Softw Syst Model (2013) 12:201–219
DOI 10.1007/s10270-011-0203-2

REGULAR PAPER

Genericity for model management operations

Louis Rose · Esther Guerra · Juan de Lara · Anne Etien · Dimitris Kolovos · Richard Paige

Received: 19 November 2010 / Revised: 6 February 2011 / Accepted: 17 April 2011 / Published online: 1 June 2011
© Springer-Verlag 2011

Abstract Models are the core assets in model-driven engineering, and are therefore subject to all kind of manipulations, such as refactorings, animations, transformations into other languages, comparisons and merging. This set of model-related activities is known as *model management*. Even though many languages and approaches have been proposed for model management, most of them are type-centric, specific to concrete meta-models, and hence leading to specifications with a low level of abstraction and difficult to be reused in practice. In this paper, we introduce ideas from *generic programming* into model management to raise the level of abstraction of the specifications of model manipulations and facilitate their reuse. In particular we adopt *generic meta-model concepts* as an intermediate, abstract meta-model over which model management specifications are defined. Such meta-model concepts are mapped to concrete meta-models, so that specifications can be applied to families of meta-models satisfying the concept requirements. As a proof of concept, we show the implementation of these ideas using the Eclipse Modeling Framework and the Epsilon family of languages for model management.

Keywords Model management · Genericity · Reusability · Epsilon · Eclipse Modelling Framework

Communicated by Dr. Gabor Karsai.

L. Rose · D. Kolovos · R. Paige
University of York, York, UK
e-mail: louis@cs.york.ac.uk

D. Kolovos
e-mail: dkolovos@cs.york.ac.uk

R. Paige
e-mail: paige@cs.york.ac.uk

E. Guerra · J. de Lara (✉)
Universidad Autónoma de Madrid, Madrid, Spain
e-mail: Juan.deLara@uam.es

E. Guerra
e-mail: Esther.Guerra@uam.es

A. Etien
INRIA Lille Nord Europe, Université Lille 1, Lille, France
e-mail: Anne.Etien@lifl.fr

Present Address:
L. Rose
Department of Computer Science, University of York, York, UK

1 Introduction

Model-driven engineering (MDE) promotes models as the principal assets in the development process. During a typical MDE process, models are subjected to all kind of manipulations such as refactorings, animations or simulations, transformations into other modelling languages, comparisons, merging, and code for all or part of the final application is generated from models as well. This set of operations on models is referred to as model management [1].

In MDE, the syntax of models is usually defined through a meta-model that specifies the linguistic concepts offered to the users. Hence, a meta-model describes a set of valid models, and we say that a model in such set *conforms* to its meta-model. In this way, a model may use and instantiate the types (i.e. meta-classes and meta-associations) defined in the meta-model.

Typically, model management operations [1,3,6] are defined in terms of specific types from a particular meta-model, and are then applied to some model conformant to this meta-model. This situation is far from ideal because the defined operations are tightly tied to a concrete meta-model,

inhibiting their reuse with other, potentially similar meta-models. For example, an operation that calculates the transitive closure of a relation usually has to be defined for each different meta-model and relation. In the context of a particular meta-model, a single operation can be used to compute the transitive closure of different relations by, for instance, defining a more abstract operation using inheritance. However, no such re-use is typically possible for relations belonging to different meta-models. Hence, nowadays there is a general lack of powerful abstraction mechanisms in model management, probably caused by the inherent type-centric nature of meta-modelling. Our goal in this paper is to advance towards the solution of this problem.

Generic programming [26] allows the specification of algorithms and data structures to abstract away from concrete types. Generic algorithms are defined over generic types for which certain requirements may be demanded, like the definition of a certain method or their applicability as arguments for a certain operation. These requirements for types are known as *concepts* [27,48]. Generic programming has been successfully used for creating libraries of highly reusable, flexible, extensible and efficient algorithms, like those of the Standard Template Library (STL) [47] and Boost [5] for C++.

In this paper we bring elements from generic programming into MDE with the purpose of increasing the level of abstraction and improving the reusability of model management operations. For this purpose we propose *meta-model concepts* (sometimes called just *concepts*) for describing the requirements needed by a certain model operation. Concepts can be mapped or *bound* to concrete meta-models. In this way, the original operation can be executed on instances of any of the bound meta-models, hence improving its generality. We also report on an implementation of these ideas using Epsilon [23], a family of languages for model management that allows the specification of: model-to-model transformations with Epsilon Transformation Language (ETL) [36], arbitrary in-place operations with Epsilon Object Language (EOL) [33], user-directed refactorings with Epsilon Wizard Language (EWL) [37], code generators with Epsilon Generation Language (EGL) [43], comparison of models with ECL [32], model merging with EML [34], and model migration with Flock [42]. Through a number of examples we will demonstrate the benefits and potential that genericity brings into model management and MDE. Although these examples rely on our particular implementation on top of the Eclipse Modeling Framework (EMF) [46] and Epsilon, the approach is easily applicable to other model management languages and environments as well [3,6,31].

The rest of the paper is organized as follows. Section 2 recalls the main elements of genericity in modelling and programming, especially the so-called generic *concepts*. Section 3 introduces the Epsilon languages for model management. Section 4 introduces our approach, based on concepts, to genericity in MDE. Section 5 presents tool support for generic model management using Epsilon, which is illustrated in Sect. 6 through a series of examples. Section 7 compares with related research and Sect. 8 concludes.

2 Genericity in modelling and programming

In this section we review the main elements of genericity in both programming and modelling, in the former paying attention to so-called *concepts*, and in the latter as realized in the UML standard [41].

2.1 Generic programming

Genericity [26] is a programming paradigm found in many languages like C++, Haskell, Eiffel or Java. Its goal is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction. It involves expressing algorithms with minimal assumptions about data abstractions, as well as generalizing concrete algorithms without losing efficiency [26]. It promotes a paradigm shift from types to algorithms' requirements, so that even unrelated types may fulfill those requirements, hence making algorithms more general and reusable [27,48].

Genericity is realized through function or class templates in many programming languages, like C++ or Java. Templates declare a number of type parameters for the given code snippet, and later can be instantiated with concrete types. They can also define requirements on the type parameters, so that only those concrete types fulfilling the requirements are considered valid. The set of requirements to be fulfilled by a type is termed a *concept* [27] in the generic programming community. Concepts usually declare the signature of the operations a given type needs to support to be acceptable in a template. Hence, templates refer to concepts to declare the requirements of their type parameters.

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

Taking as inspiration concepts as defined in generic programming, in Sect. 4 we introduce *meta-model concepts* that will be used by model management operations enabling their application to any meta-model satisfying the concepts. Next, we review the approach to genericity taken in modelling, especially in the UML.

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

```
template <typename T> requires LessThanComp<T>
  T min(T x, T y) {
    return y < x ? y : x;
  }

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

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

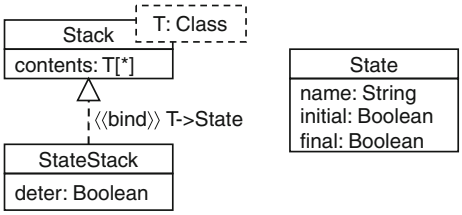


Fig. 1 A UML class template example

2.2 Generic modelling

Genericity has also been adopted in modelling, most notably in the UML 2 standard [41], in the form of classifier (e.g., class, component), package, collaboration and operation templates. These templates are provided with a list of formal parameters representing classifiers, values or features (i.e., properties and operations). Then, a template binding specifies the substitution of actual parameters for the formal parameters of the template. The presence of a template binding relationship has the same semantics as if the contents of the template were copied into the bound element, substituting the formal template parameters by the corresponding actual parameters in the binding [41].

For instance, Fig. 1 shows an example of a UML class template. The template is named *Stack* and receives a class parameter *T*. The parameter is instantiated by class *StateStack*, binding the parameter *T* to class *State*, which in addition declares additional features.

While UML 2 provides genericity elements that extend and adapt those found in programming languages like Java or C++ [2], it still lacks support to express requirements for the formal parameters in a non-intrusive way, as supported by the notion of *concept* presented in the previous section. In the context of UML 2, concepts would be a valuable means to express the requirements that parameter instantiations should fulfill in order for a template binding to be correct. Currently, this can be achieved only by requiring that some formal parameter conforms to a specific class, in a similar way as in Java, where a parameter may be required to implement a certain interface. However, if the template has several parameters, it is often not sufficient to demand requirements for each one of them in isolation, but for the set of parameters as a whole. As we will see, this is of crucial importance in our approach, as we need to express requirements of meta-models made of interrelated meta-classes.

Whereas the genericity provided by UML is mainly directed to *generic models*, in the sense that we obtain generic models by means of generic classifiers or packages, we are more interested in defining *generic behaviours*. In the context of meta-modelling, this means obtaining behaviours applicable to different meta-models, and therefore providing a means to support generic model management operations. The sequel introduces Epsilon, an extensible platform and family of languages for model management, which we will extend with genericity in Sects. 4 and 5.

3 Model management with Epsilon

Epsilon [23] (*E*xtensible *P*latform for *S*pecification of *I*ntegrated *L*anguages for m*O*del ma*N*agement) is a family of consistent and interoperable task-specific languages, which can be used to perform common MDE tasks such as code generation, model-to-model transformation, model validation, comparison, migration, merging and refactoring. Epsilon is modelling technology agnostic; though it is typically used to manage models represented using EMF, it can be used with alternative modelling technologies via its model connectivity framework, EMC. Developers can extend Epsilon with support for other modelling technologies by implementing a *driver* for EMC. Presently, Epsilon provides drivers for managing MDR models (Netbeans's metadata repository, an implementation of the JMI [50]), Z specifications, XML files and the METADEPTH [16] framework.

3.1 The Epsilon Object Language

The EOL [33] is at the core of Epsilon. It reuses the navigational mechanisms of OCL while adding support for other language features like multiple model access, statement sequencing, conditional and loop statements, and model modification capabilities. Similar to OCL, EOL permits the definition of operations for particular meta-classes. It can be used both as a stand-alone general model management language and as infrastructure on which one can build task-specific languages. Every other Epsilon language is built atop EOL.

As an example, Listing 2 shows the specification of some EOL operations to calculate the transitive closure of the inheritance relation, in the context of a particular object-oriented modelling language. While the operation *closure* on line 1 is defined globally, the other two operations are defined in the context of the *Class* meta-class, which has a reference

[2] It is an extension in the sense that, e.g., in C++ a template instantiation cannot define extra elements or have an operation as the actual parameter of a template.

```
1  operation closure() {
2    for (n in Class.all) {
3      var newCls : Set(Class) := n.closure();
4      if (newCls.size()>0)
5        for (m in newCls)
6          if (n.ancestors.excludes(m))
7            n.ancestors.add(m);
8    }
9  }
10
11 operation Class closure() : Set(Class) {
12   return self.closure(new Set(Class));
13 }
14
15 operation Class closure(set:Set(Class)):Set(Class){
16   var reachable: Set(Class) := new Set(Class);
17   for (x in self.ancestors)
18     if (set.excludes(x)) {
19       reachable.add(x);
20       set.add(x);
21     }
22   for (y in reachable)
23     y.closure(set);
24   return set;
25 }
```

Listing 2 Transitive closure of the inheritance relation

ancestors to its direct parents in the hierarchy (see lines 6 and 7). The *closure* method on line 15 calculates recursively the set of reachable *Class* instances through the *ancestor* reference. The global operation *closure* on line 1 iterates on all *Class* instances, invoking the auxiliary *closure* operation on line 11, which in turn calls the operation on line 15.

Even though the calculation of the transitive closure of an association is a common task occurring in the context of many meta-models, Listing 2 is tied to a *specific* meta-model and meta-class. Therefore, a mechanism to decouple operations from concrete meta-models and types would increase the reusability of model management operations and enable the construction of libraries of generic model management operations.

3.2 Further Epsilon languages

As stated earlier, while EOL can be used as a stand-alone language, it is also re-used in every task-specific language provided by Epsilon. For example, the ETL [36] is a rule-based model-to-model transformation language that re-uses EOL to specify guard statements, the body of transformation rules and user-defined operations. ETL supports transforming many input to many output models, rule inheritance, lazy and greedy rules, and the ability to query and modify both input and output models.

Listing 3 shows an exemplar transformation comprising one rule, which transforms a tree structure into a graph. ETL creates traces between source and target elements, which can be implicitly navigated using the '::=' operator. For example, in line 8, the operator assigns to *e.source* the `Node` in which *t.parent* has been transformed. Hence, this operator

```
1  rule Tree2Node
2    transform t : Tree!Tree
3    to n : Graph!Node {
4
5    n.name := t.label;
6    if (t.parent.isDefined()) {
7      var e : new Graph!Edge;
8      e.source ::= t.parent;
9      e.target := n;
10   }
11 }
```

Listing 3 Transforming a tree into a graph with ETL

traverses the traces created when transforming source elements, or triggers the execution of suitable rules to create the target element if the trace is not already created. The exclamation marks in lines 2 and 3 are used to separate a type name (e.g., `Node`) from the model name where its instances are to be sought (e.g., `Graph`).

Developers of model-to-model transformations could benefit from libraries of generic transformation patterns, which could be reused to build new transformations. However, again, the transformation of Listing 3 makes use of particular meta-models and types, which hinders its direct reuse.

Other languages of the Epsilon family that could benefit from genericity mechanisms include:

- The Epsilon Validation Language (EVL) [35], a model validation language that supports intra- and inter-model consistency checking, constraint dependency management and recovery actions. The availability of genericity mechanisms would enable the definition of libraries of common constraints (e.g., to check the absence of loops through a given type of link) which could be directly reused for varying meta-models.
- The EGL [43], a template-based model-to-text language for generating code, documentation and other textual artefacts from models. As we will see, generic code generators facilitate text generation from meta-models that share characteristics.
- The EWL [37], a language tailored to interactive in-place model transformations on model elements selected by the user. Decoupling EWL transformations from concrete meta-models would enable, e.g., the creation and application of libraries of generic refactorings.

Based on the notion of generic *concept* presented in Sect. 2.1, the remainder of this article presents a mechanism to increase the abstraction of model management operations, making them highly reusable.

4 Generic model management operations through meta-model concepts

4.1 Meta-model concepts

A *meta-model concept* is a meta-model that includes the structural elements needed by some model management operation (like an in-place model transformation or a model-to-text transformation), so that the operation can be defined over the concept and use its types. Concepts can be mapped to any meta-model that satisfies its requirements, therefore operations defined over concepts become generic and highly reusable even across unrelated meta-models.

This approach is shown in Fig. 2: a concept C may be bound to several meta-models, and a generic operation using the concept is applicable to any instance of the bound meta-models, hence improving its reusability. Indeed, we support a more general situation than the depicted one, as a generic operation may use several concepts, and the same concept may be used by different operations. This reusability of concepts is one of the key points and advantages of our approach.

Even though there are many ways to specify requirements for meta-models [20,27], the simplest and more uniform way is to model such requirements as meta-models as well. Thus, in our approach, a concept is a meta-model C of which its elements (meta-classes, meta-associations and attributes[3]) are to be interpreted as variables to be mapped to elements of a given particular meta-model.

4.2 Binding concepts

Mapping or *binding* a concept C to a specific meta-model MM is conceptually similar to defining a function $bind: C \rightarrow MM$ from the concept to the meta-model. This function establishes a correspondence between:

1. Each meta-class $c \in C$ and some meta-class $c' = bind(c) \in MM$,
2. Each meta-association $a \in C$ and some meta-association $a' = bind(a) \in MM$,
3. Each attribute f of meta-class $c \in C$ and some attribute $f' = bind(f)$ of meta-class $b \in MM$, where $b \in ancestors(bind(c))$.

We have used a function *ancestors* that returns the set of ancestors of an element (with respect to inheritance) including the element itself. In a similar way, we will refer to the set of descendants of a meta-class c (including itself) as $descendants(c)$. The condition for attributes allows an attribute of a meta-class in the concept to be mapped to attributes of parent meta-classes of the bound meta-class in the meta-model. This situation is illustrated in Fig. 3, where we bind attribute $f2$ to an attribute of c', and $f1$ to an attribute of the parent meta-class of c'.

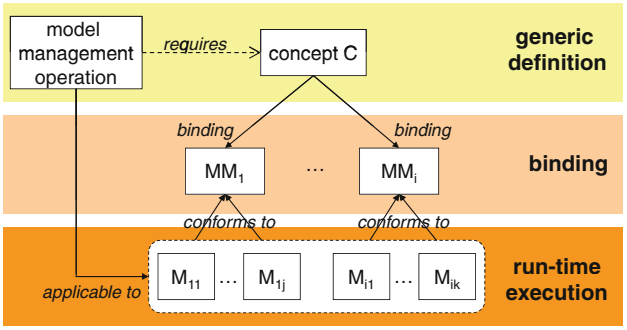


Fig. 2 Defining generic operations over concepts

Our binding function does not require mapping the inheritance relations in the concept with other inheritance relations in the meta-model. In this sense, our notion of binding is semantic and not purely syntactic, as it interprets the meaning of inheritance relations.

4.3 Compatibility conditions for the binding

In addition to establishing a correspondence between meta-classes, meta-associations and attributes of the concept and the specific meta-model, the binding function demands additional conditions. These ensure compatibility of the source and target of the mapped meta-associations, the type of attributes, and cardinalities as follows:

1. The source and target of meta-associations should be compatible: $\forall a \in C$, $bind(src(a)) \in descendants(src(bind(a)))$ and $bind(tar(a)) \in descendants(tar(bind(a)))$, where src and tar are functions returning the source and target meta-classes of association a. For simplicity we assume binary associations. We use functions src and tar to return the two connected meta-classes, but despite their name, they make no assumptions on navigability or direction. With respect to decorations of association ends:
   (a) If the association end of $src(a)$ is navigable, then so must be $src(bind(a))$ (and similar for tar). On the contrary, if the association end of $src(a)$ is not navigable, then $src(bind(a))$ can be navigable or not (and similar for tar).
   (b) If the association end of $src(a)$ is a composition, then it should be mapped to a composition in the meta-model. Conversely, if the association end of $src(a)$ is not a composition, then the mapped end in the meta-model should not be a composition.

[3] We refer to attributes with primitive data types, like integer and string.

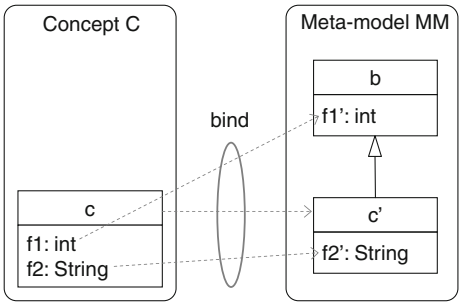


Fig. 3 Example of valid binding for attributes

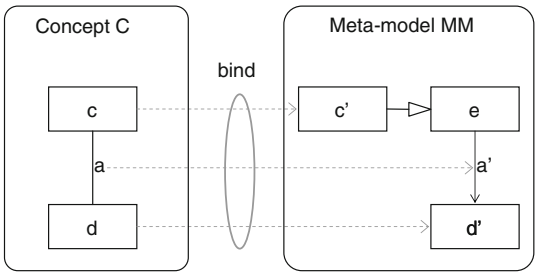


Fig. 4 Example of valid binding for an association

(c) If the association end of $src(a)$ is not required to be ordered, then it can be mapped to an ordered association end in the meta-model or not. If $src(a)$ is required to be ordered, then $bind(src(a))$ should be ordered as well. This restriction also applies to multi-valued attributes.

(d) If the association end of $src(a)$ is labelled as unique, then $bind(src(a))$ should be unique as well. If $src(a)$ is not demanded to be unique, then $bind(src(a))$ cannot be unique. This restriction also applies to multi-valued attributes.

2. The type of attributes should be compatible: $\forall f \in C$, $type(bind(f)) \trianglelefteq type(f)$, where $type$ is a function returning the type of an attribute, and $\trianglelefteq$ is the subtype relationship. If f is not mandatory in C, then $bind(f)$ cannot be mandatory in MM. If f is mandatory in C, then $bind(f)$ has to be mandatory in MM.
3. The cardinality of association ends and multi-valued attributes should be preserved.

The first condition checks that, in the concept, an association a stemming from (or ending at) a certain meta-class c can be bound to an association $bind(a)$ in any ancestor of the meta-class to which c is bound. This situation is illustrated in Fig. 4. Moreover, if a concept does not require navigability for some association end, then the mapped association end can be navigable or not. This is so as our concepts are meant to define the requirements that need to be fulfilled by a meta-model to be accepted by a generic operation that uses the concept. The meta-model can incorporate additional properties to the ones defined by the concept, but it cannot be more restrictive than this. In this way, the binding ensures that any operation that can be performed on a hypothetical instance of the concept can be performed on any instance of a valid bound meta-model as well.[4] This issue is illustrated in Fig. 4, where one association end in the meta-model is navigable but it is not in the concept. The contrary would not be allowed.

[4] We will see in a moment that this condition is actually too demanding to be useful in practice.

With respect to composition, its semantics prevent mapping a composition to a non-composition and vice-versa. The reason is that deleting the source of a composition deletes the connected children as well, which can cause unexpected situations. On the contrary, children are not deleted for simple references. The binding should preserve uniqueness constraints. In particular, if an association end is not required to be *unique* in the concept, we cannot bind it to an association end labelled as unique in the meta-model. This is so, as some operation specified at the concept level may fail when applied to meta-model instances. Finally, regarding *order* constraints, if we do not require order for an association end in the concept, we may have it or not in the meta-model, as this does not affect the execution of operations.

For the same reason, condition 2 demands the type of a mapped attribute (assuming primitive data types) to be a subtype of the type of the attribute in the concept.

The third condition on multiplicities of association ends and multi-valued attributes is required for the stricter case, which guarantees that any operation that can be applied on a concept, can also be applied in a meta-model bound to the concept. Otherwise we could have undesired situations. For instance, suppose we permit mappings to collections with a wider interval, and thus we map an association end e with cardinality 3..6 to an association end $bind(e)$ with cardinality 1..7. In this case, an initial model with seven elements in $bind(e)$ would be problematic, as the operation would not be able to determine which six elements among the seven to choose. Similarly, an initial model with one element may be an incorrect initial state for the operation. Conversely, suppose we permit mappings to collections with a narrower interval, and thus we map an association end with cardinality 3..6 to another with cardinality 4..5. In this case, the binding would not respect the basic condition that operations which are applicable to the concept should be applicable to the mapped meta-models. The reason is that a set with four elements could be added two elements in the concept but not in the mapped set, as this latter would admit five elements at most. Similarly, a set with four elements could be deleted one element in the concept but not in the mapped set.

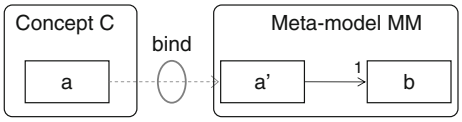


Fig. 5 A binding that does not respect subtyping

However, demanding the meta-models to define the same cardinalities for association ends and attributes as the concepts may be too strict. For this reason, in practice, we relax this third condition as follows:

1. We permit mapping collections in a concept to collections with a different interval, if the cardinality in the initial run-time model is within the bounds specified by the concept.
2. We permit mapping collections in a concept to collections with a higher lower bound, if the generic operation does not delete elements from the collection.
3. We permit mapping collections in a concept to collections with a lower upper bound, if the generic operation does not add elements to the collection.

Finally, we allow the binding function to be non-injective, obtaining additional flexibility. Hence, two elements a and b of the concept C can be mapped to the same element $c = bind(a) = bind(b) \in MM$ whenever the compatibility conditions are satisfied. Practical implementations of the binding can control whether such non-injectivity is allowed or not.

This binding process is in line with the classical algebraic specification approach of representing models and meta-models as algebraic structures, and defining relations between them as homomorphisms (functions between algebraic structures that preserve certain aspects, structural in our case) [22], called clan-morphisms in [15,28]. Further formal treatment of concepts and bindings is out of the scope of this paper.

4.4 Controlling the binding context: expressing concept usage

The binding induces a kind of subtyping relationship between the concept (supertype) and the bound meta-model (subtype) [38,45]. Subtyping implies safe substitution of a supertype by a subtype: any operation that is applicable to a supertype model should be applicable to the subtype model. However, this condition is very strong and often too restrictive, as the previous discussion on collections showed, and the next scenario will illustrate.

Figure 5 shows a valid binding according to our previous definition, which, however, does not respect subtyping. The reason is that one could specify a generic operation that creates instances of a. This operation is allowed at the concept level and always succeeds. However, when applied to an instance of the bound meta-model, it will always fail (in the sense that it will lead to an incorrect model not conformant to meta-model MM). This is so, as each time the generic operation creates an a object, an a' is created instead, leading to an inconsistent model, as each a' instance should be connected to a b instance according to the meta-model. On the contrary, if the generic operation does not create a instances, the binding should be allowed. Similarly, if the association in the meta-model MM would have one to the side of meta-class a, then deleting an a instance would be problematic because some b object would be left without connected as, breaking the cardinality constraint.

There are several ways to address this problem. For example, similar to [17], we could provide the concept with OCL constraints to be evaluated on the particular meta-model when the binding is established, in a similar way as OCL is defined on meta-models and evaluated on models. This provides fine control on the binding, but it makes the definition of concepts more complex. Another possibility is to provide bindings with negative graphical patterns which cannot be found on the bound meta-model. This is higher level, but again requires defining such patterns each time a concept is developed. Instead, we provide a simpler, higher level, pragmatic solution. Our proposal is to decorate each concept element with the *operations* (create or delete) that the generic management operation requires. We call these *usage decorations*, and are easily realized through stereotypes or annotations. In this way, if in the previous example of Fig. 5 the operation creates instances of a, the a meta-class in the concept should be decorated with *create*. This has the effect to automatically restrict the binding function in such a way to prohibit the mapping depicted in Fig. 5. Multi-valued attributes can also be decorated with usage contexts, hence allowing a finer control of the bindings of the multiplicity intervals, as discussed in previous section. Please note that usage decorations have the effect of forbidding certain bindings, as we assume that the generic operations are used as they are provided. Another, less restrictive possibility is to permit such binding, but request the user to extend the operation with appropriate actions (e.g., connect each created a' instance with a b instance). In any case, this possibility can be realized by not decorating the concept elements.

Altogether, usage decorations have the following effects on the binding function *bind*:

1. A meta-class $c \in C$ decorated with *create* cannot be bound to a meta-class $d = bind(c) \in MM$ if d (or any direct or indirect ancestor) is connected with some other

meta-class $e \in MM$ through an association a with cardinality 1 or higher to the side of e such that $\nexists a' \in C$ with $bind(a') = a$.

2. A meta-class $c \in C$ decorated with *delete* cannot be bound to a meta-class $d = bind(c) \in MM$ if d (or any direct or indirect ancestor or descendant) is connected with some other meta-class $e \in MM$ through an association a with cardinality 1 or higher to the side of d such that $\nexists a' \in C$ with $bind(a') = a$.
3. A multi-valued attribute f that is decorated with *delete* in a concept cannot be mapped to another multi-valued attribute $f' = bind(f)$ with higher lower bound in the meta-model.
4. A multi-valued attribute f that is decorated with *create* in a concept cannot be mapped to another multi-valued attribute $f' = bind(f)$ with a lower upper bound in the meta-model.

Regarding the decorations for meta-classes (first two conditions), we only forbid the connection of $d = bind(c)$ with other meta-classes through meta-associations that are not bound from a meta-association in the concept. This is so because, if they are bound, the generic operation has to take care of adding/deleting such associations (and the objects they connect) when adding or deleting c's. Otherwise, the operation would be incorrect for the very concept. Regarding deletions, we adopt a conservative approach and permit deleting d instances only if they are not connected to any e instance. A meta-model with a mandatory connection to the side of d allows unconnected d instances, which may be deleted without problems. However, as this depends on the initial model, the simplest solution is to forbid such binding for the meta-model. Finally, in line with the UML and EMF conventions, we assume that if an instance is deleted, all its references are deleted as well.

Please note that concepts are manually built by the designers of the generic operations. The designer includes in the concept all elements (meta-classes, meta-associations, attributes) that he considers essential for an operation to be executable. However, the designer may make mistakes, e.g., by forgetting a usage decoration, or by not including some necessary association. This incorrect concept may allow a binding, which may make the operation fail when executed on a concrete meta-model instance. Therefore we have type-safety only up to the *correctness* of the concept. An automatic formal check of the correctness of a concept given an operation, or an automatic derivation of the usage indications given an operation could be done if we use a formal language to express the operations, like graph transformations. As we will see in the following, we use the Epsilon languages to express the model-management operations, so this issue is left for future work.

5 Tool support

A reference implementation of the proposal outlined above has been constructed atop the EMF and the Epsilon platform. The former is arguably the most widely used MDE modelling framework today, while the latter contributes a family of model management languages, and was described in Sect. 3. In particular, for this implementation we have profited from the EMC layer, which underpins the Epsilon languages, decoupling them from the specifics of a particular modelling technology.

In Epsilon, when executing a concrete (non-generic) model management operation, the user selects one model to fulfill each of the roles specified by the operation. A model-to-model transformation, e.g., might specify two roles, a source and a target. The model management operation specifies a set of types that each role must provide, typically by specifying a concrete meta-model to which each role conforms.

When executing a generic model management operation, the user must provide a binding for each meta-model concept used by the operation. Meta-model concepts are represented using a meta-model, possibly annotated with usage indications (create, delete). The user provides a binding between the concepts and the concrete meta-models when selecting a model to fulfil a particular role. The execution of a generic model management operation uses the binding to access concrete model elements via conceptual types. This process is depicted in Fig. 6.

The implementation described in this section contributes structures for performing the binding process and for accessing bound models in model management operations specified with Epsilon. The latter is achieved with an additional

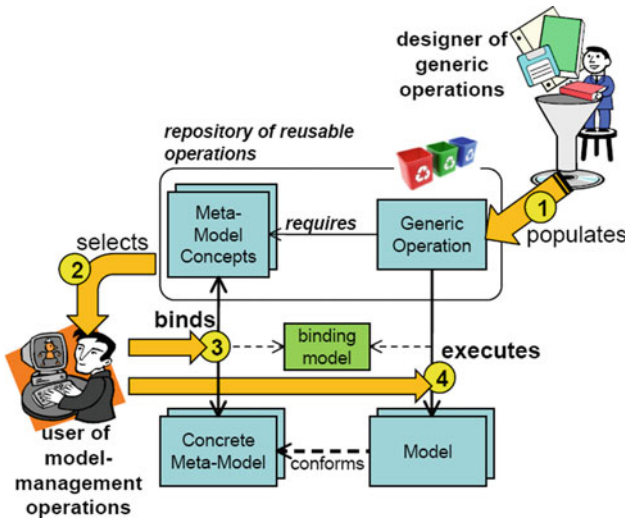


Fig. 6 Using generic operations

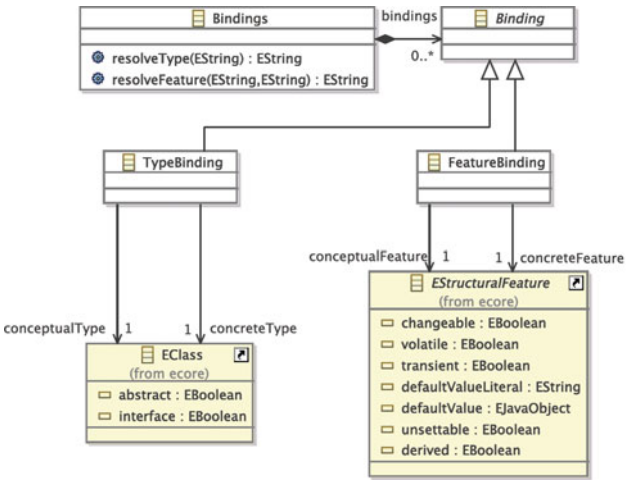


Fig. 7 Meta-model for specifying bindings

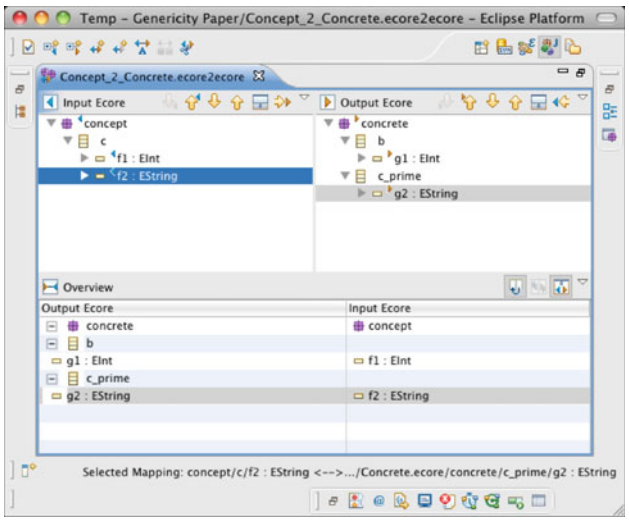


Fig. 8 Development tool for specifying bindings

driver for the EMC layer, and uses a binding model to resolve requests for conceptual types.

5.1 Binding meta-model and tools

The first phase of implementation involved the provision of a binding tool, allowing users to bind conceptual and concrete types. To facilitate interoperability between the binding tool and model management platforms such as Epsilon, we designed the binding meta-model shown in Fig. 7. The binding tool would produce models conforming to the binding meta-model, which would later be used during the execution of generic operations. Note how this meta-model is a simplification of the one provided in the UML 2 specification for template bindings [9,41], but where meta-class and feature bindings are distinguished to facilitate type resolution in an efficient manner.

We envisaged constructing a tool that implemented the bind function described in Sect. 4. However, EMF already provides a tool, which we term Ecore2Ecore, for creating mappings between two meta-models. Ecore2Ecore is typically used to support meta-model evolution, but here we re-purpose it for specifying bindings between a concept and a concrete meta-model. Ecore2Ecore is based on a simple mapping meta-model, which is similar to the one shown in Fig. 7. As an example, Fig. 8 shows the binding from Fig. 3 in Ecore2Ecore. A binding is saved as a model which is used during the execution of a generic model management operation.

Ecore2Ecore does not prevent from specifying mappings that are not valid bindings. To assist users in constructing valid bindings, we have provided a validation mechanism for checking the integrity of binding models. The validation is invoked using a toolbar item (shown in the top left-hand corner of Fig. 8) and checks, e.g., that the binding respects the create and delete usage annotations described in Sect. 4.4. The validation is specified with the EVL, and is executed on binding models. One of the constraints is shown in Listing 4 and checks condition 1 in Sect. 4.4 (all mandatory, concrete features must be bound when their containing type is bound to a conceptual type annotated with the create usage).

5.2 Accessing bound models with Epsilon

When executing a generic model management operation, bindings are used to determine concrete types from generic types. We term this process *resolution*. The way in which the resolution of generic types was implemented for Epsilon is now discussed.

The EMC layer is responsible for accessing and manipulating models. EMC provides an interface (`IModel`)—which is used by Epsilon languages to interact with models—and several drivers, each of which provides an implementation of the `IModel` interface for a particular modelling technology, such as EMF, MDR or XML. As such, an EMC driver is responsible for the way in which values are read from and written to a model, and for the way in which type information is used to access model elements.

For executing generic model management operations in Epsilon, we extended EMC with an additional driver that accesses EMF models using a binding model. The driver uses the binding to resolve types before accessing model elements. Listing 5 shows an extract from the code of the bound model driver. The `classForName` method is used to identify a meta-model type from a string, and is used by Epsilon programs to, for instance, access all instances of a particular meta-model type. In Listing 5, the `classForName` method uses an instance of `Bindings` (shown in Fig. 7) to attempt to resolve the type name. If there is no

```
context TypeBinding {
 constraint
   CreateImpliesThatAllMandatoryFeatsAreBound{
   guard: self.conceptualType.usage() == 'create'
   check: self.concreteType.unboundMandatoryFeats().isEmpty();
   message: "The conceptual type " +
          self.conceptualType.name + " " +
          "is annotated with 'create' so " +
          "the following features must be bound: " +
          self.concreteType.unboundMandatoryFeats().
          collect(sf|sf.name).concat(' ')
 }
}

operation EClass unboundMandatoryFeats() {
  return self.eAllStructuralFeatures
            .select(sf|sf.lowerbound <> 0 and
            sf.isUnbound());
}

operation EStructuralFeature isUnbound():Boolean{
  return FeatureBinding.all.forAll(m|m.concreteFeature<>self);
}
```

Listing 4 One constraint used to validate binding models

```
public class BoundEmfModel extends EmfModel implements IModel {

  private Bindings bindings = Bindings.NULL;

  @Override
  public EClass classForName(String rawType) throws EolModelElementTypeNotFoundException {
    String resolvedType=bindings.resolveType(rawType);

    return super.classForName(resolvedType == null ? rawType : resolvedType);
  }
}
```

Listing 5 Type resolution with the bound model driver

```
private String resolveProperty(EObject instance, String rawProperty) {
  EClass type = instance.eClass();
  Iterator<EClass> supertypes = getAllSuperTypes(type);
  String resolvedProperty;

  do {
    resolvedProperty = bindings.resolveFeature(type.getName(), rawProperty);
    type = supertypes.hasNext() ? supertypes.next(): null;
  } while (resolvedProperty==null && type!= null);

  return resolvedProperty==null ? rawProperty : resolvedProperty;
}
```

Listing 6 Property resolution with the bound model driver

binding for this type name, the call to resolve returns `null`, and the raw type name is used instead. This allows generic and concrete types to be mixed in the same model management operation.

The `IModel` interface provides methods for accessing the properties of a type, which the Epsilon language uses to read and write model values. The `resolveProperty` method shown in Listing 6 is part of the `BoundEmfModel` class, and is used to resolve generic property names using a binding model. Notice that resolution continues by navigating up the type hierarchy until a binding is found (using the loop on lines 6–9), allowing property bindings to be inherited as described in Sect. 4.

The `resolveFeature` method—used on line 7 of Listing 6—takes as arguments the names of a concrete type and a generic property. The former is required because, at present, Epsilon cannot store generic type information for model elements. Consequently, the `resolveFeature` method uses the inverse of the bind function to determine a generic type from the concrete type, and, subsequently, the

generic type and the generic property are used to resolve a concrete property. Because the inverse of the bind function is required to resolve properties, the current implementation prohibits non-injective mappings for generic properties.

The structures described in this section are used together to execute generic model management operations in Epsilon. First, the user binds their modelling concept (a meta-model) to a concrete meta-model using the Ecore2Ecore tools. The resulting binding model is used by Epsilon via the additional EMC driver, which performs resolution of generic types and properties. EMC decouples the Epsilon languages from the way in which models are accessed, and hence genericity could be added to all of the Epsilon languages with the addition of a single EMC driver.

6 Examples

In this section we illustrate our approach and supporting tool through several examples that demonstrate the benefits of genericity when applied to model management.

6.1 Generic behaviours

In our first example, we will assume that we would like to define a simulator for Petri nets [40]. Petri nets are a kind of automata that can be represented as a bipartite graph made of *places* and *transitions*. Places contain zero or more *tokens*, and can be connected to transitions and vice versa. A Petri net is simulated by firing enabled transitions, which delete and create tokens in the connected places. A transition is *enabled* if all its input places have at least one token each. An enabled transition may fire, and when it does, it removes one token from each input place and adds one token to each output place. In addition to execution, Petri nets have a large body of theoretical results enabling the analysis and verification of systems behaviour [40].

Many languages behave as Petri nets. For example, the semantics of workflow languages [8] and UML activity diagrams [41] are defined in terms of the token game. Also, domain specific languages like production systems [19] (where parts are consumed and produced by machines), communication systems [7] (where messages are sent and received by nodes), and data-flow languages [18] (where data are consumed and produced by processors) also share semantics with Petri nets.

For this reason, instead of defining a simulator for the concrete meta-model of Petri nets, we will build an abstract, generic simulator for its token-based semantics. For this purpose the first step will be to define a concept with all the requirements needed by the simulator. Afterwards, we will bind the concept to the meta-models of different languages

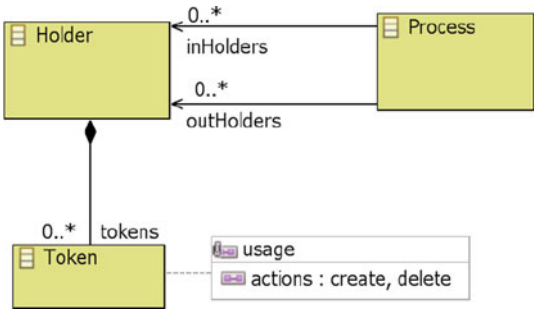


Fig. 9 *TokenHolder* concept

(among them the Petri nets meta-model), hence being able to apply the same simulator to a family of unrelated languages. This use of concepts permits classifying and defining in a uniform way the most typical semantics of modelling languages [7].

Fig. 9 shows the modelling concept that we use to define the semantics of languages similar to Petri nets, which we call *TokenHolder* semantics. The concept defines three metaclasses: *Process* (to model the active elements of the system, e.g., transitions in Petri nets), *Holder* (to model state elements, e.g., places in Petri nets), and *Token* (to model marks on holder elements, e.g., tokens in Petri nets). The input and output holders of a process, as well as the tokens of a holder, are modelled as associations. The *TokenHolder* semantics need the deletion and creation of tokens, and hence this metaclass is annotated with both *create* and *delete*.

Once the concept has been defined, we construct a generic EOL simulator that uses it. Listing 7 shows an excerpt of it. Lines 3–14 define the operation *main*, which is the entry point

```
1  main();
2
3  operation main() {
4    var maxStep : Integer := 100;
5    var numStep : Integer := 0;
6    var enabled : Set(Process) := getEnabled();
7    while (enabled.size()>0 and numStep<maxStep) {
8      var t := enabled.random();
9      t.fire();
10     writeState(numStep);
11     numStep := numStep+1;
12     enabled := getEnabled();
13   }
14 }
15
16 operation writeState(step: Integer) {...}
17
18 operation getEnabled() : Set(Process) {
19   var en : Set(Process);
20   for (t in Process.allInstances())
21     if (t.enabled()) en.add(t);
22   return en;
23 }
24
25 operation Process enabled() : Boolean {
26   return
27    self.inHolders->forAll(h|h.tokens.size()>0);
28 }
29
30 operation Process fire() : Boolean {...}
```

Listing 7 Generic *TokenHolder* simulator (excerpt)

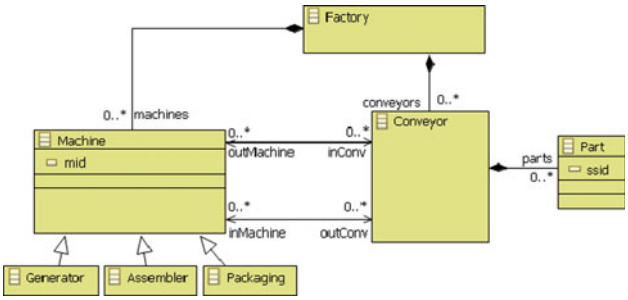


Fig. 10 A meta-model for production systems

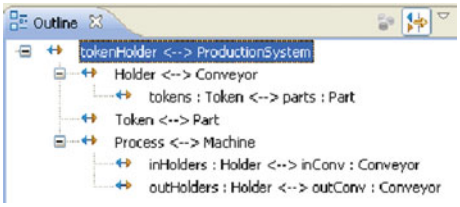


Fig. 11 Using the Ecore2Ecore tool to bind the *TokenHolder* concept to the production systems meta-model

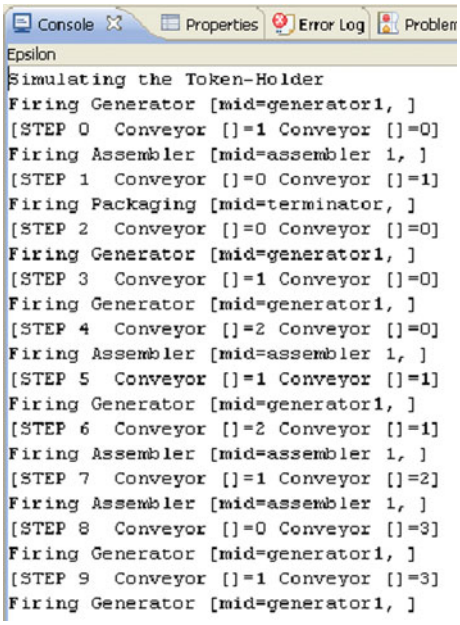


Fig. 12 Some steps in the simulation of a production system

of the simulation. Line 6 obtains the set of enabled processes, and then a loop iterates while there are enabled processes and up to a maximum number of iterations. Line 8 takes one enabled process and fires it. The listing also shows some auxiliary operations defined on the context of *Process*, such as *enabled* and *fire*. When the concept is bound, these operations will be added to the meta-class *Process* is bound to. In this way, the simulator provides these operations which do not need to be defined by the meta-models that will use the simulator.

Fig. 10 shows a meta-model for production systems. This is a domain specific modelling language to design factories made of different kinds of machines that consume and produce parts in input and output conveyors. Conveyors can hold arbitrary numbers of parts. The generator machines model a process that puts parts in the factory, in the connected conveyor. Assembler machines process the parts, and packaging machines take the parts out of the factory.

To use the generic simulator with our language for production systems, we bind the *TokenHolder* concept to the language meta-model of Fig. 10 as follows: *Conveyors* play the role of *Holders*, *Parts* play the role of *Tokens*, and *Machines* of *Processes*. With respect to the associations, the *inHolders* association end is bound to *inConv*, *outHolders* is bound to *outConv*, and *tokens* to *parts*. Figure 11 shows how this binding is specified using the Ecore2Ecore tool.

Once the binding has been established, we can execute the simulator on instances of the production system meta-model. Figure 12 shows the first steps in the simulation of a factory model made of one generator, connected to a conveyor, connected to an assembler machine, connected to another conveyor, connected to a packaging machine. The trace shows the states (number of parts in each conveyor) as well as the machine that produces each state change.

Altogether, this example showed us that by generalizing a simulator for Petri nets and defining it over a suitable concept, we were able to reuse the simulator with a different unrelated modelling language for production systems. Currently, we have reused this simulator also with a meta-model for Petri nets and another for communication systems.

6.2 Generic code generators

Once we have defined our *TokenHolder* concept, we can build many different model management operations which make use of it. Hence, concepts are also subjected to reuse. For instance, next we show the definition of a generic code generator built over the *TokenHolder* concept, and hence applicable to any meta-model for which we bind the concept. The generator synthesizes code for an external Petri net tool, so that we can reuse the existing tools and analysis methods of Petri nets to verify models of languages satisfying the *TokenHolder* concept.

Listing 8 shows an excerpt of this generic code generator, which has been written using EGL. The generator produces code in PNML format [29], which can be read by many tools like CPNTools [12] or PIPE [4].

Lines 1–3 of the listing print the XML file header. Then, lines 4–8 contain some EOL code, which is delimited by '[%' and '%]'. In particular, line 6 iterates over all of the instances of the type to which *Holder* is bound, and line 7 assigns them an auxiliary attribute *holdrID* made of the concatenation of the type name and an index. Then, the holder instance is written as a place (lines 9–16), together with its marking

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

Listing 8 Code generator for the *TokenHolder* concept (excerpt)

(i.e. the number of tokens it contains, line 14). Although not shown, the generator continues by writing the transitions and arcs.

Binding the *TokenHolder* concept to the production systems meta-model facilitates the application of the code generator to the factory example model described in the previous section. As a result, we obtain an XML file which we can load in PIPE for analysis. Figure 13 shows a snapshot of the tool being used to animate the model, as well as the reachability graph computed by the tool. The reachability graph is a graphical representation of the set of all reachable states of the net [40].

Altogether, this example showed us that the use of concepts enables writing generic code generators applicable to families of meta-models satisfying the given concept. We also learnt that the same concept can be used by different model management operations, enabling the development of libraries of concepts together with useful operations for them.

6.3 Generic refactorings

Using concepts, it is possible to define refactorings in a generic way. For example, for the purpose of analysis, there are a number of simplifications for Petri nets [40] which preserve some semantic properties (liveness, safety and boundedness). The simplified net has the advantage of being easier to analyse, as its set of reachable states is smaller. It is hence a common technique to apply these refactorings before the analysis is performed.

Using the *TokenHolder* concept, we can define such simplifications in a generic way, so that they become applicable for a family of meta-models. Listing 9 shows one such simplification. The refactoring, implemented by the generic EOL operation *FusionParallelProcess* on lines 1–8, merges two parallel processes if they have the same set of input and output holders (checked by operation *checkFPP* on lines 10–15). The operation could be executed as long as possible, to merge all possible parallel holders.

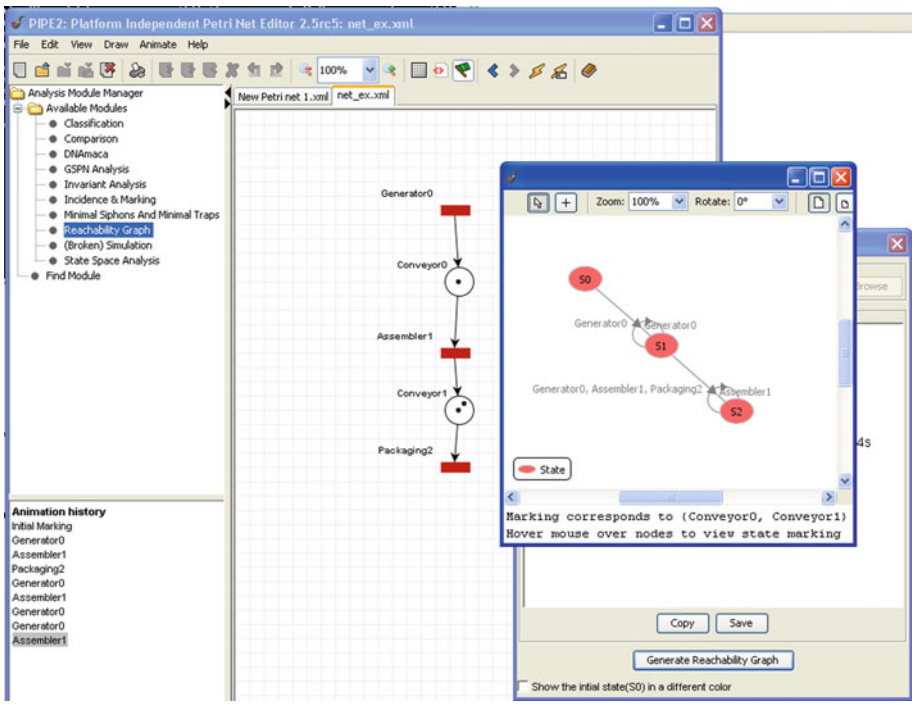


Fig. 13 Analysis of a production system model using PIPE

```
operation FusionParallelProcess(p1:Process,
            p2:Process): Boolean {
  -- Merges two parallel processes, if both have the
  -- same input and output holders
  if (not checkFPP(p1, p2)) return false;
  delete p1;
  return true;
}

operation checkFPP(p1: Process,
                   p2: Process): Boolean {
  -- Precondition for FusionParallelProcess
  return equalSets(p1.inHolders, p2.inHolders) and
         equalSets(p1.outHolders, p2.outHolders);
}

operation equalSets(s1: Set, s2: Set) : Boolean {
  -- checks if sets s1 and s2 have the same elements
  if (s1.size()<>s2.size()) return false;
  return s1.includesAll(s2);
}
```

Listing 9 Generic refactoring for the *TokenHolder* concept

Using EWL, refactorings can be invoked by the user from graphical and tree-based model editors. Listing 10 shows a generic EWL wizard used to merge two selected processes. The *guard* section in lines 2–16 checks whether the refactoring can be applied. In particular, line 5 checks that the selection of the user is actually a collection, and subsequent lines that it has size 2 (line 6), that it contains two processes (lines 7 and 8), and that they are parallel because they have the same input and output places and, therefore, can be merged (line 12). Then, the *do* section in line 20 performs the action of the refactoring. Please note that this refactoring requires a different set of usage annotations over the *TokenHolder* concept, as in this case processes can be deleted.

Hence, we have shown how generic concepts allow the definition of collections of refactorings in a meta-model independent way, hence becoming reusable. It is therefore feasible, e.g., to specify the well-known object-oriented refactorings of [25] in a generic way for their use with different meta-models.

```
wizard mergeParallel {
   guard {
     var m1 : Process;
     var m2 : Process;
     if (self.isKindOf(Collection)) {
       if (self.size()<>2) return false;
       if (not self.first().isKindOf(Process) or
           not self.last().isKindOf(Process))
          return false;
       m1 := self.first();
       m2 := self.last();
       if (not checkFPP(m1, m2)) return false;
       return true;
     }
     return false;
   }

   title : "Merges "+m1+" and "+m2

   do { delete m1; }
 }
```

Listing 10 Generic user-directed refactoring with EWL

6.4 Generic constraints and operations

As discussed in previous sections, genericity facilitates the construction of libraries of common operations in MDE, although not necessarily defined over the same concept. For instance, a very common operation in many model management tasks is calculating the transitive closure of a given relation. This operation is frequently performed as a pre-processing step in some model transformations where, e.g., inheritance hierarchies are flattened.

Listing 11 shows a generic operation *closure* defined over a concept made of a meta-class `A` and a reference `a` to itself. This operation iterates on all instances of the type `A` is bound to, adding the reachable instances to role `a` by calling the operation *closure* (line 10).

Compared with Listing 2—which implemented the same operations over a concrete meta-model—we can see that both listings are actually very similar, save for the different name of the meta-class and reference, and for the fact that Listing 2 can be executed only on instances of a particular meta-model, whereas Listing 11 is defined over a generic concept. In this way, this is an example of how operations for concrete meta-models can be generalized. Firstly, the designer builds concrete operations over concrete meta-models. Once he detects that it is useful to generalize them, he prunes the concrete meta-model, removing any accidental element that is not necessary for the given operation. The resulting pruned meta-model is hence the *concept*, which may have to be created anew, or reused if it is already available (e.g., like the *TokenHolder* concept).

In addition to generic operations, it is also possible to define libraries of generic constraints to be checked against specific meta-models. As an example, Listing 12 shows a

```
operation closure() {
  for (n in A.allInstances()) {
    var newAs : Set(A) := n.closure();
    if (newAs.size()>0)
      for (m in newAs)
        if (not n.a.includes(m)) n.a.add(m);
  }
}

operation A closure() : Set(A) {
  return self.closure(new Set(A));
}

operation A closure(set : Set(A)): Set(A) {
  var reachable: Set(A) := new Set(A);
  for (x in self.a)
    if (not set.includes(x)) {
      reachable.add(x);
      set.add(x);
    }
  for (y in reachable)
    y.closure(set);
  return set;
}
```

Listing 11 Generic operations to calculate the transitive closure of an association

```
1  import 'transitiveClosure.eol';
2  context A {
3    constraint isAcyclic {
4      check : not self.closure().includes(self)
5    }
6  }
```

Listing 12 Generic EVL constraint to check acyclicity of an association in a model

generic constraint defined on the same concept as Listing 11, and written using EVL. The constraint checks if there are cycles of a given association type, and is defined on the context of the type `A` is bound to.

6.5 Generic transformations and transformation patterns

Concepts may also be used to define transformations in a generic way. In this case we can use concepts either for the source meta-model, for the target, or for both.

As an example, Listing 13 shows part of a generic ETL transformation that transforms a model whose meta-model is bound to the *TokenHolder* concept into a system of equations which can be used for analysing the net using algebraic techniques [40].

The target meta-model *Matrices* allows for the formulation of systems of equation in terms of matrices and vectors, where a matrix is a list of vector columns, and each vector is made of a list of cells storing a number. The *pre* section (lines 1–4) is executed before the transformation starts and creates two variables to store the matrix and vector, to be populated by the transformation. While the matrix is used to store the model topology (connections between holders and processes), the vector stores the number of tokens each holder contains.

```
1  pre {
2    var matrix := new Matrices!Matrix;
3    var vector := new Matrices!Vector;
4  }
5
6  -- transforms a process into a column of the matrix
7  rule process2column
8    transform process : TokenHolder!Process
9    to          vector : Matrices!Vector {
10   for (holder in TokenHolder!Holder.all) {
11     var cell   := new Matrices!Cell;
12     cell.number := process.outHolders.count(holder)
13                    - process.inHolders.count(holder);
14     vector.cells.add(cell);
15   }
16   matrix.rowsize := matrix.rowsize + 1;
17   matrix.colsize := vector.size();
18   matrix.columns.add(vector);
19 }
20
21 -- transforms a holder into a cell of the vector
22 rule holder2cell
23   transform holder : TokenHolder!Holder
24   to          cell   : Matrices!Cell {
25   cell.number   := holder.tokens.size();
26   vector.rowsize := vector.rowsize + 1;
27   vector.cells.add(cell);
28 }
```

Listing 13 Generic ETL transformation

The listing shows two rules of the transformation. The first one (lines 6–19) is used to populate the matrix representing the model topology. The rule is applied to each process, and adds a column to the matrix, representing the connectivity of the process and each holder. Hence, the column has as many cells as holders. Each cell stores the connectivity between the given process and a holder, so that if a process and the holder are not connected, or if the holder is both input and output of the process, the cell stores a `0`. If a holder is just output to a process, it stores a `1`, while if it is only input it stores a -1.[5] The second rule (line 19) is used to generate a vector with the net marking. It is applied to each holder, for which it generates a cell storing the number of tokens of the holder. Then, the cell is added to the output vector.

Once a *TokenHolder* model is transformed into a matrix representation, we can use a mathematical software to, e.g., check if a given state is not reachable,[6] or to calculate all place and transition invariants of the original model by solving two homogeneous matrix equations [40]. Hence, our generic ETL transformation allows applying such analysis techniques on any meta-model for which we can bound the *TokenHolder* concept.

1In this example, the source domain is bound to a generic concept (so that it can be applied to several meta-models), while the target domain uses a specific meta-model (*Matrices*). It is sensible to lift a specific transformation to a generic one only if it can be reused, that is, if the concepts on which it relies can be bound to several concrete meta-models. We have identified the following useful scenarios for generic transformations:

- The transformation implements a complete functionality, so that we do not need to extend the transformation with other rules dealing with specific concerns not present in the transformation. Listing 13 is an example of this scenario.
- The transformation implements a complete functionality, but the binding of the concept into a meta-model does not cover all the types that we need to transform. In this case, we say that the generic transformation is localized [24], and we need to extend it with additional rules to handle the transformation of the unbound types. Sometimes the additional rules only perform a copy of the instances of those types. A mechanism of implicit copy [24] or

[5] Actually, the transformation is more general, as it calculates $w(t, p) - w(p, t)$, where $w(t, p)$ is the weight, or number of times that transition t is connected to place p.

[6] If state $vector'$ is reachable from $vector$, then there exists a valid firing vector w that is a solution to the equation $vector' = vector + matrix \times w$.

conservative copy [42] can then be used to realize localized transformation, automatically copying the elements not covered by the generic transformation.
- The generic transformations implement small, common transformation patterns [2]. The final transformation is built by combining several of such patterns, and possibly with further rules, manually written, which use types from specific meta-models (not generic types). This scenario realizes the idea of reusable design patterns, but applied to model transformations.

7 Related work

The generic programming community [26] has proposed concepts [20,27,48] as a mechanism to make templates more understandable and safe, allowing for static type-checking of templates. In this way, more errors can be given in a meaningful way at the template definition level, as a common problem has been the often difficult to understand errors given when instantiating a template. Concepts can themselves be reused by several generic algorithms, and efforts are currently being spent on developing libraries of meaningful concepts for different programming domains [48].

We have adapted *generic programming concepts* to MDE yielding *meta-model concepts*. In both cases, concepts act as a support for type-checking of operations. However, while programming concepts usually give the requirements for *one* type (one meta-class), meta-model concepts gather the requirements for a collection of related meta-classes. While programming concepts usually require the definition of operations, meta-model concepts demand certain structure in meta-models, and can be annotated with the actions performed by the generic operation.

Our concepts and their binding are related to the notion of model subtyping developed in [45]. That work establishes the requirements for a *subtype of* relation between two meta-models. Such relation opens the possibility of safe substitution of instances of the supertype by instances of the subtype. That is, *every* operation that can be executed at the supertype should be executable at the subtype. Our binding between a concept and a specific meta-model can be seen as establishing a subtype relation between the concept (supertype) and the meta-model (subtype). However, the subtyping relationship is too demanding for our purposes, and therefore we relax it specifying *concept usages*. These are realized as decorations on the different meta-classes of the concept specifying which operation (create, delete) the generic program performs. In this way, not every operation should be type-safe, but only those specified by the decorations, hence widening the applicability of meta-model concepts. Kühne [38] studies different variations of model subtyping and also recognizes that pure subtyping is often too restrictive in practice. He proposes *observers* that define custom relaxation criteria for specialization relationships, allowing weaker forms of subtyping. Interestingly, [38] also proposes *contexts*, which restrict the possible operations to be performed to the supertypes. Our *concept usages* are a way of specifying such *contexts*, which can be practically used when explicitly defining specialization relations between two meta-models.

Our binding is also related to the notion of clan-morphism developed in [15,28]. Clan-morphisms were originally developed in [15], to take into consideration inheritance in meta-models, and to permit the application of graph transformation rules to objects with more specific type. In [28] they were used with the purpose of modifying meta-models by means of rules. Our notion of binding considers more elements (attributes, multiplicities, uniqueness constraints, composition, usage contexts, etc.), and is used for a different purpose, however, it is not formalized, which is left for future work.

Regarding practical implementations, Kermeta [31] includes the aforementioned facilities for model typing [45]. Hence, generic behaviours can be defined in a generic meta-model and applied to any subtype meta-model. This approach has been applied to generic refactorings [39]. We believe that our *concept usages* could be adopted by this approach to relax the requirements of pure subtyping, and making generic operations more applicable. On the other hand, we could also use Kermeta's approach to meta-model pruning [44] to automatically derive concepts given a concrete operation that we want to make generic.

With respect to model management frameworks, some of the tools developed by the MDE community include the *AmmA* toolbox [3] as well as the MOMENT tool [6]. Even though both tools offer comprehensive support for different model management tasks, they could benefit from our meta-model concepts to obtain generic and reusable specifications.

The data-base community faces similar needs with respect to model management [1]. In the context of data-bases, some model-management operations are schema independent (like, e.g., *differencing* or *schema matching* for approximate data integration), while others (like data transformation) have to be specifically designed for concrete schemas. Therefore, the techniques we have presented here are also valuable in that context. There are some works directed to promote reusability of operations, e.g., the GeRoMe [30] model management tool offers a common meta-model—based on role classes—so that models in different technological spaces (like XML or OWL) can be expressed in it. This has the advantage that the same model management operators can be applied to these heterogeneous models, but there is a explicit phase—implemented as a bidirectional model transformation—of import/export between the specific technology and the common meta-model. In our approach, there is no need for such transformation, as this is done with the binding.

In [17], modelling concepts were first proposed, with an application to the definition of generic simulators. As the framework supports an arbitrary number of meta-levels, concepts can be defined not only for meta-models, but for models as well. The approach was combined with model and meta-model templates, enhancing the abstraction and reusability of models and meta-models. In the present work, we continue that line of research by a complete and more refined notion of binding, the specification of concept usages, an implementation on top of EMF and the Epsilon family of languages, and its application to a range of model management operations (and not just simulation).

Finally, some works [10,13,14,21] deal with genericity in meta-modelling. While our approach deals with generic behavior for MDE, these works are directed to genericity in data (i.e., models). Catalysis' model frameworks [21] are parameterized packages that can be instantiated by name substitution. The package templates of [10] are based on those of Catalysis, and are used to define languages in a modular way. They are based on string substitution, as the parameters of the templates are strings that are substituted in the template definition. This approach is realized in the XMF tool [11].

As we discussed in Sect. 2.2, templates are present in the UML 2 specification too. In particular, several elements like classifiers, packages, collaborations and operations can be turned into templates. UML templates declare a signature, which is a list of formal parameters to be substituted by actual parameters in a binding. Hence, UML only permits a simple way of expressing constraints for each parameter in isolation. Thus, even though powerful, we believe that the genericity in UML can be improved by allowing *concepts* (instead of simply a list of formal parameters). This is so as concepts also allow expressing required relations between the parameters, and can be reused by different templates. In addition, the standard UML binding could be refined if concepts were provided with context usages.

Some works have addressed current drawbacks of UML 2 templates. For example, [9] formalizes template binding using OCL. The work in [13,14] is directed to genericity in modelling by extending the UML 2 package templates with contracts. With a similar purpose to meta-model concepts, these contracts specify the allowed substitution of type parameters by concrete types. Note again that the work we present here is directed to genericity of the associated model management operations, but does not consider genericity of meta-models and models.

8 Conclusions and future work

The successful application of MDE techniques in industry necessitates improved support for abstraction, reusability and extendibility of all kinds of model management operations. In this paper we have proposed an approach directed to this goal, inspired by techniques already proven in the generic programming community.

We propose *meta-model concepts* for specifying the structural requirements needed by model management operations. In this way, the operations use the types defined in the concept rather than in a particular meta-model, making the operations more general and abstract. A *binding* process maps a concept to a particular meta-model, so that the operations can be executed on instances of this meta-model. Some concept elements—meta-classes and multi-valued attributes—can be decorated with usages (*creation* or *deletion*), as mandated by the generic specification. Usage contexts permit finer control of the binding and widen the applicability of concepts to cases in which a simple subtyping relationship would be too restrictive. Decoupling operations from concrete meta-model types increases the potential for re-use and their applicability to different unrelated meta-models. Indeed our approach shows the usefulness of an extra level of indirection, used so many times in computer science.

The paper also reported on a practical implementation of these ideas on top of the EMF and the Epsilon family of languages for model management. Different examples of the definition of generic simulators for models, code generators, refactorings, transformations, and generic libraries of operations and constraints showed the versatility of the proposal.

In the future, we plan to investigate ways to add more flexibility to the concepts and binding function. This might be useful as, in some situations, a concept may simply reflect a particular way of structuring the elements of a meta-model, whereas other meta-modelling solutions would be possible as well. We are also considering relationships between concepts (like extension or inheritance), and building libraries of concepts and generic operations. We are also developing a formal theory for genericity over transformations defined with graph transformations. Finally, the inclusion of arbitrary OCL conditions in the concept and the target meta-model and their implications for the binding are also under consideration.

Acknowledgements This work has been sponsored by the Spanish Ministry of Science and Innovation with project METEORIC (TIN2008-02081), and by the R&D program of the Community of Madrid with projects "e-Madrid" (S2009/TIC-1650) and GUIDE (co-financed with the Universidad Autónoma of Madrid, CCG10-UAM/TIC-5772). Parts of this work were done during the research stays of Esther and Juan at the University of York, with financial support from the Spanish Ministry of Science and Innovation (grant refs. JC2009-00015, PR2009-0019 and PR2008-0185). We would like to thank the referees for their useful and detailed comments.

References

1. Bernstein, P.A., Melnik, S.:Model management 2.0: manipulating richer mappings. In: SIGMOD Conference 2007, pp. 1–12. ACM, New York (2007)

2. Bézivin, J., Jouault, F., Palies, J.: Towards model transformation design patterns. In: EWMT'05 (2005)
3. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the large and modeling in the small. In: MDAFA'04. LNCS, vol. 3599, pp. 33–46 (2004)
4. Bonet, P., Llado, C., Puijaner, R., Knottenbelt, W. PIPE v2.5: A petri net tool for performance modelling. In: CLEI'07 (2007). http://pipe2.sourceforge.net/
5. Boost.http://www.boost.org/
6. Boronat, A., Carsí, J.A., Ramos, I.: Automatic support for traceability in a generic model management framework. In: ECMDA-FA'05. LNCS, vol. 3748, pp. 316–330. Springer, Berlin (2005)
7. Bottoni, P., Guerra, E., de Lara, J.: Enforced generative patterns for the specification of the syntax and semantics of visual languages. J. Vis. Lang. Comput. **19**(4), 429–455 (2008)
8. BPMN. http://www.bpmn.org/
9. Caron, O., Carré, B., Muller, A., Vanwormhoudt, G.: An OCL formulation of UML2 template binding. In: UML'04. LNCS, vol. 3273, pp. 27–40. Springer, Berlin (2004)
10. Clark, T., Evans, A., Kent, S.: Aspect-oriented metamodelling. Comput. J. **46**, 566–577 (2003)
11. Clark, T., Sammut, P., Willans, J.: Applied Metamodelling, a Foundation for Language Driven Development, 2nd edn. Ceteva, Chester (2008)
12. CPNTools. http://wiki.daimi.au.dk/cpntools
13. Cuccuru, A., Mraidha, C., Terrier, F., Gérard, S.: Templatable metamodels for semantic variation points. In: ECMDA-FA'07. LNCS, vol. 4530, pp. 68–82. Springer, Berlin (2007)
14. Cuccuru, A., Radermacher, A., Gérard, S., Terrier, F.: Constraining type parameters of UML 2 templates with substitutable classifiers. In: MoDELS'09. LNCS, vol. 5795, pp. 644–649. Springer, Berlin (2009)
15. de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed graph transformation with node type inheritance. Theor. Comput. Sci. **376**(3), 139–163 (2007)
16. de Lara, J., Guerra, E.: Deep meta-modelling with MetaDepth. In: TOOLS'10. LNCS, vol. 6141, pp. 1–20. Springer, Berlin (2010). http://astreo.ii.uam.es/~jlara/metaDepth/
17. de Lara, J., Guerra, E.: Generic meta-modelling with concepts, templates and mixin layers. In: MoDELS'10. Part I, LNCS, vol. 6394, pp. 16–30. Springer, Berlin (2010)
18. de Lara, J., Guerra, E., Bottoni, P. Triple patterns: compact specifications for the generation of operational triple graph grammar rules. In: GT-VMT'07. Electronic Communications of the EASST, vol. 6 (2007)
19. de Lara, J., Vangheluwe, H.: Automating the transformation-based analysis of visual languages. Formal Aspects Comput. **22**, 297–326 (2010)
20. Dos Reis, G., Stroustrup, B.: Specifying C++ concepts. In: POPL'06, pp. 295–308. ACM, New York (2006)
21. D'Souza, D.F., Wills, A.C.: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addison-Wesley Longman Publishing Co. Inc., Reading (1999)
22. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer, Berlin (2006)
23. Epsilon. http://www.eclipse.org/gmt/epsilon/ (2010)
24. Etien, A., Muller, A., Legrand, T., Blanc, X.: Combining independent model transformations. In: SAC'10, pp. 2239–2345. ACM, New York (2010)
25. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
26. García, R., Jarvi, J., Lumsdaine, A., Siek, J.G., Willcock, J.: A comparative study of language support for generic programming. SIGPLAN Not. **38**(11), 115–134 (2003)
27. Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Dos Reis, G., Lumsdaine, A.: Concepts: linguistic support for generic programming in C++. SIGPLAN Not. **41**(10), 291–310 (2006)
28. Hermann, F., Ehrig, H., Ermel, C.: Transformation of type graphs with inheritance for ensuring security in e-government networks. In: FASE'09. LNCS, vol. 5503, pp. 325–339. Springer, Berlin (2009)
29. Hillah, L., Kindler, E., Kordon, F., Petrucci, L., Treves N.: A primer on the Petri net markup language and ISO/IEC 15909-2. Petri Nets Newsl. **76**, 9–28 (2009). http://www.pnml.org
30. Kensche, D., Quix, C., Chatti, M.A., Jarke, M.: Gerome: a generic role based metamodel for model management. J. Data Semant. **8**, 82–117 (2007)
31. Kermeta. http://www.kermeta.org/
32. Kolovos, D.S.: Establishing correspondences between models with the Epsilon Comparison Language. In: ECMDA-FA'09. LNCS, vol. 5562, pp. 146–157. Springer, Berlin (2009)
33. Kolovos, D.S., Paige, R.F., Polack F.: The Epsilon Object Language (EOL). In: ECMDA-FA'06. LNCS, vol. 4066, pp. 128–142. Springer, Berlin (2006)
34. Kolovos, D.S., Paige, R.F., Polack F.: Merging models with the Epsilon Merging Language (EML). In: MoDELS'06, vol. 4199, pp 215–229. Springer, Berlin (2006)
35. Kolovos, D.S., Paige, R.F., Polack F.: On the evolution of OCL for capturing structural constraints in modelling languages. In: Rigorous Methods for Software Construction and Analysis. LNCS, vol. 5115, pp. 204–218 (2009)
36. Kolovos, D.S., Paige, R.F., Polack F.: The Epsilon Transformation Language. In: ICMT'08. LNCS, vol. 5063, pp. 46–60. Springer, Berlin (2008)
37. Kolovos, D.S., Paige, R.F., Polack, F., Rose, L.M.: Update transformations in the small with the Epsilon Wizard Language. J. Object Technol. **6**(9), 53–69 (2007)
38. Kühne, T.: An observer-based notion of model inheritance. In: MoDELS'10. Part I, LNCS, vol. 6394, pp. 31–45. Springer, Berlin (2010)
39. Moha, N. Mahé, V. Barais, O. Jézéquel. J.-M.: Generic model refactorings. In: MoDELS'09. LNCS, vol. 5795, pp. 628–643 (2009)
40. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989)
41. OMG. UML 2.3 specification. http://www.omg.org/spec/UML/2.3/
42. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model migration with Epsilon Flock. In: ICMT'10. LNCS, vol. 6142, pp. 184–198. Springer, Berlin (2010)
43. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack,F.: The Epsilon Generation Language. In: ECMDA-FA'08. LNCS, vol. 5095, pp. 1–16. Springer, Berlin (2008)
44. Sen, S., Moha, Baudry, B. Jézéquel. J.-M.: Meta-model pruning. In: MoDELS. LNCS, vol. 5795, pp. 32–46 (2009)
45. Steel, J., Jézéquel, J.-M.: On model typing. SoSyM **6**(4), 401–413 (2007)
46. Steinberg, D., Budinsky, F., Paternostro, M., Merks E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional, Reading (2008). http://www.eclipse.org/modeling/emf/
47. Stepanov A., Lee M.: The standard template library. Technical Report 95-11(R1), HP Laboratories (1995)
48. Stepanov, A., McJones, P.: Elements of Programming. Addison-Wesley, Reading (2009)
49. Stroustrup,B.: The C++0x remove concepts decision. Dr.Dobbs (2009) http://www.ddj.com/cpp/218600111
50. Sun. Java Metadata Interface. http://java.sun.com/products/jmi/index.jsp

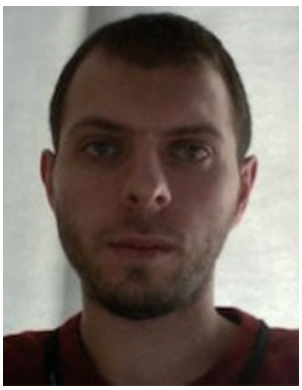
Author Biographies

Louis Rose is a PhD candidate and Research Associate in the Department of Computer Science at the University of York, United Kingdom. His PhD thesis examines software evolution in the context of MDE. His e-mail address is louis@cs.york.ac.uk and his web-page is http://www.cs.york.ac.uk/~louis.

Esther Guerra PhD in Computer Science from Universidad Autónoma (Madrid). From 2010, she is working in the Computer Science Department of the Universidad Autónoma in Madrid, and previously, she worked in the Computer Science Department at Carlos III University in Madrid. She has been a doctoral researcher at the Institute of Theoretical Computer Science (TU Berlin) and at the University of Rome "Sapienza", as well as a post-doctoral researcher at the University of York (UK). Her research interests focus on meta-modelling, formalization of software patterns, and formal techniques in model-driven development primarily for model transformation. Her e-mail address is Esther.Guerra@uam.es and her web-page is http://www.ii.uam.es/~eguerra.

Juan de Lara is an associate professor at the Computer Science Department of the Universidad Autónoma in Madrid, where he teaches Software Engineering, Model-Driven Development, and Automata Theory. He holds a PhD degree in Computer Science, and works in areas such as modelling and simulation, meta-modelling, visual languages and graph transformation. He has been a post-doctoral researcher at the MSDL lab (McGill University), the institute of theoretical computer science (TU Berlin), the department of computer science of the University of Rome "Sapienza" and the University of York (UK). His e-mail address is Juan.deLara@uam.es and his web-page is http://www.ii.uam.es/~jlara.

Anne Etien received a PhD degree in Computer Science from the University of Paris 1 Pantheon-Sorbonne, France in 2006. She is now Associate Professor in the University of Lille 1 in France and makes her research at the LIFL and the INRIA Lille Nord Europe. Her area of interest includes model driven engineering. More specifically, she works on various aspects of model transformations, including chaining, evolution, reusability, traceability, genericity. Her e-mail address is Anne.Etien@lifl.fr and her web page is http://www.lifl.fr/~etien/.



Dimitris Kolovos is a lecturer in Enterprise Systems in the Department of Computer Science of the University of York. To date, he has published more than 50 articles in international journals, conferences and workshops in the field of MDE, and is currently leading the development of the Epsilon open source MDE platform (http://www.eclipse.org/gmt/epsilon). In the past, he has participated in several collaborative projects on MDE including the ModelWare and ModelPlex EU IP projects, and the MADES EU STREP project, and is currently a co-investigator in the COMPASS project which is funded by Eurocontrol under the SESAR-JU initiative. His e-mail address is dkolovos@cs.york.ac.uk and his web-page is http://www.cs.york.ac.uk/~dkolovos.

Richard Paige is professor of Enterprise Systems at the Department of Computer Science, University of York, UK. He leads research on MDE, agile methods, formal methods, and distributed systems. He is principal investigator on a number of large-scale modelling projects, including MADES and INESS and the SESAR-JU COMPASS project. He is the director of the UK Engineering Doctorate Centre in Large-Scale Complex IT Systems, and is on the editorial boards of Elsevier's Journal of Systems Architecture, and Springer's Software and Systems Modelling. He was program chair or co-chair for TOOLS Europe 2008, ICMT 2009, ECMDA 2009, and ICECCS 2010. His e-mail address is paige@cs.york.ac.uk and his web-page is http://www.cs.york.ac.uk/~paige.