A Component Model for Model Transformations

Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara

Abstract—Model-Driven Engineering promotes an active use of models to conduct the software development process. In this way, models are used to specify, simulate, verify, test and generate code for the final systems. Model transformations are key enablers for this approach, being used to manipulate instance models of a certain modelling language. However, while other development paradigms make available techniques to increase productivity through reutilization, there are few proposals for the reuse of model transformations across *different* modelling languages. As a result, transformations have to be developed from scratch even if other similar ones exist.

In this paper, we propose a technique for the flexible reutilization of model transformations. Our proposal is based on generic programming for the definition and instantiation of transformation templates, and on component-based development for the encapsulation and composition of transformations. We have designed a component model for model transformations, supported by an implementation currently targeting the Atlas Transformation Language (ATL). To evaluate its reusability potential, we report on a generic transformation component to analyse workflow models through their transformation into Petri nets, which we have reused for eight workflow languages, including UML Activity Diagrams, YAWL and two versions of BPMN.

Index Terms—Model-Driven Engineering, Model Transformation, Reusability, Genericity, Component-Based Development.

1 Introduction

Model transformations are programs that take one or more models as input, and produce a number of output models. The aim of transformations is automating model manipulation when possible, while reducing the number of errors in this manipulation. This technology is key in Model-Driven Engineering (MDE) [1], where it is used to implement model refactorings, model refinements, model synchronization mechanisms, and translators of models into other formalisms for analysis, among other tasks.

The increasing adoption of MDE is leading to the construction of model transformations of raising complexity. However, building new transformations from scratch is costly, error prone, and requires specialized skills. Hence, transformation developers would benefit from mechanisms enabling the construction of new transformations by reusing proven, existing ones, adapted to the particular problem to be solved.

In current MDE practice, there is a proliferation of meta-model variants for the same languages. This is partially caused by the focus of MDE on domain-specific languages (DSLs), and due to simplifications or variations introduced in large meta-models (like UML class diagrams and BPMN-like process modelling languages) to make them fit for the project purpose. For instance, the ATL meta-model zoo[1] includes 11 different meta-models for Petri nets, 14 meta-models describing conference organization systems, and 6 variations of the Java meta-model. This variety hampers reuse because transformations are developed for a particular meta-model and cannot be reused for other related ones. Appropriate mechanisms for transformation reutilization would alleviate this problem, and we claim that they are essential for the success of the MDE paradigm at industrial scale.

Kusel at el. [2] identify the barriers that hinder the reuse of transformations, including: (1) transformations are tight to concrete meta-models, and it is not possible to reuse them for semantically related meta-models, (2) lack of meta-information and missing repositories, which makes it difficult to search and find transformations, (3) challenging and limited specialization of existing transformations, which limits the adaptation of a transformation to unforeseen contexts, and (4) insufficient integration support. Thus, the most used approach in MDE is code scavenging. Developers typically search for related transformations, pick some rules and adapt them to the meta-models at hand. Integration is usually done ad-hoc by creating a build script (e.g., with ANT) that configures the transformation within a transformation chain for its execution over concrete models and meta-models.

Component-Based Software Development (CBSD) advocates the construction of systems by systematically reusing and adapting pre-built software units or *components*. The expected benefits are increased reuse, reduced production cost and shorter time to market. As noted in [3], any CBSD methodology has an underlying *component model* defining "*what components are,*

• *J. Sánchez Cuadrado, E. Guerra and J. de Lara are with the Department of Computer Science, Universidad Autónoma de Madrid, Spain. E-mail: {jesus.sanchez.cuadrado, esther.guerra, juan.delara}@uam.es*

1. http://www.emn.fr/z-info/atlanmod/index.php/Ecore

how they are constructed, how they can be composed and how they can be deployed". While component models have been defined for several application domains [4], [5], there are no appropriate proposals for MDE. A component model for model transformations requires a mechanism for the adaptation of transformations to different contexts (i.e., use of different meta-models), as well as composition and integration mechanisms to allow creating transformation components that are truly reusable by third-parties, who may create new components as composites of existing ones.

In previous works [6], [7], we applied ideas from generic programming to the definition of generic, reusable model transformation templates. Transformation templates are not defined over concrete meta-models, but over *concepts* [8], [9], [10] which gather the requirements that a meta-model should fulfil to yield a proper template instance. A concept is similar to a meta-model, but its elements (classes, references, attributes) are interpreted as variables that need to be *bound* to elements of specific meta-models. The binding induces a retyping and adaptation of the transformation template, so that it becomes applicable to the instances of the bound meta-models. As concepts can be bound to several meta-models, they become reusable for all of them.

A concept can be seen as a pivot meta-model [11]. In general, to reuse a transformation defined over a pivot with a new meta-model, one must create a transformation from the meta-model to the pivot, and then chain both transformations. Instead, we provide an alternative with three main advantages: (1) there is no need to generate an intermediate instance model of the pivot, but the template gets adapted to the new meta-model, being more efficient in terms of performance and memory footprint, (2) traceability between the source and target models of the transformation is automatic because there is no intermediate step, and (3) there is support to bind meta-models and concepts, which is normally simpler than using a full-fledged transformation language. Thus, a key element in our approach to be practical is the expressiveness of the binding, for which we use a dedicated DSL.

In this paper, we improve our binding DSL to consider more complex adaptations, and propose a component model for model transformations based on the notion of generic transformation. The model supports both simple and composite components, which are treated in a unified way by using concepts as their interfaces. *Simple* components encapsulate a transformation template and expose one or more concepts specifying the requirements that meta-models need to fulfil to apply the component. Components may also expose features [12], which can be used to configure the behaviour of the transformation template, or to select between different execution paths in a composite component. *Composite* components allow instantiation and integration of existing components, so that from the reuse perspective, there is no difference between simple and composite components.

We frame our proposal into the four dimensions for software reuse proposed by Krueger [13]: abstraction, selection, specialization and integration. Driven by these dimensions, and with the aim of addressing the shortcomings identified in [2], we discuss how concepts are a suitable abstraction to define the interface between transformation components, we propose an approach to find and select components, a mechanism to adapt components to particular meta-models, and we show how to integrate components to perform complex transformation tasks.

We have implemented our component model in a tool, called Bentō, contributed as open source at http://www.miso.es/tools/bento.html. We have used Bentō to evaluate our proposal by building a generic component to analyse workflow models by their transformation into Petri nets, being able to apply this component to eight out of nine relevant modelling languages. Additionally, we have studied the suitability of concepts as abstraction mechanism, finding that they tend to be simpler and easier to comprehend than full-fledged meta-models.

Paper organization. Section 2 introduces model transformations, motivating the need for reuse mechanisms. Section 3 overviews our component model, which is detailed in the following sections: Section 4 proposes concepts as a way to abstract transformations, Section 5 explains how to specialize transformations for different meta-models, Section 6 presents mechanisms to facilitate component selection, and Section 7 discusses their composition mechanisms. The different sections are illustrated using a component to transform workflow languages into Petri nets for analysis. In Section 8, we evaluate our proposal with respect to its reusability potential, and the abstraction power of concepts. Finally, Section 9 compares with related research and Section 10 presents the conclusions and future work. Two appendices available in the online supplemental material describe the rules to adapt an ATL template according to a binding, and contain the complete running example.

2 Model Transformation

MDE proposes an active use of models in the different phases of the software development. The allowed structure of models is normally described through meta-models, which are models describing the abstract syntax of the modelling language. Hence, meta-models define the allowed classes, relations and attributes that can be instantiated in models.

In MDE, model manipulation is automated through model transformations [14]. The kind of transformations we tackle in this paper are model-to-model, which transform a source model conforming to a source meta-model, into a target model conforming

to another meta-model. Alternatively, transformations can also be used to manipulate models *in-place*.

Several model-to-model transformation languages exist. In this paper, we use the Atlas Transformation Language (ATL) [15] as it is one of the most widely used in practice. ATL transformations are made of rules. Each rule defines the way in which some configuration of objects in the source model should be used to produce certain configuration of objects in the target model. Moreover, the attributes and references of the objects created in the target model are initialized using OCL expressions [16].

As an example, Listing 1 shows part of an ATL transformation from BPMN models into Petri nets. The goal is being able to analyse some properties of the input BPMN models, like the absence of deadlocks or incorrect executions due to unfinished paths in parallel splits, by using some available Petri net tool. The source meta-model of the transformation is the Intalio's implementation of the BPMN meta-model[2], partially shown in Figure 1. Thus, the transformation is only expected to work with models that are instances of this meta-model. The target meta-model for Petri nets, shown in Figure 2, is hand-made.

```
1  module intalio2pn;
2  create OUT : PN from IN : BPMN;
3
4  -- the start event is transformed into a place with 1 token
5  rule start_event {
6    from activity : BPMN!Activity ( activity.activityType = #EventStartEmpty )
7    to place : PN!Place (
8      name <- 'Initial',
9      tokens <- 1 )
10 }
11
12 -- each task is transformed into a place without tokens
13 rule task {
14   from activity : BPMN!Activity ( activity.activityType = #Task )
15   to place : PN!Place (
16     name <- activity.name,
17     tokens <- 0 )
18 }
19
20 -- edges connecting 2 activities that have been converted into places,
21 -- are transformed into transitions
22 rule place_place {
23   from edge : BPMN!SequenceEdge (
24     edge.source.toPlaceAsOutput and
25     edge.target.toPlaceAsInput )
26   to transition : PN!Transition (
27     in <- edge.source,
28     out <- edge.target )
29 }
```

Listing 1. Excerpt of an ATL transformation from Intalio's BPMN meta-model to Petri nets.

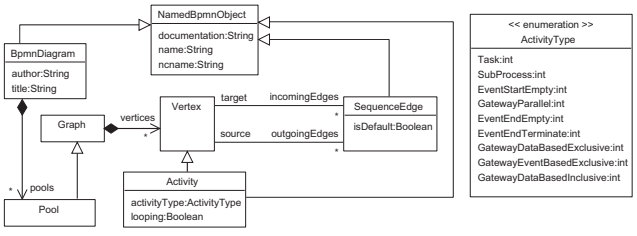


Fig. 1. Excerpt of Intalio's BPMN meta-model.

2. See http://www.intalio.com/products/bpms.

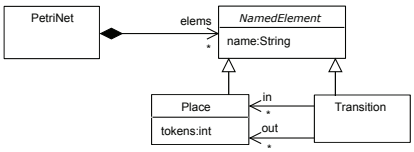


Fig. 2. Petri nets meta-model.

The listing shows just three rules. The first two map activities to places, marking the place with a token if the activity is initial, which is given by the value of the activityType attribute. The third rule creates a Petri net transition for each BPMN edge connecting two activities. Its filter uses two OCL helpers (toPlaceAsOutput and toPlaceAsInput) to check that the source and target activities that the edge connects correspond to places (as some activity types, like parallel gateways, are transformed into transitions and need to be connected in a different way by other rules). The complete transformation is significantly more complex, since it needs to deal with challenging mappings such as gateways that may take more than one path, thus requiring deep knowledge of workflow and Petri net languages and advanced ATL developing skills.

Hence, an issue is that this transformation is tied to the Intalio's BPMN meta-model, and cannot be used with other BPMN meta-models, like the OMG standard BPMN2.0 meta-model [17]. Should we want to transform other workflow-like languages like UML Activity Diagrams or Grafcet [18] diagrams into Petri nets, we would need to develop similar transformations, but we cannot reuse the one in Listing 1 because it is defined over the types in the Intalio's meta-model. The only current option is to adapt the transformation manually to the new context, which is error prone because the developer needs to understand all details of the original transformation to rewrite it.

In this scenario, it would be useful to have a reusable component which transforms similar workflow languages into Petri nets. The next section describes an approach to the creation of such generic transformation components.

3 A Component Model for Model Transformations

In CBSD, systems are built by reusing components and composing them. We apply this view to MDE by contributing a component model for model transformations. Our model satisfies the features expected by any CBSD approach [3]: components are pre-existing reusable units that can be used quicker than writing code from scratch, they are instantiable in different contexts to maximize reuse, they are composable, and different parties can produce and use them.

In the simplest setting, our transformation components contain a reusable transformation template, and its expected input and output models are exposed

via typed ports. The type of ports is described either by a meta-model or by a *concept*. A concept is a description of the structural requirements that a meta-model needs to fulfil to allow the instantiation of the component with the meta-model. Thus, a component can be used with any meta-model that fulfils the requirements of its associated concepts. Concepts resemble meta-models, and transformation templates are standard transformations defined over the types declared in the concepts. For the moment, we will consider transformation components as black boxes with well-defined interfaces given by concepts.

Figure 3(a) shows the graphical representation of a reusable component to transform workflow languages (like BPMN and UML Activity Diagrams) into Petri nets. The input model of the component must be compatible with the FD concept, which includes variables representing workflow elements such as task, gateway and edge (not shown in the figure). The component generates a Petri net that is described by the PN concept. We use the "lollipop" notation and decorate input ports with required interfaces (e.g., FD), and output ports with provided ones (e.g., PN).

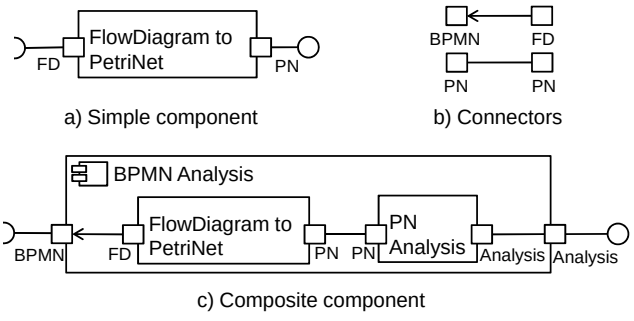


Fig. 3. Transformation component example.

The context of use of a component is defined by means of connectors that wire its exposed ports with the ports of other components or with concrete meta-models. In the latter case, the connector is realized by binding each element in the concept to some element in the meta-model. This induces an adaptation of the transformation template inside the component, becoming applicable to the model instances of the bound meta-model. Figure 3(b) shows the graphical notation used to specify a connector binding the FD concept to the BPMN meta-model. If two ports declare the same meta-model or concept, no binding is needed, and their connector is depicted by a line.

A transformation component can be directly used to transform models, or it can be combined with other components to build composite ones. Figure 3(c) shows a composite component to analyse BPMN models. It includes the FlowDiagram to PetriNet component, specialized for BPMN by binding the FD concept to the BPMN meta-model, and chained with the PN Analysis component, which analyses the Petri net that results from the transformation. The resulting BPMN Analysis component can be integrated in further chains.

Our component model is described by the meta-model excerpt of Figure 4. A Component is an abstract entity that performs an operation on a set of source and target models. The component interface is given via ports that declare the Meta-Model of the manipulated models. A Concept is a special kind of meta-model that admits *bindings* to meta-models. A *simple* component (TransformationComponent) encapsulates a transformation template, created using some transformation language. While we support ATL and Java templates, adaptation through bindings is only available for ATL. A CompositeComponent is made of several components, simple or composite, interconnected using a composition language (subclasses of CompositionStep, omitted in the figure).

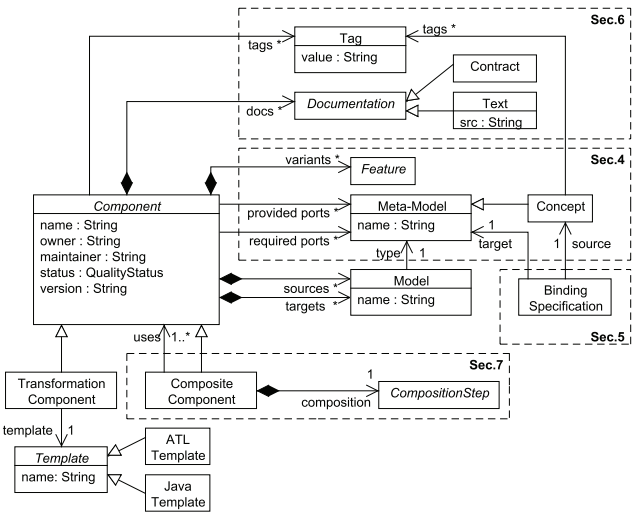


Fig. 4. Excerpt of the meta-model representing the component model for model transformations.

We have framed this component model into the four dimensions of reuse proposed by Krueger [13]: abstraction, specialization, selection and integration. We use them to present the details of our model in the rest of the paper, but first, we briefly introduce them with respect to the meta-model presented above.

In our approach, *concepts* are the *abstraction* mechanism, becoming the *reuse* interface of components. Thus, components are black-boxes, as users only need to understand the attached concepts to reuse a component. Components may be configured via features (class Feature) to select between different behaviours. Concepts and features are presented in Section 4.

Our *specialization* mechanism is the binding, which is a mapping between a concept and a meta-model. It is specified through a dedicated language (class BindingSpecification) that bridges some structural heterogeneities between the concept and the meta-model. The binding induces an adaptation of the transformation template to make it compatible with the bound

meta-model. Bindings are presented in Section 5.

To enable *selection* of components, both concepts and transformations may attach Tags. Moreover, transformations can be described textually or more formally through a *contract* using the PAMOMO transformation specification language [19] (class Contract). These mechanisms are explained in Section 6.

Finally, we provide a composition language that allows the *integration* of different transformations components to build a transformation chain (class CompositionStep). This language is presented in Section 7.

As a running example, we use our component model to make the transformation presented in Section 2 reusable. Our goal is to create a reusable component for the Petri net-based analysis of workflow languages, as several researchers have proposed transformations from their particular workflow languages into Petri nets [20]. In order to illustrate composite components (Section 7) we use a generic Petri net analysis component and two graph layout components, which are used to build a complete analysis component for workflow languages. The rest of the paper is presented with relevant excerpts of this component, while the complete example is available online, in the supplemental material.

4 Abstraction Mechanisms for Reusable Transformations

Abstraction is the primary mechanism of reuse, as an abstraction ultimately describes a related collection of reusable entities of some nature [21]. Abstractions provide a higher-level view of an artefact, suppressing low-level details. Hence, they have a visible part, either fixed or variable, and a hidden part [13].

Our components hide the implementation details of a transformation, and make visible a concept, configuration parameters and a description. While the concept and parameters are variable parts of the component abstraction, the description is fixed. Thus, concepts abstract a family of languages, and allow the specialization of a transformation template for a language of the family by means of a binding (see Section 5). In addition, components can have parameters to configure certain aspects. These two abstraction mechanisms are explained in the next subsections.

4.1 Transformation concepts

A transformation concept gathers the requirements that a meta-model should fulfil to serve as source or target of a transformation template. In our approach, concepts have the form of a meta-model; however, their elements (classes, attributes, references) are variables that need to be bound to elements of a specific meta-model. Additionally, concept classes can be annotated with a cardinality [6], which is an interval specifying the minimum and maximum number of bindings that can be provided for the class. The default cardinality is [1..1], meaning that the concept class needs to be bound to exactly one meta-model class. A lower bound 0 indicates that the concept class can be left unbound, that is, it is not mandatory and may have no correspondence in a specific meta-model. The upper limit * means there is no upper bound.

In the running example, the source concept must abstract the common elements found in workflow languages that the transformation needs to deal with. Figure 5 shows the designed concept. Workflows are made of interconnected nodes, some of them representing tasks or activities, while others are gateways in charge of routing the process flow. Tasks and the different types of gateways are modelled as subclasses of an abstract class Node. We include as gateways the homonymous patterns in the well-known catalogue of control workflow patterns of van der Aalst et al. [22], as the semantics of these patterns is well-documented and precisely defined. All types of gateway have cardinality 0..1 because the pattern they represent may be present or not in a particular modelling language.

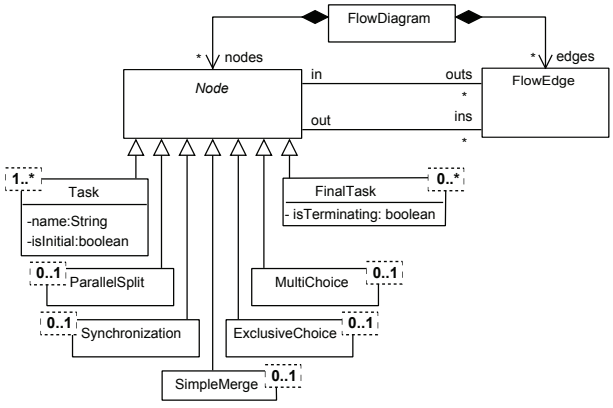


Fig. 5. A concept for workflow languages

The semantics of final tasks depends on the particular modelling language. While in languages like YAWL, reaching a final task concludes the execution of the workflow, in BPMN the execution only finishes when all active branches terminate. There are also languages which accommodate both semantics. For example, Activity Diagrams have both FlowFinalNodes for ending just a certain execution branch, and ActivityFinalNodes which end the complete execution. Our concept abstracts both options by means of the class FinalTask, which owns the flag isTerminating to indicate whether reaching an object of the bound type concludes the workflow execution. The cardinality 0..* allows class FinalTask to be bound to several meta-model classes, enabling final tasks with different semantics.

Hence, a concept provides a canonical representation for the features of a family of meta-models (e.g., the different ways to represent a final task in workflow languages), and the transformation template is written for this common representation using

the regular constructs of the transformation language. The heterogeneities between the concept and each particular meta-model will be bridged by establishing a binding between them (see Section 5).

While concepts are syntactically similar to plain meta-models (except for the cardinality on classes), conceptually, there is an important difference. While a concept is the interface of a transformation template and contains only the elements accessed by the template, meta-models reflect the complexity of some domain and may contain many elements irrelevant for the particular transformation. For instance, the meta-model of OMG's BPMN defines more than 130 classes, while our workflow concept has 10. Thus, concepts are usually simpler than domain meta-models, and therefore transformations over concepts tend to be simpler as well, as they do not contain *accidental* details of the domain. An evaluation of the abstraction power of concepts is provided in Section 8.2.

4.2 Transformation variants

The behaviour of some transformations may depend on features that are tacit knowledge of the source domain. For example, initial tasks are mandatory in languages like YAWL, but they are optional in others like BPMN. The same happens with final tasks. These different options result in different translations into Petri nets. While we need to be able to choose the right option for the concrete input meta-model, such options do not naturally fit in a concept because they are not structural properties of single objects, but they belong to the language semantics. Thus, we model those variants as a feature model [12] attached to the component, as the meta-model in Figure 6 shows.

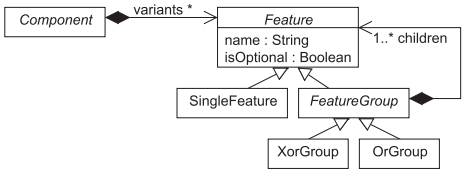


Fig. 6. Slice of the component meta-model of Figure 4: describing transformation variants.

As an example, Figure 7 shows a transformation component exposing a feature model with two options for initial and final states: *optional* and *mandatory*.

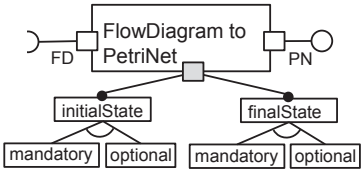


Fig. 7. Component with features.

When executing the component, the transformation template can inspect the values of the selected features through a model that is dynamically generated. A generic library is provided to facilitate looking up feature values in the transformation template.

5 Specialization

A transformation component must be specialized or adapted to its context of use. In our case, the specialization consists in making the constituent transformation template applicable to an existing meta-model. This is achieved by specifying a binding between the interface of the component (i.e., the concept) and the meta-model, in order to identify the meta-model elements that play the different "roles" required in the concept. This binding induces an adapted component that is applicable to the instances of the bound meta-model. Figure 8 shows this working schema.

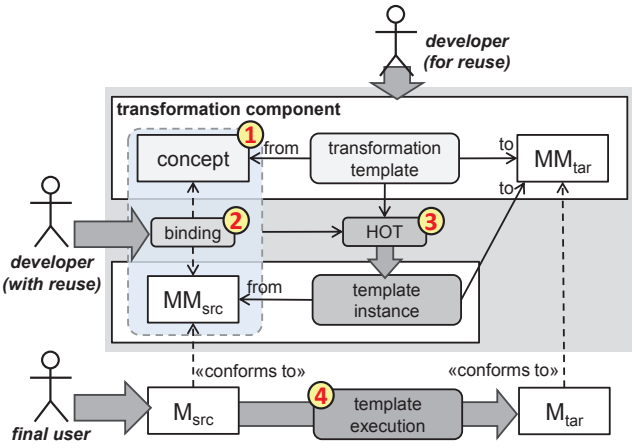


Fig. 8. Transformation adaptation through binding.

Thus, a transformation template is defined over a concept (label 1 in the figure). When this concept is bound to a specific meta-model MM_{src} (label 2), the *binding* induces a rewriting of the template, so that a transformation specialized for the specific meta-model is obtained. This rewriting is performed by a high-order transformation (HOT) (label 3). In this way, the transformation template becomes reusable as it can be bound to different meta-models and therefore used in different contexts. The figure shows a binding of the source concept of the transformation template, while the target is a fixed meta-model. Having a concept in the target is also possible, though the binding of target concepts is more restricted than those in the source (see [7]). For simplicity, in the following we assume a binding for the source and a fixed target.

5.1 Binding concepts to meta-models

In the simplest case, the binding is a direct mapping from the elements in the concept to elements in a meta-model, which induces a type renaming in the transformation template. This is enough for meta-models that contain a part structurally equal to the

concept; however, this is uncommon. Thus, to enable the reuse of a transformation component with a wider range of meta-models, we provide a flexible binding mechanism that is able to fix some structural heterogeneities between a concept and a meta-model.

To facilitate the definition of such bindings, we have designed a DSL that improves the one presented in [6], [7] by supporting more complex bindings (Section 5.2) and binding inference (Section 5.3). Listing 2 shows an example of its usage to define part of the binding from the Workflow concept (Figure 5) to the Intalio's BPMN meta-model (Figure 1). Lines 2–3 refer to the files with the definition of the concept and the bound meta-model. Lines 5–7 define three bindings for classes. The first one maps the concept class FlowDiagram to the meta-model class BpmnDiagram. Lines 9–12 show a class binding that includes a when OCL expression to select only those BPMN Activity instances satisfying the expression. Lines 14–15 bind the feature FlowDiagram.nodes to an OCL expression. The context of the expression (self) is BpmnDiagram, as this is the meta-model class to which FlowDiagram is bound to. The result of evaluating the expression is a collection of Vertex, which is compatible with the type Node expected by the collection, as Node is bound to Vertex (line 6). Line 17 defines a renaming feature binding, and line 18 a feature binding to an OCL expression stating that an Intalio task is initial when its attribute activityType equals the enumerate literal EventStartEmpty. The rest of the binding specification is similar. Notably, the same pattern is repeated in the class bindings of gateways, in which we need to use OCL expressions to select Activity instances according to their activityType, as well as their incoming and outgoing edges.

```
1  binding intalio2fd {
2    concept FD : "FlowDiagram.ecore"
3    metamodel BPMN : "IntalioBPMN.ecore"
4
5    class FlowDiagram to BpmnDiagram
6    class Node to Vertex
7    class FlowEdge to SequenceEdge
8
9    class ParallelSplit to Activity when
10       self.activityType = #GatewayParallel and
11       self.incomingEdges->size() = 1 and
12       self.outgoingEdges->size() > 1
13
14   feature FlowDiagram.nodes =
15       self.pools->collect(p | p.vertices)->flatten()
16
17   feature Node.outs is outgoingEdges
18   feature Task.isInitial = self.activityType = #EventStartEmpty
19     ...
20 }
```

Listing 2. Binding for Intalio's BPMN meta-model.

The primitives of our DSL for bindings are described by the meta-model shown in Figure 9. We support three kinds of bindings (subclasses of Binding) which allow mapping classes, "virtual classes" defined inline as part of the binding, and features.

First, a ClassBinding maps a class in the concept to zero or more classes in the meta-model, according to its declared cardinality. Mapping several classes in

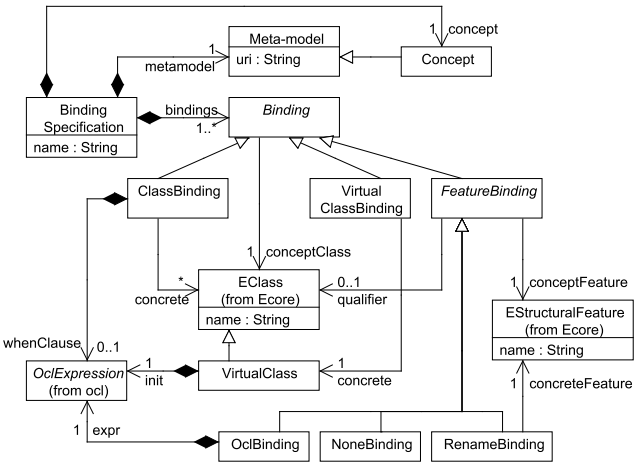


Fig. 9. Meta-model to describe bindings.

the concept to the same meta-model class is allowed. This binding type may include an OCL expression (whenClause), evaluated in the context of the bound meta-model class, to rule out certain instances from the mapping. As an example, lines 9–12 of Listing 2 show a class binding with a whenClause.

A VirtualClassBinding maps a class in the concept to a user-defined *virtual class*. Virtual classes are used when there is some mandatory class in the concept for which there is no meta-model class counterpart, but the meta-model represents the same information using other more complex structure. In such a case, a VirtualClass can be defined as part of the binding specification (we omit this relation in the meta-model for simplicity), and its instances are created by means of an OCL expression (init) that is evaluated before executing the instantiated transformation.

A FeatureBinding defines the binding for a feature (attribute or reference) of a concept class. If this class is bound to several meta-model classes, then the binding must be qualified with the concrete meta-model class in order to disambiguate (relation qualifier). There are three kinds of feature bindings: RenameBinding, OclBinding and NoneBinding. RenameBinding establishes a direct mapping with a feature in the bound meta-model. OclBinding maps the feature to an OCL expression, which is evaluated in the context of the bound meta-model class and must return a suitable value for the binding (i.e., of compatible type with the feature). We call this kind of binding *adapters*. NoneBinding is used to let a feature unmapped, but is only allowed for bindings of target concepts. For instance, in Listing 2, line 17 is a rename binding, while line 18 is an adapter.

Regarding the inheritance relations appearing in the concepts, they do not need to be mapped, as our binding is actually a structural mapping, in the line of structural subtyping in programming languages [23]. This means that the binding is defined for a *flattened* version of the concept, and it is only compulsory to map the leaf classes in inheritance hierarchies together

with their features (both owned and inherited).

Table 1 summarizes the well-formedness rules that bindings should obey (adapted from [7]). In the table, X.ancestors is the set of ancestors of class X, including X itself, while X.features is the set of attributes and references defined in class X. Condition 1 states that the features defined by a concept class, can be bound to owned or inherited features of the mapped meta-model class. Condition 2 states that the binding should preserve the subtyping relations, i.e., subtype classes in a concept should be mapped to subtype classes in the bound meta-model, and similar for non-subtype classes. Conditions 3–6 are requirements for the type and characteristics of references and attributes (we use $\trianglelefteq$ to denote subtyping). These conditions only apply to RenameBindings, as heterogeneities (e.g., regarding cardinality and type of references and attributes) can be solved through adapters.

TABLE 1
Well-formedness conditions for bindings.

	Scope	Condition
1	features	Given a class A $\in$ concept, and a feature f $\in$ A.features, then bind(f) $\in$ X.features for some class X $\in$ bind(A).ancestors
2	subtyping	Given two distinct classes A,B $\in$ concept s.t. bind(A), bind(B) are defined, then A $\in$ B.ancestors $\Rightarrow$ bind(A) $\in$ bind(B).ancestors $\wedge$ A $\notin$ B.ancestors $\Rightarrow$ bind(A) $\notin$ bind(B).ancestors
3	composition	Given a composition reference r $\in$ concept, then bind(r) is a composition
4	type of attributes	Given an attribute a $\in$ concept, then bind(a).type $\trianglelefteq$ a.type
5	type of references	Given a reference r to a class A $\in$ concept, then bind(r).target $\trianglelefteq$ bind(A)
6	cardinality	Given a reference r $\in$ concept, then bind(r).mincard $\geq$ r.mincard $\wedge$ r.maxcard $\neq$ * $\Rightarrow$ bind(r).maxcard $\leq$ r.maxcard

5.2 Bridging heterogeneities

A key feature of our approach is the ability of bindings to fix a wide range of heterogeneities between a concept and a meta-model, in order to increase the reuse opportunities of a given template. From the experience gained by defining bindings to complex meta-models (see Section 8), we have built a classification of heterogeneities that our DSL can solve. Table 2 introduces the most common ones, using examples based on possible concepts and meta-models for class diagrams, and giving a binding that solves the heterogeneity. This classification is not exhaustive since the use of adapters may fix other unforeseen heterogeneities. Section 8 evaluates to what extent these heterogeneities appear in practice. The online supplemental material includes the rules to adapt a transformation template according to a binding.

5.3 Inference of adapters

Even though adapters are a general mechanism to resolve heterogeneities, there are recurring heterogeneities that require writing boilerplate OCL adaptation code. Some of these patterns are discussed in [24]. To avoid encoding such bindings repeatedly, we have extended our DSL for the automatic inference of the adaptation code. This is done by analysing the differences between a concept feature and the corresponding meta-model feature, and generating the OCL code that resolves the heterogeneity. Table 3 summarizes the supported inference rules. For example, an *automated filtering* occurs when a collection in the concept (e.g., ownEls) is mapped to another one of a supertype, so that an expression is generated that selects only the elements of the right type, filtering out the objects of sibling classes. The last column of the table shows the generated adapter, which replaces the feature binding in the third column.

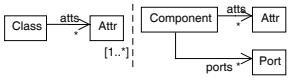
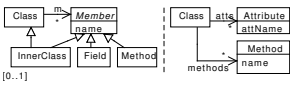
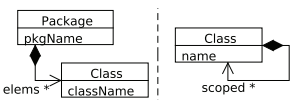
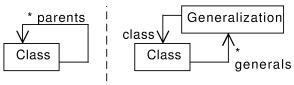
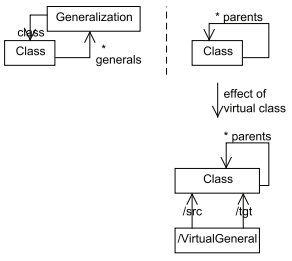
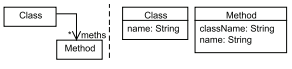
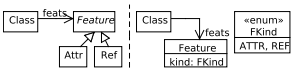
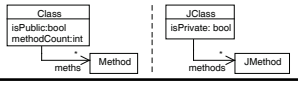
Other binding adapters inferred by our DSL convert multiplicities (widening from the meta-model to the concept), resolve differences in the direction of a reference, or split/merge references (reference merging is a collection of multiple *automated filterings*). There are also special cases of these adapters when references have cardinality 0..1 or 1 (instead of "*").

5.4 Restricting transformation applicability

Normally, meta-models are bigger than the concept they get bound to. While these extra elements may be unimportant for the transformation, sometimes they have an impact. For example, rich workflow languages may have more primitives than the ones supported by our Workflow concept. This is the case of the *XOR-join* (or *multi-merge*) gateway, present in BPMN but not in our concept. A model that contains instances of the ignored primitives may be incorrectly transformed into a Petri net, because either these instances will not be transformed, or alternatively, they may be treated as Nodes (should Nodes have assigned a default transformation rule), which may not reflect the semantics of the primitives. To solve this problem, our solution reports warnings when an input model contains instances of a type which is not mapped to any element in the concept, but some of its supertypes are mapped. This may indicate that the concrete meta-model contains elements with a special semantics not foreseen by the concept, whose presence in a model may trigger an incorrect transformation.

Additionally, the developer of the binding can define OCL constraints to restrict the input models to which the adapted transformation will be applicable. This is useful when the provided binding does not support some feature of the mapped meta-model. For example, while in BPMN it is possible to start a process via an Event-Based Exclusive Gateway, the developer may decide to postpone this possibility to

TABLE 2
Natively resolvable heterogeneities.

Name	Description	Binding	Diagram (concept to the left)
Class split	A concept class is mapped to more than one meta-model class. This means that several meta-model classes play the same role as a concept class, but they lack a common ancestor that can be mapped to the concept class.	**class** Attr **to** Attr, Port −− *In this case the example also includes* −− *reference split (see Table 3)* **feature** Class.atts **is** atts, ports	
Flatten hierarchy	An abstract superclass is not mapped. In this case, mapping only the leaf classes is allowed, whenever all features inherited from the superclass are mapped. Some leaf classes can be mapped to none if their lower cardinality is 0.	**class** Class **to** Class **class** Field **to** Attribute **class** Method **to** Method **class** InnerClass **to NONE** **feature** Field.name **is** attName **feature** Method.name **is** name	
Class merge	Two or more classes in the concept, belonging to different hierarchies, are mapped to a unique meta-model class. This means that the meta-model class has several distinct behaviours.	**class** Package **to** Class **class** Class **to** Class **feature** Package.pkgName **is** name **feature** Class.className **is** name **feature** Package.elems **is** scoped	
Association to class	An association in the concept is represented as an intermediate class in the meta-model (special case of *association to navigation expression*).	**class** Class **to** Class **feature** Class.parents = **self**.generals−>**collect**(g | g.'class')	
Class to association	A class in the concept is represented as an association in the meta-model. A user-defined virtual class mapped to the concept class is created. Its instances are populated by means of an OCL expression over the meta-model. The virtual class can be normally used as a standard meta-model class, though it is only available at run-time.	**class** VirtualGeneral { **ref** src: Class **ref** tgt: Class } **init** = Class::allInstances()−>**collect**(c1| c1.parents−>**collect**(c2| #VirtualGeneral { src = c1, tgt = c2 })) **class** Class **to** Class **class** Generalization **to virtual** VirtualGeneral **feature** Generalization.'class' **is** tgt **feature** Class.generals = VirtualGeneral::allInstances()−>**select**(g| g.src = **self**)	
Association to navigation expression	A reference in the concept is represented by other means in the meta-model.	**class** Class **to** Class **class** Method **to** Method **feature** Class.meths = Method::allInstances()−> **select**(m | m.className = **self**.name)	
Subclass to enumerate	A class hierarchy in the concept is represented in the meta-model using a single class with an attribute that takes value from an enumerated type.	**class** Class **to** Class **class** Attr **to** Feature **when self**.kind = #ATTR **class** Ref **to** Feature **when self**.kind = #REF	
Attribute conversion	Conversion between primitive datatypes, and from non-primitive to primitive datatypes.	**class** Class **to** JClass **feature** isPublic = not **self**.isPrivate **feature** methodCount = **self**.methods−>**size**()	

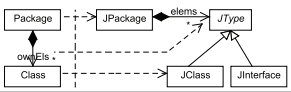
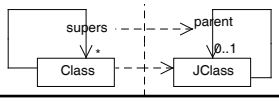
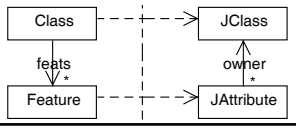
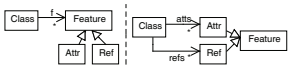
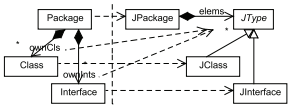
facilitate the definition of the first versions of the binding. For this purpose, he must provide an OCL constraint, so that the system notifies that any model containing this gateway is not supported.

6 Selection of Components

Selection is the process by which users are able to locate and select a reusable artefact from a collection. It is typically facilitated by concise abstractions, which can be understood and compared easily. In our case, concepts are abstractions of meta-models that only comprise the elements used by a transformation template. In this way, concepts characterise the required structure of the input and output meta-models of a component, and can be used to discriminate between different components in a collection.

In addition, we propose three ways to characterise components: their tagging by keywords, text-based documentation, and contract-based documentation. The latter is realised using PAMOMO [19], [25], a formal, pattern-based, declarative language to express transformation contracts. These describe *what* a transformation does, abstracting from its concrete implementation by a transformation template. PAMOMO contracts can include three kinds of properties: *preconditions* that any input model to the transformation must fulfil, *postconditions* that any model generated by the transformation satisfies, and *invariants* specifying

TABLE 3
Adapter inference.

Name	Diagram (concept to the left)	Binding	Induced adapter
Automated filtering		**class** Package **is** JPackage **class** Class **is** JClass **feature** Package.ownEls **is** elems	**feature** Package.ownEls = **self**.elems−>**select**(e | e.oclIsKindOf(JClass))
Multiplicity conversion (0..1 to 0..n)		**class** Class **is** JClass **feature** Class.supers **is** parent	**feature** Class.supers = Sequence { **self**.parent }−> **excluding**(OclUndefined)
Opposite navigation direction		**class** Class **is** JClass **class** Feature **is** JAttribute **feature** Class.feats **is** JAttribute.owner	**feature** Class.feats = JAttribute::allInstances()−> **select**(a | a.owner = **self**)
Reference split		**class** Class **to** Class **class** Attr **to** Attr **class** Ref **to** Ref **feature** Class.f **is** atts, refs	**feature** Class.f = **self**.atts−>**union**(**self**.refs)
Reference merge		**class** Package **to** JPackage **class** Class **to** JClass **class** Interface **to** JInterface **feature** Package.ownCls **is** elems **feature** Package.ownInts **is** elems	**feature** Package.ownCls = **self**.elems−>**select**(e | e.oclIsKindOf(JClass)) **feature** Package.ownInts = **self**.elems−>**select**(e | e.oclIsKindOf(JInterface))

relations between the input and output models which follow the pattern "*if the input model contains the given graph structure, the transformation generates a certain output graph structure*". In all cases, it is possible to indicate whether the property is positive to express expected graph configurations, or negative to indicate forbidden ones. Moreover, properties can be defined to hold only when certain combinations of features are selected, or independently of any feature selection (the default).

Figure 10 shows part of the contract for our running example, containing one precondition, one postcondition and two invariants. The precondition *OneStartEvent* is positive – indicated by the letter P before the precondition's name – and it is associated to the value of the feature initialState. The property states that if the variant mandatory for the feature initialState is selected, then any input model to the instantiated transformation component should have an initial task. The remaining properties in the figure hold regardless of the selected features. The postcondition *InitialPlace* is negative – indicated by the letter N before the postcondition's name – and expresses that in the generated net, the places that contain some token do not have incoming arcs. Finally, the two invariants correspond to the actual transformation of synchronization objects (*Synchronization*) and exclusive choice objects (*ExclusiveChoice*). The interested reader can consult PAMOMO's formal semantics at [25].

Contracts serve as documentation, and can be used to automate the testing of the transformations (see [26] for details). They may also facilitate the construction

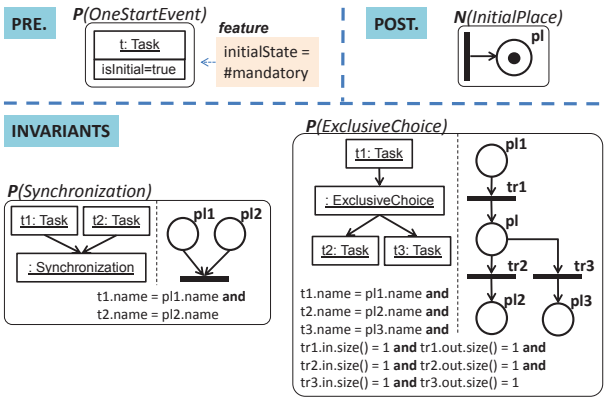


Fig. 10. Some properties of the transformation.

of chains of components by identifying whether the postconditions of a given component are compatible with the preconditions of a subsequent one. An automation of this checking is left for future work.

7 INTEGRATING TRANSFORMATIONS

Integration allows creating a software system by combining reusable artefacts. In an MDE setting, it is achieved by linking the output models of a transformation with the input models required by another, forming a transformation chain. Our transformations are encapsulated in components, and expose concepts and meta-models which can be seen as a specification of the compositionality requirements for a component. Hence, we can chain a component c_1 to c_2 if the output concept/meta-model of c_1 can be bound to the input

concept of c_2, or is the same. Thus, our approach seamlessly supports composite components, as they have the same interface as simple components.

Figure 11 shows a slice of our component meta-model with the main elements of the integration language. To facilitate the creation of rich composite components, we support three composition constructs: component instantiation, sequencing and variant selection, which correspond to meta-classes Apply, Seq and Xor. Seq allows the definition of a sequence of *composition steps*. Xor permits selecting a transformation path using a simple expression language (subclasses of Expression). Apply is used to instantiate a component, linking the formal parameters (models declared in the called component) with the actual parameters (models declared in the callee). The following three subsections give examples of these constructs.

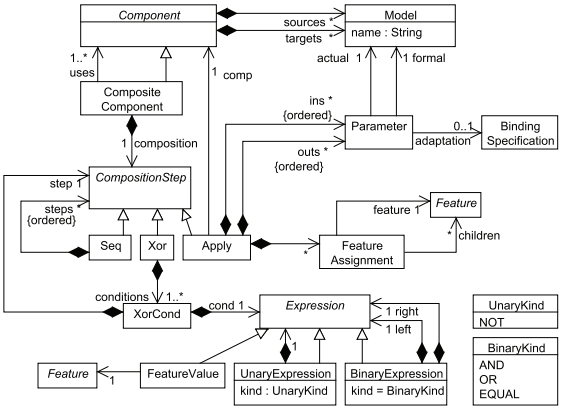


Fig. 11. Slice of the component meta-model of Figure 4: component integration.

7.1 Component instantiation

A component is instantiated by wrapping it into a composite component, and linking the input/output models of the former component to the input/output models of the composite one. As part of this linking, it is possible to indicate a binding, which induces an automatic adaptation of the instantiated component. Additionally, it is possible to assign a value to the feature variants exposed by the component.

Listing 3 instantiates the component for transforming workflow languages into Petri nets, using a binding to the Intalio meta-model. The listing declares the composite component in line 1, imports the component to be instantiated in line 2 (fd2pn.gcomponent), and the binding used for the instantiation in line 3 (bindIntalio.gbind, partially shown in Listing 2). Bindings have a name (bindIntalio in line 3) that is used to apply the binding (line 10). Lines 5–8 declare the component interface. In this case, the source of the composite component is the Intalio meta-model (line 5). The actual instantiation of the fd2pn component is done in lines 10-12. The syntax bindIntalio(in) applies the binding of the Intalio meta-model to the workflow concept, while fd2pn(bindIntalio(in)) induces the adaptation of the fd2pn component using the binding. Finally, the with keyword introduces pairs *feature* = *value* to select the component variants.

```
1  composite component Intalio2PN {
2    uses "fd2pn.gcomponent"
3    binding bindIntalio = "bindIntalio.gbind"
4
5    source metamodel Intalio : "Intalio.ecore"
6    target concept PN : "PetriNet.ecore"
7    source model in : Intalio
8    target model out : PN
9
10   compose apply fd2pn(bindIntalio(in)) -> (out)
11     with initialState = #optional,
12          finalState = #optional
13 }
```

Listing 3. A composite component with binding.

The instantiation mechanism permits creating a concrete component that can be readily executed. This is the case of the example, as all variable parts of the component (i.e., concepts and variants) are given a concrete value. Alternatively, it is possible to create a composite component that needs to be further instantiated by leaving some variable element unbound.

7.2 Sequencing components

Component sequencing permits connecting a series of components, which are executed in sequence.

Figure 12 shows how to chain the transformation component for workflow languages with an analysis component – named PNAnalysis and implemented in Java – which encapsulates invocations to the PIPE2 tool [27]. The Petri net model produced by the FlowDiagram to PetriNet component is directly passed to the analyser since it exposes the same concept PN as input port. Then, the analyser generates two models: one reports problems regarding workflow completion, reachability, etc. (Analysis), and the other contains the reachability graph to allow its visualization or further analysis (StateSpace). To facilitate the visualization of this reachability graph, the chain includes a third component called GraphLayouts which takes a graph as input and creates a layout model annotating each graph node with its position (see Figure 13). A binding makes the layout component compatible with the graph produced by the analysis component.

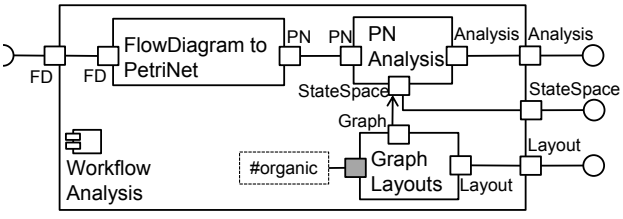


Fig. 12. A component with a transformation chain.

The resulting component chains one Java and two transformation components, and can be used to analyse models of different workflow languages, as the

component is still defined over the workflow concept, which can be bound to a variety of languages.

7.3 Transformation selection

A composite component may need to decide between several execution paths in a transformation chain, depending on the selected configuration for its exposed features. For this purpose, the value of these features can be queried at execution time. We currently support exclusive choice and an expression language to define execution variants and check their value.

Figure 13 shows the composite component for graph layouting. It admits two kinds of layouts, customizable through a feature with two possible values: circle and organic. Inside the component, the value selected for the feature is used to apply one of the two transformations for each layout.

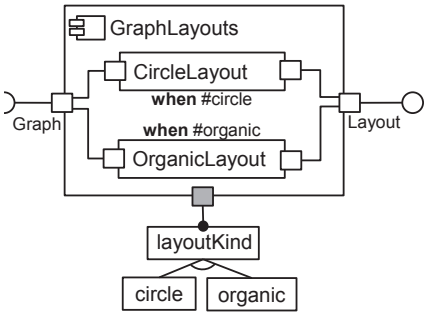


Fig. 13. A component with transformation selection.

8 Evaluation

In this section, we evaluate two aspects of our component model. First, we evaluate its reusability potential by applying a synthetic mapping benchmark [28], and creating bindings from the Workflow concept to several meta-models. Second, we discuss on the abstraction power of concepts to act as the reuse interface of components. The section concludes with a discussion of the obtained results and lessons learnt. Our Bentō tool, as well as the different experiments of this section, are available at: http://www.miso.es/tools/bento.html.

8.1 Reusability potential

In order to evaluate the reuse potential of our components, we have analysed the flexibility of our binding DSL to adapt a component to unforeseen contexts (i.e., to different meta-models).

First, we used the *STBenchmark* [28], which identifies ten frequent scenarios for schema mapping in database systems. We are able to solve nine of them, with the exception of the manipulation of primitive values (e.g., splitting a string address into three strings for the street, number and postal code) as this is limited by the capabilities of OCL.

Then, we evaluated the applicability of our approach in more comprehensive scenarios by binding our Workflow concept to nine languages with workflow-like semantics. The criterion to select the languages was the availability of a meta-model created by a third-party, which we could import into an Ecore meta-model. The only exception was *Bender*, a DSL that we created in the context of an industrial project to describe the workflow of telephony services [29]. For all meta-models, we studied the kind of heterogeneities solved, as well as the complexity of the bindings in terms of the primitives and OCL expressions used. Table 4 summarizes the results of this evaluation. The first section (*DSL Constructs*) counts the number of constructs of our DSL used in the bindings; the second section (*Heterogeneities*) shows the kind of heterogeneities solved using DSL constructs; the third section (*Automated features*) corresponds to inference of adapters; the last two sections show metrics of the size and complexity of the bindings.

Our Workflow concept has 10 classes, 6 associations and 3 attributes. Thus, the simplest binding would consist in 10 class renamings and 9 feature renamings. However, all bound meta-models required solving some heterogeneity beyond renamings. This indicates that approaches for transformation reuse should provide means to bridge heterogeneities. For instance, an important source of variability in our case study was the way in which meta-models represent the flow of elements. Figure 14 shows the five different representations we found in the studied meta-models.

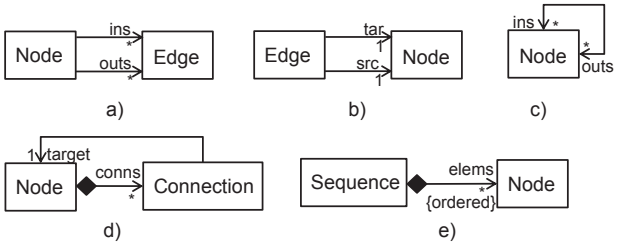


Fig. 14. Different ways of modelling a flow of elements.

We were able to successfully reuse the fd2pn component with eight out of the nine meta-models considered. In the successful cases, the bindings that contain more class/attribute/association renamings correspond to meta-models that are more similar to our concept. This is the case for Activity Diagrams (*UML AD* in Table 4). Conversely, bindings that bridge many heterogeneities have few renamings and sometimes complex OCL expressions, like in *Intalio*, *BPMN*, *EPC* and *YAWL*. Thus, the complexity of the binding highly depends on the similarity between the concept and the bound meta-model. The only meta-model we were not able to bind to our concept is the BPEL meta-model, because BPEL represents the flow of elements as a tree of nested actions (see Figure 14(e)), while our

TABLE 4
Bindings to different workflow languages.

	Intalio BPMN	OMG BPMN	UML AD	EPC	YAWL	Grafcet	Apromore	Bender
DSL constructs								
Class renaming	3	4	8	3	3	6	8	4
Class to classes (cardinality > 1)	-	1	1	1	1	2	-	2
Class filtering (*when* clause)	7	5	-	7	6	-	2	-
Class to none	-	-	1	-	-	1	-	4
Class to virtual class	-	-	-	-	1	-	-	-
Feature renaming	5	6	8	5	4	2	5	6
Feature adapter (OCL bindings)	4	5	5	6	7	17	4	8
Heterogeneities (see definitions in Table 2)								
Class split	-	1	1	1	1	2	-	2
Class merge	-	-	-	3	-	-	-	1
Class to association	-	-	-	-	1	-	-	-
Attribute renaming	1	2	2	2	1	-	1	1
Attribute to literal[a]	1	2	4	2	4	2	1	2
Attribute conversion	1	1	1	1	-	-	1	1
Association renaming	4	4	6	3	3	-	4	5
Association to class	2	-	-	-	-	1	-	1
Association to navigation expr.	2	2	-	3	3	16	2	5
Flatten hierarchy	-	-	-	-	-	1	-	1
Subclass to enumerate	7	1	-	-	5	-	-	-
Automated features (see definitions in Table 3)								
Automated filtering	-	2	-	1	-	-	-	1
Opposite navigation direction	-	-	-	-	-	-	2	2
Reference split	-	-	-	2	-	-	-	-
Reference merge	-	-	-	-	-	1	-	-
Lines of code								
Binding specification	34	28	29	42	37	43	25	35
Original transformation[b,c]	267	267	246	267	267	257	267	190
Adapted transformation[b]	353	357	314	679	478	687	313	896
Complexity of OCL expressions (total / average)								
Complexity of OCL bindings	14/2.80	16/2.67	13/1.44	39/4.87	26/2.89	66/3.88	16/3.20	29/2.90
Complexity of class filtering (*when*)	83/11.86	47/9.40	–	48/6.86	34/5.67	–	4/2.00	–

a. Adapters that return a primitive value (e.g., true or 'initial').
b. The transformations were automatically formatted.
c. For each language, we removed the unused features from the original transformation, for a more realistic comparison.

concept (and most languages in our study) represents the flow as a graph. Even if we can map basic features in BPEL such as action sequencing by writing a fairly complex binding, other features like the notions of choice and merge are too different.

Regarding the OCL expressions used in the bindings, we have measured their complexity by counting the number of nodes in the abstract syntax tree of each expression. For example, the expression `self.edges->isEmpty()` counts as two nodes. The last two rows in Table 4 show the total and average complexity of the OCL expressions used in feature bindings and class filters. For instance, the binding for `ParallelSplit` in the case of *OMG BPMN* has the following filter (6 nodes): `self.outgoing->size() = 1 and self.incoming->size() > 1`. Although filters may become large if many conditions need to be checked, they do not involve complex model navigations.

Other source of complexity in our case study is that our concept does not include constructs to represent hierarchical organization of flow diagrams. Sometimes, this can be solved by writing adaptation code to "flatten" the diagram, while in other cases, this may restrict the applicability of the adapted transformation. As a rule of thumb, if the component developer foresees that a construct in the concept may facilitate future bindings, he should include it as optional.

To compare the effort of defining a binding with respect to developing the transformation from scratch, we have counted the number of rules, helpers, LOC (without blank lines or comments) and OCL expressions in the transformation template, which amount to 11 rules, 23 helpers, 267 LOC and 488 nodes in OCL expressions. In contrast, the relatively complex binding for *Intalio BPMN* has 10 class bindings, 5 feature renamings, 5 feature adapters (similar to helpers), 34 LOC and 97 OCL nodes. This shows that bindings are normally shorter and simpler than full transformations, as bindings do not deal with the "essential" complexity of the transformation, but they only bridge the accidental complexity (from the transformation perspective) introduced by domain meta-models.

Finally, a further benefit of our approach is that, to reuse a component, the developer only requires expertise in his own domain meta-model (e.g., the specific workflow language), in order to define a binding to the concept. Hence, he does not require deep knowledge of the fixed elements (e.g., Petri nets), and the implementation details of the template remain as a black-box, as there is no need to tackle, devise or manipulate the transformation logic. We evaluate the abstraction power of concepts in the next section.

8.2 Abstraction power of concepts

An ad-hoc approach to transformation reuse, based on copy-paste from existing transformations, requires facing the complexity of the transformation logic and the meta-model over which it is defined, in order to adapt the former to the new usage context. This may be difficult, especially for complex transformations or large meta-models. Instead, we propose concepts as a succinct interface for reuse.

To assess whether concepts are a better reuse interface than plain meta-models, this section evaluates the complexity of reusing an existing transformation defined over a meta-model (ad-hoc approach), compared to reusing a transformation component with equivalent behaviour but which provides a concept as reuse interface. For the ad-hoc approach, we have used the existing transformation UML2Measure[3] from the ATL Zoo, which computes 51 object-oriented metrics for UML2 models. Its input meta-model is UML2, and the output meta-model allows representing measurements. On the other hand, we have developed an equivalent transformation component over a source concept, called C_{OO}, specifically designed for this component. Finally, to compare both solutions, we have built another transformation component out of the existing UML2Measure transformation. To this end, we identified the effective meta-model of the transformation, which is the subset of meta-model classes, associations and attributes used by the transformation. Then, we defined the transformation component using the effective meta-model as concept, and the original transformation as transformation template. The concept has been called C_{UML}.

Since one of the difficulties when reusing a transformation is understanding the manipulated data structures, we have compared the concept C_{OO} developed with reusability in mind, and the concept C_{UML} resulting from the ad-hoc approach. C_{OO} (9 classes, 5 attributes, 11 associations and 7 inheritance relations) is smaller than C_{UML} (17 classes, 1 enumerate type, 3 attributes, 12 associations and 17 inheritance relations). The size of C_{UML} is large due to the inherent complexity of the UML2 meta-model, which has many intermediate abstract classes and deep inheritance hierarchies. We did not collapse the inheritance chains in C_{UML} because, without analysing the transformation, it was uncertain whether this would modify the transformation behaviour. In contrast, C_{OO} is small because both the transformation component and the concept were created with reusability in mind.

Table 5 shows a detailed comparison of their complexity, using some object-oriented metrics proposed in [30] and adapted to meta-models in [31], related to understandability and functionality quality attributes. High values of DSC, ANA, DCC and ADI influence negatively the understandability. This makes C_{OO} more suitable as interface for reuse, since it will be easier to identify correspondences in the specialization stage. Interestingly, the value of the NOH metric for UML and C_{UML} is higher than for C_{OO}, indicating that the meta-models provide more functionality than the concept. This is indeed an important characteristic: a concept is an interface tailored for the functionality of a specific transformation.

TABLE 5
Metrics applied to UML 2.4, the UML-based concept (C_{UML}) and the handcrafted concept (C_{OO}).

Metric	UML	C_{UML}	C_{OO}
Design Size in Classes (DSC)	247	17	9
Number of Hierarchies (NOH)	246	11	5
Average Number of Ancestors (ANA)	6.91	1.75	0.88
(Average) Direct Class Coupling (DCC)	15.20	1.18	1.44
Average Depth of Inheritance (ADI)	5.60	1.32	0.66

While a tailor-made concept is simpler, the second question is whether this leads to simpler bindings. To evaluate this, we have defined bindings from the C_{UML} and C_{OO} concepts to three meta-models: UML, Ecore and KM3. Table 6 summarizes their complexity. The bindings from C_{UML} to Ecore and KM3 have many renamings, which in principle might indicate that they were simple to identify. However, as C_{UML} has deep inheritance hierarchies, sometimes it was difficult to determine which class in the hierarchy should be bound to a certain meta-model class.

In contrast, concepts usually have shallow hierarchies, which facilitates finding the meta-model elements to be bound and the flattening of hierarchies. As Table 6 shows, the bindings from C_{OO} to Ecore and KM3 are shorter, being mostly renamings, and requiring few simple adapters. The complexity of the binding from C_{OO} to UML is similar. Besides, the transformation for C_{OO} is shorter than the original (485 vs 740 LOC), since it was developed from scratch. In our experience, using concepts facilitates the implementation of a transformation, as the developer does not need to deal with the accidental details of concrete modelling languages.

In summary, our evaluation shows that the concepts developed with reusability in mind tend to be simpler, and the meaning of each concept element is more clearly identifiable. Moreover, a well-designed concept becomes a good abstraction mechanism for transformation reuse. The downside is that they have to be handcrafted to be effective (in particular, calculating the effective meta-model automatically may not be enough [32]). One of our lines of future work is developing automatic methods to derive concise concepts from existing meta-models and transformations, e.g., collapsing hierarchies and compacting classes, while preserving the transformation behaviour.

3. http://www.eclipse.org/atl/atlTransformations/#UML22Measure

TABLE 6
Bindings to C_{UML} and C_{OO} concepts.

	C_{UML}			C_{OO}		
	UML	Ecore	KM3	UML	Ecore	KM3
DSL constructs						
Class renaming	18	11	12	9	7	7
Class to classes (cardinality > 1)	-	2	1	-	-	-
Class filtering (*when* clause)	-	-	-	-	-	-
Class to none	-	4	4	-	2	2
Class to virtual class	-	1	1	-	-	-
Feature renaming	15	7	8	8	6	6
Feature adapter (OCL bindings)	-	6	5	8	8	8
Heterogeneities (see definitions in Table 2)						
Class split	-	2	1	-	-	-
Class to association	-	1	1	-	-	-
Attribute renaming	3	1	1	2	2	2
Attribute to literal	-	1	1	-	3	3
Attribute conversion	-	1	1	3	-	-
Association renaming	12	6	7	6	4	4
Association to class	-	-	-	1	-	-
Association to navigation expr.	-	4	3	5	5	5
Lines of code						
Binding specification	37	41	41	29	27	27
Original transformation	740	740	740	485	428	428
Adapted transformation	740	809	794	521	527	521
Complexity of OCL expressions (total / average)						
Complexity of OCL bindings	-	20/3.33	16/3.2	41/5.16	22/2.75	22/2.75
Complexity of class filtering (*when*)	-	-	–	-	-	-

8.3 Threats to validity

The component and bindings in the running example (workflow languages) were defined by us. To mitigate the risk that the transformation was written to favour the subsequent bindings, we split the tasks: first, one author developed the component (concept and transformation template), and the other two authors defined the bindings once the component was ready. No change was made to the component once the binding stage started.

Even if we could have bound a wider spectrum of meta-models in our running example, the ones chosen cover a large range of heterogeneities (cf. Figure 14). The meta-models were not purposely chosen with a bias towards easily resolvable heterogeneities, but we selected meta-models developed by third parties, focussing on standards like BPMN and UML Activity Diagrams. Indeed, the only self-developed meta-model (Bender) was not the simplest in the case study.

The transformation template for the C_{OO} concept was manually implemented based on the original UML2Measure. However, the concept was defined independently of the UML meta-model by analysing the measurements computed by the original transformation. This was done to avoid reflecting any design decision of the UML meta-model into the concept. Then, the template was written from scratch, trying to preserve the transformation structure (i.e., similar rules, helper operations and OCL expressions).

In this case, we experienced more difficulties establishing bindings for C_{UML} than for C_{OO}. Since this is a subjective observation, we applied metrics that confirmed that C_{UML} was less comprehensible than C_{OO}. However, other factors such as the size and structure of the bound meta-models were not taken into account in the evaluation.

We have used the LOC metric to compare the complexity of transformations and binding specifications. To prevent a bias related to the code format, we formatted all texts in the same way using the ATL formatter and removing blank lines. Although the results show reusability gain using our approach (less LOC of the binding than in the reused transformation), we would need to confirm these results by measuring the effort needed to write a binding with respect to writing an ATL transformation. Nonetheless, we foresee less effort in the case of bindings because the primitives provided by our binding DSL are more limited, but focused on the task, than those provided by a general-purpose transformation language.

The main threat to the external validity is that the example is limited to workflow languages. To minimize the risk that the heterogeneities in these languages coincide with the ones we can tackle, we carried out the synthetic *STBenchmark*, which proposes scenarios for database systems. We have also developed a component that calculates metrics for object-oriented modelling languages, and defined bindings from the C_{OO} concept to UML, Ecore and KM3. This provides further evidence that our binding is able to adapt to unforeseen contexts and different domains.

A last threat is that our template adaptation procedure is only available for ATL. The features of other transformation languages may limit the range of resolvable heterogeneities. For instance, we foresee a more rigid adaptation procedure for QVT-Relations,

which would need to respect the bi-directional semantics of the transformation. On the contrary, for QVT-Operational, which features explicit rule calls and unidirectional behaviour, similar results to ATL could be achieved. Confirming this hypothesis is future work.

8.4 Discussion

Transformation reuse. Our specialization mechanism is key in our component model. It is based on adapting the transformation template to make it compatible with the bound meta-models. Thus, a relevant question is whether the supported adaptations cover practical, real-life scenarios. Our experiments evince that this is the case, and that the heterogeneities we are able to solve for ATL are expressive enough. Nevertheless, we cannot claim that all adaptations are possible or practical, as some bindings might be too complex, as in the case of BPEL. Moreover, supporting other transformation languages different from ATL requires developing an adaptation procedure for them.

An alternative approach, typically used in compiler construction to reuse analysis and compiler backends, is to define an intermediate language and transform from some programming language to the intermediate language. In MDE, the usual approach to reuse a transformation defined over a meta-model MM, for a different meta-model MM', is to write a second transformation from MM' to MM, and then apply the transformation of interest. This implies writing a full-fledged transformation from MM to MM' and executing two transformations instead of one. If the transformation needs to be applied frequently, or models are large, this may incur in sever efficiency penalties. The traceability between models conformant to MM' and the target models of the transformation also gets complex to analyse due to the intermediate model.

Instead of writing a regular transformation, we propose the use of a DSL which facilitates the specification of bindings, allows checking the syntactic correctness of the binding with respect to the concept and bound languages, and provides facilities such as binding inference.

Hence, our approach is an alternative to the creation of intermediate models. Nonetheless, a binding from a concept C to a meta-model MM could also be used to produce a transformation from MM to C, by generating a copy transformation from C to C, and then adapting it for MM. This option can be used when the transformation language is complex to adapt, like in the case of Java components.

Binding development. From our experience defining bindings, we have learnt that the simpler the concept, the simpler the binding, but also that the size and complexity of the bound meta-models have some influence. Advanced tooling would facilitate the specification of bindings by providing auto-completion mechanisms, meta-model flattening facilities and binding suggestions. Providing such tooling is one of our lines of work to improve Bentō.

We have also found that some heterogeneities are easier to resolve than others. For instance, resolving the *class-to-association* heterogeneity is harder than *association-to-class* (see Table 2). Thus, it is recommended to avoid classes acting as associations (e.g., Edge) in concepts. However, this choice can make the transformation template more complex, as was the case in the *Workflow to Petri net* component. This trade-off should be addressed taking into account the most common constructs of the domain.

Our system is able to check that a binding satisfies certain syntactic restrictions, normally encoded in the concept. However, the user is in charge of checking that the semantics provided by the concept fits the semantics of the bound meta-model. For example, the *Workflow to Petri net* component does not handle hierarchical tasks, and thus it can only be used to transform models without such tasks. To help in the detecting this situation, we provide warnings if the binding leaves elements unmapped (in the style of [33]), which may indicate a semantic error. Additionally, a transformation component can be configured via a feature model with the different semantics expected in the bound meta-models. If a meta-model has a semantics not foreseen by the feature model, then the component cannot be reused.

Component development. Regarding the effort required to build a component, developing a transformation template is less costly than writing a regular transformation with the same capabilities for a concrete meta-model (e.g., Intalio BPMN). This is because the accidental complexity in concrete meta-models is removed from concepts, which only need to focus on the essential complexity of the transformation (e.g., how to translate workflow constructs to Petri nets constructs). For instance, the hand-crafted transformation template for the C_{OO} concept is shorter (about 500 LOC) than the transformation developed for UML (more than 700 LOC), and therefore, it will be likely less error-prone. However, the additional effort lies on analysing the domain to extract a concept and a feature model that gathers the variability in the domain. This effort is inherent to any approach to reusability, where cost-effectiveness is achieved when the component is reused a few times.

Once a component is created, tested and deployed, its evolution is similar to the case of regular transformations. However, the components with a large degree of variability may profit from modularization techniques like the ones proposed in [34] (a related technique applicable to MDE and ATL is proposed in [35]), and product lines techniques applied to model transformations [36], [37]. Additionally, our components include a version number to allow several versions of the same component to coexist.

Altogether, our component model provides a com-

prehensive approach to transformation reuse that permits the classical approach but also more advanced features, like template adaptation, composite components and variants. From a practical perspective, a common model also facilitates reuse because it is possible to build tools around it, such as repositories, browsers and development environments.

9 Related Work

As acknowledged in [2], model transformations are mostly developed from scratch. Some of the reasons are the dependency on concrete meta-models, the lack of repositories with selection mechanisms, the difficulty to specialize transformations to different contexts, and the insufficient support for integration in the large. Our component model overcomes these shortcomings by covering the four dimensions of software reuse [13]: abstraction, specialization, selection and integration.

In the following, we review related approaches and techniques in the field of generic programming, components and model transformations reuse.

9.1 Generic programming and concepts

The underlying reuse mechanism of our proposal is based on generic programming, a paradigm found in many languages like C++, Haskell or Java [38]. C++ supports generic programming by a template system. An operation can be made generic by defining a template function that contains a set of type parameters, though the requirements of these parameters are not explicit. Concepts were proposed to overcome this limitation [9], but they have not been eventually included in C++0x, the last revision of C++ [39]. In Haskell, the requirements of a generic operation (i.e., a polymorphic function) are expressed through *type classes* [40]. A type can be made an instance of a given type class to make it compatible with it. Thus, implementing a type class is the Haskell equivalent to our bindings. In Scala, requirements on a type parameter can be expressed with a *trait*, implemented in a so-called *object*, and automatically selected for instantiation using the *implicit* mechanism [41].

Demeter interfaces [42] are used in Adaptive Programming as a way to decouple visitor-based computations from concrete class graphs. Similar to our binding, a Demeter interface can be bound to a concrete class graph, inducing an adaptation in the visitors' code. Interfaces can declare traversal strategies (navigation expressions) and simple constraints on how they can be bound.

The expressive power of these approaches is comparable to our basic binding mechanism plus adapters, although they require mapping every type parameter exactly once. Moreover, our approach resolves many heterogeneities by providing richer binding mechanisms, like adapter inference.

The Clafer modelling language [43] mixes meta-modelling, to describe the structural part of a language, and feature modelling, to describe variants on the structure. Instead, we separate structure (the concept) from the variants because only the former needs to be bound to concrete meta-model elements, while the variants are selected and usually correspond to different transformation strategies or execution paths.

9.2 Meta-model evolution

Several works deal with meta-model evolution [44], [45], [46] and the subsequent co-evolution of associated artefacts, typically models [46], but some works also deal with transformations [47]. The differences between a meta-model and its evolved version can be represented in different ways, like a difference model or a sequence of operations. Some approaches, like [48], are able to reconstruct the later from the former. There are also catalogues of common meta-model refactorings [44], enabling model migration.

Our work is related to these approaches, if we consider the binding from a concept to a meta-model as a meta-model evolution problem. Instead of relying on difference models or operation sequences, we provide a dedicated language to express the binding. This is adequate in our context, because the meta-model has not evolved from the concept but they have been created independently. Hence, there is no operator sequence or meta-model difference model to start with. Second, most approaches are concerned with migrating the instances, while we deal with transformation adaptation. Compared to [47], we provide a richer set of mechanisms for bridging heterogeneities, while the authors describe adaptations due to class renaming and reference split. Interestingly, they acknowledge the need for ways to specifying complex transformation evolutions, when such evolutions can be performed in different ways. Our adapters constitute one such mechanism.

9.3 Component models

Several general-purpose component models have been proposed [49], normally encapsulating components as objects or as architectural units (e.g., using an architecture description language). Our approach is specific for MDE, and the notions of component and component model are not embedded into another abstraction, but are first-class elements described by a meta-model.

Among the component models related to our approach, Koala [5] components are built atop Darwin, where the implementation is in C. Similar to our bindings, it is possible to write glue code binding a *required* to a *provided* interface. This code gets compiled into C. Koala components require configuration parameters through *diversity interfaces* and *diversity spreadsheets*. Interfaces and components are stored in repositories,

but there are neither search nor component description facilities.

Parametrized modules for algebraic specifications were proposed in the eighties [50]. Such ideas have been used in the context of graph transformation modules, which expose graph transformation rules as description of the module interfaces. In [51], the authors review several ways to map the meta-model and the rules of required and provided interfaces. While those mappings correspond to type renamings, our approach offers greater flexibility through adapters.

Specific to MDE, MDA Tool Components [52] are a packaging mechanism for model-based artefacts. Its focus is to enable the extension of modelling tools with pre-built components, which include artefacts to develop models with some modelling language. In [53], MDA Components are seen as a chain of PIM-PSM transformations with fixed source and target meta-models. In contrast, our transformation components can be adapted to different meta-models. Besides, no implementation of MDA Components has been released. In [54], an MDA component is made of a specification, an implementation and a set of test cases. A method is proposed to build a trustable component by relating these three elements using mutation analysis. These techniques, and others for model transformation testing [19], can be used to test the transformation templates in our components.

9.4 Transformation reuse and composition

Works on model transformation reuse can be classified into type-centric or type-independent (i.e., coupled or decoupled from concrete meta-models). The former include reuse mechanisms for single rules, like rule inheritance [55], and reuse mechanism for whole transformations, such as superimposition [56] and phases [57]. These proposals are restricted to the original meta-models or to extensions of them. In [34], an architecture to modularize code generators based on model transformations is presented. The generator modules are reusable and extensible, but limited to the original source language. Similar techniques adapted to out-place transformations, and thus applicable to ATL, are proposed in [35]. We can use all these mechanisms to improve the internal quality of our transformation templates.

Regarding type-independent reuse approaches, there are some proposals of fine-grained mechanisms based on reusable parameterized rules [58], [59], [60], and coarse-grained ones aimed at reusing complete transformations [61], [62], [63]. These approaches vary on their flexibility to resolve structural heterogeneities. For instance, only one-to-one mappings are allowed in [58], [59], [62], while one-to-many mappings are allowed in [61], and a wider range of heterogeneities can be solved by composing mapping rules in [60]. The expressiveness of the interfaces proposed for reuse also differs, ranging from unrelated parameterised types [58], [59] to sets of collaborating roles [61], [62] (similar to our notion of concept) where it is possible to configure some binding rules, like forbidding a class to play (i.e. to be bound to) two given roles in the set [62]. Some of these works are specific for model refactorings [61], [62], and most use an interpreted approach for the reuse of transformations: the mapping is used to resolve the concrete types at execution-time using a level of indirection. Instead, we use a compiled approach where a HOT creates a specific transformation for the mapped types, which results in better runtime performance. This compilation is also supported by [59] and by the DUALLY approach to make architectural languages interoperable [33], but their capabilities to resolve heterogeneities are limited. In [63], reuse is achieved by adapting the target meta-model to make it a subtype of the expected one. Our approach is as flexible as [60], but we support the use of regular transformation languages like ATL to define transformation templates, and there is no need to adapt the models and meta-models to be transformed. Another major difference of our approach is the use of concepts as an abstraction mechanism to serve as reuse interface.

There are several architectural design languages supporting the composition and orchestration of transformations. For instance, Wires [64] is a dedicated language for composing ATL transformations. It is similar to our language for wiring components, but we add template instantiation and parameterization mechanisms. UniTI [65] is more platform-independent. The MCC environment [66] offers a scripting language with composition operators enabling the design of transformation chains. In [67], the authors propose mechanisms to compose transformation chains by defining correspondence meta-models. In [68], the authors present a tool integration framework where MDE processes can be described and executed. Finally, MTC Flow [69] enables the definition of transformation flows using heterogeneous languages like ATL, QVT-O, Acceleo and Epsilon.

Table 7 compares the features of the main approaches to define transformation chains with our Bentō component model. In particular, Bentō is the only one enabling the composition of transformations by their adaptation to different meta-models.

Altogether, the contribution of our work is a flexible component model for model transformations, presenting advantages with respect to existing works: (i) reusable transformations are defined over concepts making them simpler to define; (ii) transformations can be adapted to specific meta-models through bindings, promoting their reutilization in a black-box manner; (iii) bindings induce a transformation adaptation, which results in an efficient reutilization approach; (iv) transformations are encapsulated in generic components, which promotes composability of transformations; and (v) components expose features, helping

TABLE 7
Comparison of transformation chaining languages.

	MTC Flow	UniTI	Wires	MCC	transML	Bentō
Sequential composition	√	√	√	√	√	√
Parallel composition	√	-	√	√	√	√
Conditional composition	√[a]	-	√	-	-	√[b]
Loops	-	-	√	-	-	-
HOT/execution[c]	-	-	√	-	√	-
Composite components	-	-	√	√	√	√
Transformation language	many	lang. indep.	ATL	Java	lang. indep.	ATL, Java
Features	-	-	-	-	-	√
Transformation adaptation	-	-	-	-	-	√

a. on tag values
b. on feature values
c. execution of transformations generated by HOTs

in the definition of transformation variants.

10 Conclusions and Future Work

In this paper, we have presented a novel reutilization approach for model transformations based on the definition of generic transformation templates over concepts, which can be bound to different metamodels. Transformation templates are encapsulated into components, which can be configured through features, and composed to form composite components. We have developed a tool that supports our approach. Moreover, we have evaluated our proposal with respect to flexibility of reuse and concept abstraction power using several realistic scenarios.

In the future, we would like to consider new kinds of components, like components for code generation or in-place transformation, as well as further transformation languages in addition to ATL. We also plan to use PAMOMO specifications as composability criteria for components (as currently they are only used for documentation), and explore new types of deployments for components, like web services. We plan to automate the process of making an existing transformation reusable by using advanced meta-model pruning techniques and automated transformation adaptation. We would also like to capitalize on existing metamodel evolution and differencing techniques, to semi-automatically derive a first version of the binding. We believe these techniques would help in transferring the approach into practice, for which initiatives for open component repositories (similar to the ATL zoo) are also necessary. Finally, it would be interesting to empirically evaluate how well developers are able to specify concepts and how well the text-based documentation, contracts, and tagged components allow other developers to identify reusable transformations.

Acknowledgements. We thank the reviewers for their comments. This work was supported by the Spanish Ministry of Economy and Competitivity with project Go-Lite (TIN2011-24139) and the EU commission with project MONDO (FP7-ICT-2013-10, #611125).

References

[1] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, ser. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.
[2] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger, "Reuse in model-to-model transformation languages: are we there yet?" *Software and System Modeling*, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10270-013-0343-7
[3] K.-K. Lau and Z. Wang, "Software component models," *IEEE Trans. Soft. Eng.*, vol. 33, no. 10, pp. 709–724, 2007.
[4] K. Saks, "JSR 318: Enterprise java beans, version 3.1," http://download.oracle.com/otndocs/jcp/ejb-3.1-mrel-evalu-oth-JSpec/, 2009.
[5] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, pp. 78–85, 2000.
[6] J. Sánchez Cuadrado, E. Guerra, and J. de Lara, "Generic model transformations: *Write once, reuse everywhere*," in *ICMT'11*, ser. LNCS, vol. 6707. Springer, 2011, pp. 62–77.
[7] ——, "Flexible model-to-model transformation templates: An application to ATL," *JOT*, vol. 11, no. 2, pp. 4:1–28, 2012.
[8] J. de Lara and E. Guerra, "From types to type requirements: Genericity for model-driven engineering," *Software and System Modeling*, vol. 12, no. 3, pp. 453–474, 2013.
[9] D. Gregor, J. Järvi, J. G. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine, "Concepts: linguistic support for generic programming in C++," in *OOPSLA*. ACM, 2006, pp. 291–310.
[10] A. Stepanov and P. McJones, *Elements of Programming*. Addison Wesley, 2009.
[11] R. Chenouard and F. Jouault, "Automatically discovering hidden transformation chaining constraints," in *MoDELS*, ser. LNCS, vol. 5795. Springer, 2009, pp. 92–106.
[12] K. Czarnecki and U. W. Eisenecker, "Components and generative programming," in *ESEC / SIGSOFT FSE*, ser. LNCS, vol. 1687. Springer, 1999, pp. 2–19.
[13] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, pp. 131–183, 1992.
[14] K. Czarnecki and S. Helsen, "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45, no. 3, pp. 621–646, 2006.
[15] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 31–39, 2008, see also http://eclipse.org/atl/.
[16] Object Management Group, "OCL Specification Version 2.0," http://www.omg.org/docs/ptc/05-06-06.pdf, 2005.
[17] BPM, http://www.bpmn.org/.
[18] R. David and H. Alla, *Petri Nets and Grafcet: Tools for Modelling Discrete Event Systems*. Prentice-Hall, Inc., 1992.
[19] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger, "Automated verification of model transformations based on visual contracts," *Autom. Softw. Eng.*, vol. 20, pp. 5–46, 2013.
[20] H. Zha, W. M. P. van der Aalst, J. Wang, L. Wen, and J. Sun, "Verifying workflow processes: a transformation-based approach," *Software and System Modeling*, vol. 10, no. 2, pp. 253–264, 2011.
[21] P. Wegner, "Varieties of reusability," in *Tutorial: Software Reusability*. Washington, D.C.: IEEE CS, 1987, pp. 24–38.
[22] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003, see also http://www.workflowpatterns.com/.
[23] M. Abadi and L. Cardelli, *A theory of objects*. Springer, 1996.
[24] M. Wimmer, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger, J. Sánchez Cuadrado, E. Guerra, and J. de Lara, "Reusing model transformations across heterogeneous metamodels," *ECEASST*, vol. 50, 2011.

[25] E. Guerra, J. de Lara, D. S. Kolovos, and R. F. Paige, "A visual specification language for model-to-model transformations," in *VL/HCC'10*. IEEE CS, 2010, pp. 119–126.

[26] E. Guerra, "Specification-driven test generation for model transformations," in *ICMT'12*, ser. LNCS, vol. 7307. Springer, 2012, pp. 40–55.

[27] P. Bonet, C. Llado, R. Puijaner, and W. Knottenbelt, "PIPE v2.5: A petri net tool for performance modelling," in *CLEI'07*, 2007, http://pipe2.sourceforge.net/.

[28] B. Alexe, W.-C. Tan, and Y. Velegrakis, "STBenchmark: towards a benchmark for mapping systems," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 230–244, 2008.

[29] N. Buezas, E. Guerra, J. de Lara, J. Martin, M. Monforte, F. Mori, E. Ogallar, O. Pérez, and J. Sánchez Cuadrado, "Umbra Designer: Graphical modelling for telephony services," in *ECMFA*, ser. LNCS, vol. 7307. Springer, 2013, pp. 179–191.

[30] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Soft. Eng.*, vol. 28, no. 1, pp. 4–17, 2002.

[31] H. Ma, W. Shao, L. Zhang, Z. Ma, and Y. Jiang, "Applying OO metrics to assess UML meta-models," in *UML'04*, ser. LNCS, vol. 3273. Springer, 2004, pp. 12–26.

[32] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel, "Meta-model pruning," in *MoDELS*, ser. LNCS, vol. 5795. Springer, 2009, pp. 32–46.

[33] I. Malavolta, H. Muccini, P. Pelliccione, and D. A. Tamburri, "Providing architectural languages and tools interoperability through model transformation technologies," *IEEE Trans. Software Eng.*, vol. 36, no. 1, pp. 119–140, 2010.

[34] Z. Hemel, L. C. Kats, D. M. Groenewegen, and E. Visser, "Code generation by model transformation: a case study in transformation modularity," *Software & Systems Modeling*, vol. 9, no. 3, pp. 375–402, 2010.

[35] A. Etien, A. Muller, T. Legrand, and R. F. Paige, "Localized model transformations for building large-scale transformations," *Software and System Modeling*, vol. in press, 2013.

[36] M. Voelter and I. Groher, "Handling variability in model transformations and generators," in *7th OOPSLA Workshop on Domain-Specific Modeling*, 2007.

[37] J. S. Cuadrado and J. G. Molina, "Approaches for model transformation reuse: Factorization and composition," in *Theory and Practice of Model Transformations*. Springer, 2008, pp. 168–182.

[38] R. García, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock, "A comparative study of language support for generic programming," *SIGPLAN Not.*, vol. 38, no. 11, pp. 115–134, 2003.

[39] B. Stroustrup, "The C++0x remove concepts decision," *Dr.Dobbs*, 2009, http://www.ddj.com/cpp/218600111.

[40] S. P. Jones, Ed., *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, 2002. [Online]. Available: http://haskell.org/definition/haskell98-report.pdf

[41] B. C. Oliveira, A. Moors, and M. Odersky, "Type classes as objects and implicits," *SIGPLAN Not.*, vol. 45, no. 10, pp. 341–360, 2010.

[42] T. Skotiniotis, J. Palm, and K. J. Lieberherr, "Demeter interfaces: Adaptive programming without surprises," in *ECOOP*, ser. LNCS, vol. 4067. Springer, 2006, pp. 477–500.

[43] K. Bak, K. Czarnecki, and A. Wasowski, "Feature and meta-models in Clafer: Mixed, specialized, and coupled," in *SLE'10*, ser. LNCS, vol. 6563. Springer, 2010, pp. 102–122.

[44] M. Herrmannsdoerfer, S. Vermolen, and G. Wachsmuth, "An extensive catalog of operators for the coupled evolution of metamodels and models," in *SLE*, ser. LNCS, vol. 6563. Springer, 2010, pp. 163–182.

[45] D. D. Ruscio, L. Iovino, and A. Pierantonio, "Coupled evolution in model-driven engineering," *IEEE Software*, vol. 29, no. 6, pp. 78–84, 2012.

[46] G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *ECOOP*, ser. LNCS, vol. 4609. Springer, 2007, pp. 600–624.

[47] D. D. Ruscio, L. Iovino, and A. Pierantonio, "A methodological approach for the coupled evolution of metamodels and atl transformations," in *ICMT*, ser. LNCS, vol. 7909. Springer, 2013, pp. 60–75.

[48] S. Vermolen, G. Wachsmuth, and E. Visser, "Reconstructing complex metamodel evolution," in *SLE*, ser. LNCS, vol. 6940. Springer, 2011, pp. 201–221.

[49] K.-K. Lau and Z. Wang, "Software component models," *IEEE Trans. Soft. Eng.*, vol. 33, no. 10, pp. 709–724, 2007.

[50] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. Springer, Monographs in Theor. Comp. Sci., 1990.

[51] G. Engels, R. Heckel, and A. Cherchago, "Flexible interconnection of graph transformation modules," in *Formal Methods in Software and Systems Modeling*, ser. LNCS, vol. 3393. Springer, 2005, pp. 38–63.

[52] R. Bendraou, P. Desfray, M.-P. Gervais, and A. Muller, "MDA tool components: a proposal for packaging know-how in Model Driven Development," *Software and System Modeling*, vol. 7, no. 3, pp. 329–343, 2008.

[53] L. Favre and L. Martinez, "Formalizing MDA components," in *Reuse of Off-the-Shelf Components*, ser. LNCS, vol. 4039. Springer, 2006, pp. 326–339.

[54] J.-M. Mottu, B. Baudry, and Y. Traon, "Reusable MDA components: A testing-for-trust approach," in *MoDELS'06*, ser. LNCS, vol. 4199. Springer, 2006, pp. 589–603.

[55] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, W. Schwinger, D. Kolovos, R. Paige, M. Lauder, A. Schürr, and D. Wagelaar, "Surveying rule inheritance in model-to-model transformation languages," *JOT*, vol. 11, no. 2, pp. 3:1–46, 2012.

[56] D. Wagelaar, R. V. D. Straeten, and D. Deridder, "Module superimposition: a composition technique for rule-based model transformation languages," *Software and System Modeling*, vol. 9, no. 3, pp. 285–309, 2010.

[57] J. Sánchez Cuadrado and J. G. Molina, "Modularization of model transformations through a phasing mechanism," *Software and System Modeling*, vol. 8, no. 3, pp. 325–345, 2009.

[58] E. Kalnina, A. Kalnins, E. Celms, and A. Sostaks, "Graphical template language for transformation synthesis," in *SLE'09*, ser. LNCS, vol. 5969. Springer, 2010, pp. 244–253.

[59] D. Varró and A. Pataricza, "Generic and meta-transformations for model transformation engineering," in *UML'04*, ser. LNCS, vol. 3273. Springer, 2004, pp. 290–304.

[60] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schoenboeck, and W. Schwinger, "Surviving the heterogeneity jungle with composite mapping operators," in *ICMT'10*, ser. LNCS. Springer, 2010, pp. 260–275.

[61] J. Hannemann, G. C. Murphy, and G. Kiczales, "Role-based refactoring of crosscutting concerns," in *AOSD'05*. ACM, 2005, pp. 135–146.

[62] J. Reimann, M. Seifert, and U. Aßmann, "Role-based generic model refactoring," in *MoDELS'10*, ser. LNCS, vol. 6395. Springer, 2010, pp. 78–92.

[63] S. Sen, N. Moha, V. Mahé, O. Barais, B. Baudry, and J.-M. Jézéquel, "Reusable model transformations," *Software and System Modeling*, vol. 11, no. 1, pp. 111–125, 2010.

[64] J. E. Rivera, D. Ruiz-Gonzalez, F. Lopez-Romero, J. Bautista, and A. Vallecillo, "Orchestrating ATL model transformations," in *MtATL 2009*, 2009, pp. 34–46.

[65] B. Vanhooff, D. Ayed, S. V. Baelen, W. Joosen, and Y. Berbers, "UniTI: A unified transformation infrastructure," in *MoDELS'07*, ser. LNCS, vol. 4735, 2007, pp. 31–45.

[66] A. Kleppe, "MCC: A model transformation environment," in *ECMDA-FA'06*, ser. LNCS, vol. 4066. Springer, 2006, pp. 173–187.

[67] A. Yie, R. Casallas, D. Deridder, and D. Wagelaar, "Realizing model transformation chain interoperability," *Software and System Modeling*, vol. 11, no. 1, pp. 55–75, 2011.

[68] A. Balogh, G. Bergmann, G. Csertán, L. Gönczy, Á. Horváth, I. Majzik, A. Pataricza, B. Polgár, I. Ráth, D. Varró, and G. Varró, "Workflow-driven tool integration using model transformations," in *Graph Transformations and Model-Driven Engineering*, ser. LNCS, vol. 5765. Springer, 2010, pp. 224–248.

[69] C. Alvarez and R. Casallas, "MTC Flow: A tool to design, develop and deploy model transformation chains," in *ACadeMics Tooling with Eclipse Workshop*. ACM, 2013, see also http://www.mtcflow.com/.