share similarities, as the role of the binding in concept-based reuse is the same played by the typing in multi-level-based reuse. Moreover, a concept could be used with similar purposes to a domain-specific meta-modelling language, to define families of domain-specific languages. However, concepts (and their associated operations) can be developed independently of the meta-models to be bound, even *after* these meta-models exist, whereas domain-specific meta-modelling languages need to be defined *before* any meta-model of the family can be built. While a meta-model (like the one in Listing 5) can be bound to many concepts, it is typed by a single meta-modelling language (in this case, the one in Listing 4). It is up to future work to devise means to make multi-level modelling more flexible, allowing multiple, *a posteriori* typings.

6.3. Visualization

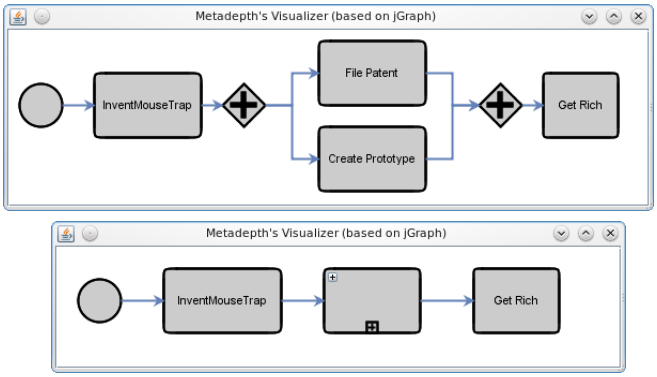

Figure 24: Example of a BPMN model, before and after applying an abstraction.

So far we have considered abstractions at the abstract syntax level. However, modelling languages have a concrete syntax as well, typically graphical. When an abstraction is applied, the model visualization should 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. The engine uses 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. Moreover, the engine uses a default visualization for the elements added by mixins, and which therefore do

not belong to the bound meta-models. Then, our engine interprets concrete syntax models and renders their visualization with the jGraph library. Figure 24 shows the visualization of a BPMN model, before and after applying the `block` abstraction.

7. Assessment: generality and reusability

This section assesses the generality of our approach by showing how it can be used to define existing well-known abstraction catalogues for model-based systems, business process models and Petri nets.

In [50], the author presents a catalogue of abstraction patterns for software architecture models. Some of these abstractions are specific to component systems and can be defined in our framework as domain-specific abstractions over a suitable concept for component systems, whereas others can be built using our horizontal abstractions. For example, the *"port group"* pattern – which merges all ports of a component into a single one – is our *source parallel* abstraction, while the *"black box"* pattern – which groups a set of connected components – is our *block* abstraction. The author also proposes some abstraction patterns for behavioural modelling languages, like state machines, which we can express using our horizontal abstractions. For instance, the *"group transition"* pattern – which merges states having transitions with the same trigger and leading to a common state – can be expressed with our *target parallel* abstraction, as Figure 9 shows for the *aggregation* variant. Interestingly, we propose the *block* horizontal abstraction to aggregate both state machines and activity diagrams. This confirms that it can be seen as a horizontal abstraction, applicable to several modelling languages. While the goal of [50] is to present abstraction patterns for specific modelling notations, ours is to provide a practical means to define reusable abstractions across different languages. Table 1 summarizes the proposed abstractions and how they can be represented using our catalogue. Most of them can be described using horizontal abstractions. Some others (like *platform layering* and *complex summary*) are too abstract and require the guidance of an expert user, e.g., to decide the layers to be included in an application, how they should communicate and which components to include in each one of them. This shows the generality of our horizontal abstractions.

Table 1: Software modelling abstractions proposed in [50].

Name	**Domain**	**Description**	**Abstraction in our catalogue**
Port group	Software arch.	merges all ports of a component into a single port	Source parallel
Black box	Software arch.	groups a set of connected components	Block
Black line	Software arch.	abstracts a set of components as an edge	Sequential
Cable pattern	Software arch.	abstracts a set of edges into a single edge	Parallel
Port group	Software arch.	abstracts a set of ports into a single port	Source parallel
Platform layering	Software arch.	separates an application in different layers	–
Summary message	Sequence diagrams	merges several messages into a single message	Domain-specific
Complex summary	Sequence diagrams	merges several life lines into a single life line	–
Summary state	State machines	merges a block of states into a single state	Block
Group transition	State machines	merges states having transitions with the same trigger and leading to a common state	Target parallel
Summary activity	Activity diagrams	merges a block of activities into a single activity	Block

Most abstraction techniques are designed for a particular notation. For example, there are techniques tailored for abstracting process models in BPMN [52]. A catalogue is presented in [53], which includes abstractions for event-driven process chains, like sequential, block, loop

and dead-end abstractions. Table 2 shows a summary of such abstractions. We can define some of them using our horizontal abstractions, whereas others were built using a specific concept for workflow languages. The fourth column of the table shows some bindings that we have made to concrete modelling languages, showing the reusability of our approach.

Table 2: Domain-specific abstractions for Business Process Models ([52, 53]).

Name	**Description**	**Abstraction in our catalogue**	**Binding**
Sequential	merges sequential elements between two blocks	Sequential Workflow sequence	BPMN Intalio UML activity diagrams
Block	abstracts elements between split/join connectors	Block Workflow block	BPMN Intalio UML activity diagrams
Loop	abstracts loops between two gateways	Workflow loop	BPMN Intalio UML activity diagrams
Dead-end	removes dead paths stemming from a gateway	Workflow remove path	BPMN Intalio UML activity diagrams
Semantic abstraction	abstraction based on similarity	Similarity + k-means	BPMN Intalio UML activity diagrams

Abstractions are also heavily used in verification to obtain simpler models that can be analysed more efficiently. For example, Table 3 shows the 6 reduction rules for Petri nets presented in [45], as well as how to define them using our framework in a generic, reusable way. This is an advantage of our approach, as in all these specific domains, abstraction implementations are frequently tied to the specificities of a language, and are therefore hardly reusable even for similar notations in the same domain.

Table 3: Domain-specific abstractions for Petri nets ([45]).

Name	**Description**	**Abstraction in our catalogue**	**Binding**
Fusion Series Places	merges two places and an intermediate transition	Sequential Petri net FSP	Petri nets DSLs
Fusion Series Transitions	merges two transitions and an intermediate place	Sequential Petri net FST	Petri nets DSLs
Fusion Parallel Places	merges parallel places between two transitions	Parallel Petri net FPP	Petri nets DSLs
Fusion Parallel Transitions	merges parallel transitions between two places	Parallel Petri net FPT	Petri nets DSLs
Elim. of Self-Loop Places	eliminates places with loops through a transition	Loop removal Petri net ELP	Petri nets DSLs
Elim. of Self-Loop Transitions	eliminates transitions with loops through a place	Loop removal Petri net ELT	Petri nets DSLs

8. Related work

Modelling languages provide abstractions, or primitives, that capture the *essential* characteristics of systems, dismissing the unimportant parts for the purpose of the modeller. Since

models can be very complex, abstractions become important as a means to understand, modify and manage this complexity. 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 [44], and is pervasive in software engineering [29]. However, to the best of our knowledge, there are no works aimed at systematizing and providing a practical framework to reuse abstractions across modelling languages, but there are only concrete proposals for specific modelling notations and tools.

For instance, many modelling languages (e.g. Statecharts [22], BPMN [46] or UML component diagrams [47]) have particular hierarchical primitives that enable element aggregation, which can be used for model abstraction. The foundations of such constructs can be traced back to Harel's higraphs [23], which enrich "flat" hyper graphs with depth (hierarchical primitives) and cartesian products. Higraphs can be used as a basis to build notations like hierarchical entity relationship diagrams or Statecharts [22]. There is also the possibility of "zooming out" a certain node, supressing low-level details. While this operation can be seen as a tool visualization feature, in this work we concentrate on abstraction operations having an impact on the model structure.

In most cases, abstractions have been proposed for specific notations or tools, like BPMN [52], Statecharts [54] or Petri nets [45]. If the modelling language contains hierarchical primitives (like BPMN or Statecharts), then abstractions may introduce aggregation elements (e.g. organizing activities in subprocesses). If the language lacks such primitives (like Petri nets), abstractions are normally restricted to merge or delete (filter) elements. Abstractions have been proposed not only for behavioural notations – like the previous ones – but for structural notations as well, like Entity Relationship Diagrams [43, 51], components [50] or class diagrams [11]. In some cases, like in [51], the language (Entity Relationships) is increased with hierarchical constructs that aggregate related elements. Language extensions are considered by our mixins. In any case, the proposed abstractions in the literature are normally tied to the specificities of a language or tool, and cannot be reused for other similar languages.

Some abstractions can be considered model refactorings [13], as they preserve the model semantics. Many refactorings for different modelling languages have been proposed, some of which are abstractions. For example, in [54], the authors identify the "fold outgoing transition", which is the target parallel abstraction shown in Figure 9. As in the previous case, such refactorings are normally tied to specific notations or tools, and cannot be reused for other similar languages.

In the area of software modelling and MDE, two kinds of models are distinguished in [37]: 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 the systematic abstraction of token models.

Related to model abstraction, 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 [39]. This technique can be seen as a particular case of our view abstraction, when the obtained view conforms to the original meta-model. In [3], 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 [42]. 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, [57] 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 [57] 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.

While there are works that use clustering techniques like FCA and RCA in MDE [1, 27], their main purpose is to refactor class diagrams or meta-models in order to find a good inheritance hierarchy of classes. Usually, the reason is that FCA needs boolean attributes, which in class diagrams can be emulated by checking if classes define the same attribute (name and data type) or not. There are few works aiming at generic, reusable implementations of clustering techniques for MDE. In [12], the authors propose a generic mechanism (similar to our annotation models) to use RCA/FCA for restructuring the inheritance hierarchy of meta-models. However, they do not consider applying FCA to models for other purposes, and lack the possibility to define equivalence classes and scaling operations for attribute values.

Abstraction has also been studied theoretically. Hobbs [26] suggests that, in the course of reasoning, we conceptualize the world at different levels of granularity. He defines the notion of indistinguishability that allows mapping 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 [16], 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 [17], e.g. to ease automated deduction. Keet [32] uses abstraction to help the comprehension of ontologies. In [41], granularity abstraction is applied to natural language processing by representing natural language as logical forms that are mapped to coarse-grained forms to enable sentence analysis. Kascheck [31] 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 [24, 25]. For example, in [25], 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 [14], 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 [36]. 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 a rich set of mechanisms for the development of reusable abstractions for modelling languages, in the context of MDE.

9. Conclusions and future work

In this paper, we have presented a set of techniques to define reusable abstractions, applicable to several modelling languages in a meta-model independent way. Our abstractions are generic operations defined over minimal concepts, to be bound to the meta-models of the modelling languages. We support two types of concepts: *structural*, to be used when the concept and the bound meta-models are structurally similar, and *hybrid*, which allows solving heterogeneities between them in a flexible way. In addition, abstractions can be further configured by overriding default hook methods for particular modelling notations. Abstraction operations can use generic clustering techniques, which are configured through *annotation models*. We have presented two such generic clustering techniques, which can be used as similarity criteria for abstractions of arbitrary modelling languages. If the target modelling language lacks abstraction elements (e.g. aggregate objects), they can be added through the application of *mixins*. Finally, we can also obtain reusability for families of modelling languages by defining the abstractions over suitable domain-specific meta-modelling languages.

We have presented a catalogue of horizontal abstractions, as well as domain-specific abstractions for Petri net like languages, process modelling and workflow languages. The approach is implemented in the MetaDepth tool, which provides support for concept-based and multi-level based genericity, as well as for the visualization of aggregate objects. To the best of our knowledge, this is the first time that a systematic and generic support for reusable abstractions for modelling language is proposed in the literature.

Altogether, our approach makes easier the incorporation of abstraction mechanisms to new or existing modelling languages, in a non-intrusive way. Abstractions are defined once over a suitable concept, and then can be reused by different meta-models, so that the effort to reuse one of such abstractions is smaller than that of developing the abstraction from scratch. Moreover, concepts provide a suitable, minimal interface to the abstraction operation, so that the developer does not need to confront the complexity of the reused code, but he only needs to understand the concept behind.

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 clustering techniques, and to include mechanism for the detection of non-confluent abstraction applications. We are also working in allowing multiple *a-posteriori* typing in multi-level modelling, permitting a more flexible multi-level-based reusability.

Acknowledgements. Work funded by the Spanish Ministry of Economy and Competitivity with project "Go Lite" (TIN2011-24139), and the R&D programme of Madrid Region with project "eMadrid" (S2009/TIC-1650).

References

[1] G. Arévalo, J.-R. Falleri, M. Huchard, and C. Nebut. Building abstractions in class models: Formal concept analysis in a model-driven approach. In *MoDELS*, volume 4199 of *LNCS*, pages 513–527. Springer, 2006.

[2] C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.

[3] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Modeling model slicers. In *MoDELS'11*, volume 6981 of *LNCS*, pages 62–76. Springer, 2011.

[4] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[5] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order (2. ed.)*. Cambridge University Press, 2002.

[6] 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`.
[7] J. de Lara and E. Guerra. From types to type requirements: Genericity for model-driven engineering. *Software and Systems Modeling*, page (in press), 2011.
[8] J. de Lara and E. Guerra. Domain-specific textual meta-modelling languages for model driven engineering. In *ECMFA12*, volume 7349 of *LNCS*, pages 259–274. Springer, 2012.
[9] J. de Lara, E. Guerra, and J. S. Cuadrado. Abstracting modelling languages: A reutilization approach. In *CAiSE'12*, volume 7328 of *LNCS*, pages 127–143. Springer, 2012.
[10] J. de Lara and H. Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *J. Vis. Lang. Comput.*, 15(3-4):309–330, 2004.
[11] A. Egyed. Semantic abstraction rules for class diagrams. In *ASE*, pages 301–304, 2000.
[12] J.-R. Falleri, G. Arévalo, M. Huchard, and C. Nebut. Use of model driven engineering in building generic FCA/RCA tools. In *CLA*, volume 331 of *CEUR Workshop Proceedings*, 2007.
[13] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.
[14] M. Frisch, R. Dachselt, and T. Brückmann. Towards seamless semantic zooming techniques for UML diagrams. In *SOFTVIS*, pages 207–208. ACM, 2008.
[15] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
[16] C. Ghidini and F. Giunchiglia. A semantics for abstraction. In *ECAI*, pages 343–347. IOS Press, 2004.
[17] F. Giunchiglia and T. Walsh. A theory of abstraction. *Artif. Intell.*, 57(2-3):323–389, 1992.
[18] C. Gonzalez-Perez and B. Henderson-Sellers. A powertype-based metamodelling framework. *Software and Systems Modeling*, 5(1):72–90, 2006.
[19] E. Guerra, J. de Lara, and P. Díaz. Visual specification of measurements and redesigns for domain specific visual languages. *J. Vis. Lang. Comput.*, 19(3):399–425, 2008.
[20] E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, and O. M. dos Santos. *trans*ML: A family of languages to model model transformations. In *MoDELS'10*, volume 6394 of *LNCS*, pages 106–120. Springer, 2010.
[21] E. Guerra, J. de Lara, A. Malizia, and P. Díaz. Supporting user-oriented analysis for multi-view domain-specific visual languages. *Information & Software Technology*, 51(4):769–784, 2009.
[22] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.
[23] D. Harel. On visual formalisms. *Commun. ACM*, 31(5):514–530, 1988.
[24] B. Henderson-Sellers. Random thoughts on multi-level conceptual modelling. In *The Evolution of Conceptual Modeling*, volume 6520 of *LNCS*, pages 93–116. Springer, 2011.
[25] 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.
[26] J. Hobbs. Granularity. In *IJCAI'85*, pages 432–435. M. Kaufmann, 1985.
[27] M. Huchard, M. R. Hacene, C. Roume, and P. Valtchev. Relational concept discovery in structured datasets. *Annals of Mathematics and Artificial Intelligence*, 49(1-4):39–76, Apr. 2007.
[28] S. Jablonski, B. Volz, and S. Dornstauder. A meta modeling framework for domain specific process management. In *COMPSAC'08*, pages 1011 –1016. IEEE Computer Society, 2008.
[29] M. Jackson. Aspects of abstraction in software development. *Software and Systems Modeling*, 11(4):495–511, 2012.
[30] E. Juan, J. Tsai, T. Murata, and Y. Zhou. Reduction methods for real-time systems using delay time Petri nets. *IEEE Transactions on Software Engineering*, 27(5):422 –448, 2001.
[31] R. Kaschek. A little theory of abstraction. In *Modellierung*, volume 45 of *LNI*, pages 75–92. GI, 2004.
[32] C. Keet. Enhancing comprehension of ontologies and conceptual models through abstractions. In *AI*IA*, volume 4733 of *LNCS*, pages 813–821. Springer, 2007.
[33] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE CS, 2008.
[34] 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.
[35] D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Transformation Language. In *ICMT'08*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.
[36] O. Köth and M. Minas. Structure, abstraction, and direct manipulation in diagram editors. In *Diagrams*, volume 2317 of *LNCS*, pages 290–304. Springer, 2002.
[37] T. Kühne. Matters of (meta-)modeling. *Software and Systems Modeling*, 5(4):369–385, 2006.
[38] T. Kühne and D. Schreiber. Can programming be liberated from the two-level style: multi-level programming with deepJava. In *OOPSLA*, pages 229–244. ACM, 2007.
[39] K. Lano and S. Kolahdouz. Slicing techniques for UML models. *Journal of Object Technology*, 10:11:1–49, 2011.
[40] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. University of California Press, 1967.

[41] 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.
[42] N. Moha, V. Mahé, O. Barais, and J. Jézéquel. Generic model refactorings. In *MODELS'09*, volume 5795 of *LNCS*, pages 628–643. Springer, 2009.
[43] D. L. Moody and A. Flitman. A methodology for clustering entity relationship models - a human information processing approach. In *ER*, volume 1728 of *LNCS*, pages 114–130. Springer, 1999.
[44] P. Mosterman and H. Vangheluwe. Computer automated multi-paradigm modeling: An introduction. *Simulation*, 80(9):433–450, 2004.
[45] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.
[46] OMG. OMG's BPMN home page. `http://www.bpmn.org/`.
[47] OMG. UML 2.3 specification. `http://www.omg.org/spec/UML/2.3/`.
[48] A. Polyvyanyy, S. Smirnov, and M. Weske. The triconnected abstraction of process models. In *BPM*, volume 5701 of *LNCS*, pages 229–244, 2009.
[49] S. E. Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007.
[50] B. Selic. A short catalogue of abstraction patterns for model-based software engineering. *Int. J. Software and Informatics*, 5(1-2):313–334, 2011.
[51] P. Shoval, R. Danoch, and M. Balaban. Hierarchical entity-relationship diagrams: the model, method of creation and experimental evaluation. *Requir. Eng.*, 9(4):217–228, 2004.
[52] S. Smirnov, H. Reijers, and M. Weske. A semantic approach for business process model abstraction. In *CAiSE'11*, volume 6741 of *LNCS*, pages 497–511, 2011.
[53] S. Smirnov, H. A. Reijers, M. Weske, and T. Nugteren. Business process model abstraction: a definition, catalog, and survey. *Distributed and Parallel Databases*, 30(1):63–99, 2012.
[54] G. Sunyé, D. Pollet, Y. L. Traon, and J.-M. Jézéquel. Refactoring uml models. In *UML*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001.
[55] W. M. P. van der Aalst. Formalization and verification of event-driven process chains. *Information & Software Technology*, 41(10):639–650, 1999.
[56] M. Völter and T. Stahl. *Model-driven software development*. Wiley, 2006.
[57] 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.
[58] M. Wynn, H. Verbeek, W. van der Aalst, A. ter Hofstede, and D. Edmond. Reduction rules for YAWL workflows with cancellation regions and or-joins. *Inf. & Softw. Techn.*, 51(6):1010–1020, 2009.