On Model Subtyping

Clément Guy, Benoit Combemale, Steven Derrien, James Steel, Jean-Marc Jézéquel

On Model Subtyping

Clément Guy[1], Benoit Combemale[1], Steven Derrien[1],
Jim R. H. Steel[2], and Jean-Marc Jézéquel[1]

[1] University of Rennes1, IRISA/INRIA, France
[2] University of Queensland, Australia

Abstract. Various approaches have recently been proposed to ease the manipulation of models for specific purposes (*e.g.,* automatic model adaptation or reuse of model transformations). Such approaches raise the need for a unified theory that would ease their combination, but would also outline the scope of what can be expected in terms of engineering to put model manipulation into action. In this work, we address this problem from the model substitutability point of view, through model typing. We introduce four mechanisms to achieve model substitutability, each formally defined by a subtyping relation. We then discuss how to declare and check these subtyping relations. This work provides a formal reference specification establishing a family of model-oriented type systems. These type systems enable many facilities that are well known at the programming language level. Such facilities range from abstraction, reuse and safety to impact analyses and auto-completion. **Key words:** SLE, Modeling Languages, Model Typing, Model Substitutability

1 Introduction

The growing use of Model Driven Engineering (MDE) and the increasing number of modeling languages has led software engineers to define more and more operators to manipulate models. These operators are defined in terms of model transformations expressed at the language level, on the corresponding metamodel. However, new modeling languages are still generally designed and tooled from scratch with few possibilities to reuse structure or model manipulations from existing modeling languages.

To address the need for a more systematic *engineering* of model transformations, various approaches have recently been proposed. These approaches include model transformation reuse [1,2,3,4,5,6] and automatic model adaptation [7,8,9,10]. Although these approaches do meet their goals, they remain somewhat disconnected from each other, and lack a unified theory enabling both their combination and comparison. Such a formalization would also help defining the scope of what can be expected (from a engineering point of view) to put model manipulation into action.

In this paper, we tackle the problem from the model substitutability point of view, through model typing. Model typing provides a well-defined theory that considers models as first-class entities, and typed by their respective model types [3]. In addition to the previous work on model typing focusing on the typing relation (*i.e.*, between a model and its model types), we introduce four model subtyping relations. These relations provide model substitutability, that is they enable a model typed by A to be safely used where a model typed by B is expected, A and B being model types.

This work provides a formal reference specification establishing a family of model type systems. These type systems enable many facilities that are well known at the programming language level, ranging from abstraction, reuse and safety to auto-completion.

This paper is structured as follows. We first illustrate the need for a systematic engineering for model manipulation using examples from the optimizing compilation community (Section 2). We then provide some background on MDE and model typing as introduced by Steel *et al.* [3] in Section 3. In Section 4 we formally define four model subtyping relations, based on two criteria: the structure and the considered subset of the involved model types. Section 5 addresses the ways to declare and check these subtyping relations. Section 6 classifies existing approaches providing reuse facilities for model manipulation with respect to the specification of model-oriented type systems provided in sections 4 and 5. Finally, Section 7 concludes the paper and summarizes our ideas for future work.

2 Illustrative Examples

Optimizing compilers have always used abstractions (*i.e.*, models) of the compiled program to apply numerous analyses, which are often tedious to implement. Thus, optimizing compilation seems a good candidate when looking for a domain in which model manipulation engineering facilities would be valuable.

Most analyses performed by optimizing compilers leverage sophisticated algorithms implemented on different types of compilers' intermediate representations (IRs). Implementation of such algorithms is known to be a tedious and error-prone task. As a consequence, providing modularity and reuse is a crucial issue for improving compiler quality but also compiler designer productivity.

Dead Code Elimination (DCE) is an example of such an algorithm. It is a classical compiler optimization which removes unreachable code from the control flow graph of a program (*e.g.*, the `else` branch of a `if` whose condition is always true) [11]. Although the DCE algorithm is relatively straightforward, there exist more complex analyses, such as those leveraging abstract interpretation techniques [12]. Such analyses can infer accurate invariants over the values of the variables of the analyzed program. These invariants can then be used to detect possible program errors (*e.g.*, buffer overflows) or to offer program optimization opportunities.

As with many compiler optimizations, the scope of applicability of these algorithms is wide, including most imperative programming languages. We consider three examples of such languages: two compiler IRs, the GeCoS and the ORCC IRs; and a toy procedural language, Simple. GeCoS[1] is a retargetable C compiler infrastructure targeted at embedded processors. ORCC[2] is a compiler for CAL[3], a dataflow actor language in which actions are described with a standard imperative semantics. Both IRs are metamodel-based: models conforming to these metamodels are used as abstractions of

[1] Cf. `http://gecos.gforge.inria.fr`
[2] Cf. `http://orcc.sourceforge.net/`
[3] Cf. `http://ptolemy.eecs.berkeley.edu/papers/03/Cal/index.htm`

the compiled program. Finally, Simple is a language on which is defined P-Interproc[4], an interprocedural analyzer implementing several abstract interpretation analyses.

Rather than being reimplemented for each targeted languages (*e.g.*, GeCoS IR, ORCC IR and Simple), we would expect DCE and our abstract interpretation analyses to be defined once and then reused across these languages. Of course, this reuse should be done safely and should be transparent for the programmer. More precisely, the only thing the programmer should care about when building his compilation flow is whether his model is eligible (or not) for a given model manipulation.

Facilities such as abstraction (hiding unnecessary details from the programmer), reuse (sharing a model manipulation between different metamodels), and safety (forbidding erroneous reuse) could be provided by a type system specifically targeted at models. Such a model-oriented type system would greatly help in increasing both programmers productivity and model-oriented software quality.

3 Background

In this section, we first present the MOF (Meta-Object Facility) metalanguage, the basis for metamodels, and thus model manipulation operators. We then present model types as introduced by Steel *et al.* [3] on which we base our model subtyping relations. Finally, we discuss the limits of the model subtyping relation proposed by Steel *et al.*

3.1 Model Driven Engineering

The Meta-Object Facility (MOF) [13] is the OMG's standardized meta-language, *i.e.*, a language to define metamodels. As such it is a common basis for a vast majority of model-oriented languages and tools. A metamodel defines a set of models on which it is possible to apply common operators. Therefore, model substitutability must take into account MOF and the way it expresses metamodels.

Figure 1 displays the structure of EMOF (Essential MOF) which contains the core of the MOF meta-language in the form of a class diagram. EMOF supports the definition of the concepts and relationships of a metamodel using `Class`es and `Property`s. `Class`es can be abstract (*i.e.*, they cannot be instantiated) and have `Property`s and `Operation`s, which declare respectively attributes and references, and the signatures of methods available from the modeled concept. `Class`es can have several superclasses, from which they inherit all `Property`s and `Operation`s. A `Property` can be composite (an object can only be referenced through one composite `Property` at a given instant), derived (*i.e.*, calculated from other `Property`s) and read-only (*i.e.*, cannot be modified). A `Property` can also have an opposite `Property` with which it forms a bidirectional association. An `Operation` declares the `Type`s of the exceptions it can raise and ordered `Parameter`s. `Property`s, `Operation`s, and `Parameter`s are `TypedElement`s; their type can be either: a `Datatype` (*e.g.*, `Boolean`, `String`, etc.) or a `Class`. `Parameter`s, `Property`s and `Operation`s are `Multiplicity Element`s. As such, they have a multiplicity (defined by a lower and an upper bound), as well as orderedness and uniqueness.

[4] Cf. `http://pop-art.inrialpes.fr/interproc/pinterprocweb.cgi`

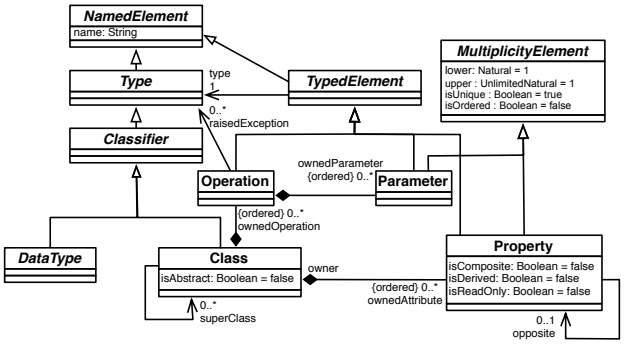


Fig. 1. The EMOF core with class diagram notation

Metamodels can be seen as class diagrams, each of their concepts being instantiable by objects belonging to models. However, metamodel concepts are also instances of MOF elements and thus a metamodel can be drawn as an object diagram where each concept is an instance of one of the MOF elements (*e.g.*, classes `Class` or `Property`).

Because model subtyping takes place at the metamodel level, the latter representation facilitates the definition of model subtyping relations by depicting metamodels and their contained concepts as objects with attributes and properties. Thus, we will use the object diagram representation in preference to the more common class diagram one.

3.2 Model Typing

Model Types were introduced by Steel *et al.* [3], as an extension of object typing to provide abstraction from object types and enable model manipulation reuse.

Definition 1. (Model type) *A type of a model is a set of types of objects which may belong to the model, and their relations.*

MOF classes are closer to types (interfaces) than to object classes, thus a model type is closely related to a metamodel. The difference between model types and metamodels lies in their respective relations with models. A model has one and only one metamodel to which it conforms. This metamodel contains all the types needed to instantiate objects of the model. Conversely, a model can have several model types which are subsets of the model's metamodel.

Because model types and metamodels share the same structure, it is possible to extract the type of a model from its metamodel (we call the model type containing all the types from a model's metamodel the *exact type* of the model). Figure 2 represents a model m_1 which conforms to a metamodel MM_1 and is typed by model types MT_A and MT_B, MT_B being the *exact type* of m_1 extracted from MM_1. Both metamodels and model types conforms themselves to MOF.

MOF delegates the definitions of contracts (*e.g.*, pre and post-conditions or invariants) to other languages (*e.g.*, OCL, the Object Constraint Language [14]). Hence nei-

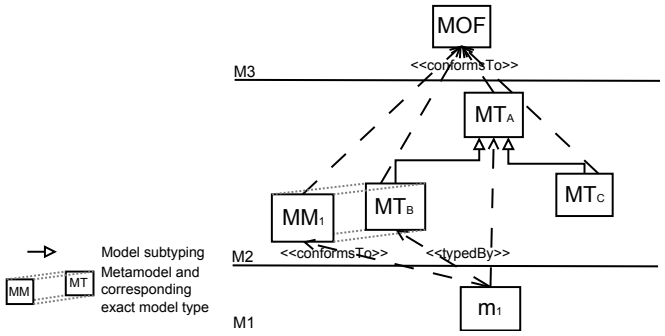


Fig. 2. Conformance, model typing and model subtyping relations

ther the original paper on model typing [3] nor this one considers contracts in subtyping relations, but focuses on features of object types contained by model types.

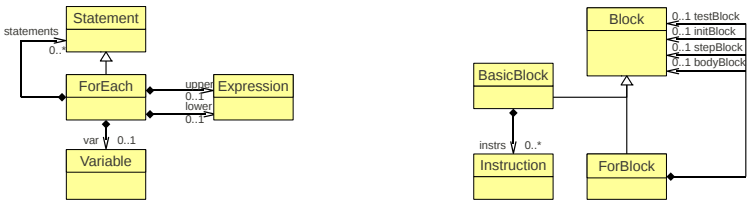
Substitutability is the ability to safely use an object of type *A* where an object of type *B* is expected. Substitutability is supported through subtyping in object-oriented languages. However, object subtyping does not handle type group specialization (*i.e.*, the possibility to specialize relations between several objects and thus groups of types).[5] Such type group specialization have been explored by Kühne in the context of MDE [16]. Kühne defines three model specialization relations (specification import, conceptual containment and subtyping) implying different level of compatibility. We are only interested here in the third one, subtyping, which requires an *uncompromised mutator forward-compatibility*, *e.g.*, substitutability, between instances of model types.

Model Type Matching is a model subtyping relation proposed by Steel *et al.* to enable safe model manipulation reuse in spite of limits of object subtyping. To this end, they use the *object type matching* relation defined by Bruce *et al.* [17], which is more flexible than subtyping. For more details, we refer the reader to Steel's PhD thesis [18].

Definition 2. (Model type matching proposed by Steel *et al.* [3]) *Model Type MT_B matches model type MT_A if for each object type C in MT_A there is a corresponding object type with the same name in MT_B such that every property and operation in $MT_A.C$ also occurs in $MT_B.C$ with exactly the same signature as in $MT_A.C$.*

Limits of Model Type Matching However, *model type matching* as presented by Steel *et al.* is subject to some shortcomings. First, the type rules they present, and their implementation in Kermeta[6], violate their definition of *type matching* by permitting the relaxation of lower multiplicities, i.e. by allowing a non-mandatory attribute to be matched by a mandatory one, which could potentially lead to an invalid model.

[5] We refer the reader interested in the type group specialization problem to the Ernst's paper [15].
[6] Cf. `http://www.kermeta.org`



(a) `foreach` loop in ORCC IR (b) `for` loop in GeCoS IR

Fig. 3. Extracts of ORCC IR and GeCoS IR metamodels

In addition, and more significantly, finding a model type common to several model types (*e.g.*, GeCoS IR, ORCC IR and Simple) is not always possible, even if they share numerous concepts (*e.g.*, concepts used in DCE). This impossibility is due to structural heterogeneities between the metamodels [19]. Figure 3 presents such heterogeneities between excerpts from the GeCoS IR and the ORCC IR metamodels representing `foreach` (from ORCC IR) and `for` loops (from GeCoS IR, and thus C). The former (Figure 3(a)) is a simpler loop than the latter (Figure 3(b)), iterating only by steps of one on a given variable between bounds, where a C `for` can have complete code blocks as initialization, step and test.

Thus to reuse a model manipulation (*e.g.*, DCE), a subtyping mechanism should provide for the definition of an adaptation, needed to *bind* different structures to a single one on which the manipulation is defined. In our example, such an adaptation could be the transformation of `foreach` loops into more generic `for` loops, using the variable and the lower bound to produce an initialization block, the variable and the upper bound to produce a test block, and automatically producing a step block with a step of one.

This adaptation should be able to adapt back the result of the manipulation, because this manipulation could modify the model it processes or return a result containing elements of the model. For example, DCE modifies the representation of the program by removing code. Once the optimization has been processed on a common representation, it should be possible to adapt back the structure to impact the result of DCE in the original structure (*i.e.*, an ORCC IR or GeCoS IR model).

Although defining common optimizations on a minimal dedicated structure seems to best fit the need for modularity and reuse, we need to consider the presence of legacy code. For example, DCE is already implemented for the GeCoS IR. Reusing this implementation on ORCC IR would avoid the creation of a generic model type and the reimplementation of the optimization. However, the GeCoS IR does not contain only the concepts required for DCE. More particularly, it contains concepts which do not exist in ORCC IR (*e.g.*, pointers). Therefore, a model subtyping mechanism should be able to accept a subtype which only possesses the concepts of the supertype required for the reuse of a specific model manipulation.

4 Model Subtyping Relations

Object-oriented type systems provide important systematic engineering facilities, including abstraction, reuse and safety. We strongly believe that these facilities can also be provided for model manipulation through a model-oriented type system. However, the existing model subtyping relation has shown some limitations.

For this reason, in this section we review four subtyping relations between model types, based on two criteria: the presence of heterogeneities between the two model types (Subsections 4.1 and 4.2) and the considered subset of the model types (Subsections 4.3 and 4.4). Such a model subtyping relation is pictured in Figure 2 by the generalization arrow between model types MT_A and MT_B. Through this subtyping relation, models typed by MT_A are substitutable to models typed by MT_B, *i.e.*, model manipulations defined on MT_B can be *safely reused* on model typed by MT_A.

4.1 Isomorphic Model Subtyping

An obvious way to safely reuse on a model typed by MT_B a model manipulation from a model type MT_A is to ensure that MT_B contains substitutable concepts (*e.g.*, classes, properties, operations) for those contained by MT_A. As mentioned in Section 3, it is not possible to achieve type group (or model type) substitutability through object subtyping.

MOF Class Matching Thus, we use an extended definition of *object type matching* introduced by Bruce *et al.* [17] and used by Steel *et al.* to define their *model type matching* relation. Our *object type matching* relation is similar to, but stricter than the latter, because class names must be the same, as must lower and upper bounds of multiplicity elements. Moreover, every mandatory property in the matching type requires a corresponding property in the matched type, in order to prevent model manipulation from instantiating a type without its mandatory properties.

Definition 3. (MOF class matching) *MOF class T' matches T (written $T' <\# T$) iff:*

1 $T.name = T'.name$
2 $T'.isAbstract \Rightarrow T.isAbstract$
3 $\forall op \in T.ownedOperation, \exists S' \in SuperClasses(T')$ *such that* $\exists op' \in S'.ownedOperation$ *and:*
  3.1 $op.name = op'.name$
  3.2 $op'.type <\# op.type \vee op.type <: op'.type$
  3.3 $\forall p \in op.ownedParameter, \exists p' \in op'.ownedParameter$ *such that:*
    (a) $\exists U' \in SubClasses(p'.type)$ *such that* $U' <\# p.type \vee p.type <: p'.type$
    (b) $p.rank = p'.rank$
    (c) $p.lower = p'.lower$
    (d) $p.upper = p'.upper$
    (e) $p.isUnique = p'.isUnique$
  3.4 $\forall e' \in op'.raisedException, \exists e \in op.raisedException$ *such that* $e' <\# e \vee e' <: e$
4 $\forall a \in T.ownedAttribute, \exists S' \in SuperClasses(T')$ *such that* $\exists a' \in S'.ownedAttribute$ *such that:*

4.1 $a.name = a'.name$
4.2 $a'.isReadOnly \Rightarrow a.isReadOnly$
4.3 $a.isComposite = a'.isComposite$
4.4 $a'.type <\# a.type \vee (a'.type <: a.type \wedge a.isReadOnly)$
4.5 $a.lower = a'.lower$
4.6 $a.upper = a'.upper$
4.7 $a.opposite \neq void \Rightarrow a'.opposite \neq void \wedge a.opposite.name = a'.opposite.name$
4.8 $a.isUnique = a'.isUnique$

5 $\forall a' \in T'.ownedAttribute,\ a'.lower > 0 \Rightarrow \exists S \in SuperClasses(T)$ *such that* $\exists a \in S.ownedAttribute \wedge a.name = a'.name$

Where $SuperClasses(T)$, is the set of all superclasses of T, $SubClasses(T)$, the set of all its subclasses, both including T and $<:$ is the object subtyping relation.

Model Type Matching Given the conditions under which objects may be substitutable in the context of a model type, we can define a *model type matching* relation which ensures the safe type group substitutability. Based on the definition of MOF class matching, we redefine the *model type matching* relation as follows:

Definition 4. (Model type matching) *The* model type matching *relation is a binary relation* $\sqsubseteq$ *on ModelType, the set of all model types, such that* $(MT_B, MT_A) \in \sqsubseteq$ *(also written* $MT_B \sqsubseteq MT_A$*) iff* $\forall T_A \in MT_A, \exists T_B \in MT_B$ *such that* $T_B <\# T_A$.

The *model type matching* relation can be seen as a kind of subgraph isomorphism which takes into account the MOF specificities (*e.g.*, inherited properties and operations). For this reason we call *isomorphic* model subtyping relation a relation which satisfies the *matching* relation.

4.2 Non-isomorphic Model Subtyping

The fact that MT_B does not match MT_A does not mean that it is not appropriate for substitution. Indeed, the condition for safely substituting a model m for another is that m contains all the necessary information expected to be handled safely by the called model manipulation or to access the desired features. But this information can be under another form than expected (*e.g.*, with different class names) in which case m may be substitutable if the expected form of the information is retrieved.

Model Adaptation is the process of retrieving the information from a model in the form expected. It consists in adapting a model m_B into a model m_A which can be handled by the operation or through which it is possible to access to the desired feature. Thus a *model adaptation* is a way to *create* a *model type matching* relation between two model types. A model adaptation is a function defined at the model type level and applied on models. It takes a model m_B typed by MT_B and returns a model m_A with the same information, but in the form defined by MT_A, *i.e.*, a model whose type matches MT_A.

Definition 5. (Model adaptation) *A model adaptation is a function* $adapt_{MT_A}$ *from* MT_B *to* MT_C*, where* MT_A*,* MT_B *and* MT_C *are model types and such that* $MT_C \sqsubseteq MT_A$.

One way to achieve such an adaptation is to implement a model transformation from MT_B to MT_A, in which case $MT_C = MT_A$. Another way is by adding missing types and derived properties from MT_A to MT_B, creating a new model type MT_C with $MT_C \sqsubseteq MT_B$ and $MT_C \sqsubseteq MT_A$. This is the approach followed by Sen *et al.* [5].

Bidirectional Model Adaptation , that is coupled forward adaptation from MT_B to MT_A and backward adaptation from MT_A to MT_B may be needed, depending whether the adaptation is done to reuse an *endogenous* or an *exogenous* model manipulation [20].

If the adaptation to MT_A is done in order to reuse an *endogenous* manipulation, a backward adaptation is necessary in order to reflect changes made to the adapted model on the original model. Conversely, a backward adaptation is not necessary if the reused feature is an *exogenous* manipulation.

The forward and backward adaptation together form a bidirectional adaptation, which enables the adaptation of a model typed by MT_B into a form which fits the expected model type MT_A but also to reflect the result in the original model. Moreover, a roundtrip adaptation, *i.e.*, applying the forward adaptation then the backward adaptation to the result should lead to an unchanged model. To this end, we use here rules defined by Foster *et al.* for well-behaved lenses (*i.e.*, bidirectional transformation operators) [21].

Definition 6. (Bidirectional model adaptation) *A bidirectional model adaptation $adapt_{MT_A}$ between model types MT_B and MT_A comprises a function $adapt_{MT_A} \nearrow$ from MT_B to MT_C and a function $adapt_{MT_A} \searrow$ from $MT_B \times MT_C$ to MT_B, where MT_C is a model type such that $MT_C \sqsubseteq MT_A$ and:*

- $adapt_{MT_A} \nearrow (adapt_{MT_A} \searrow (m_B, m_C)) = m_C, \forall (m_B, m_C) \in MT_B \times MT_C$
- $adapt_{MT_A} \searrow (m_B, adapt_{MT_A} \nearrow (m_B)) = m_B, \forall m_B \in MT_B$

Bidirectional adaptation can be provided through bidirectional transformations. Bidirectional transformations are studied in different disciplines of computer science (*e.g.*, MDE, graph transformations and databases) to synchronize two data structures (a *source* and a *view*) [22,23]. In our case, the *source* is the model typed by MT_B found in a context where a model typed by MT_A (our *view*) is expected.

4.3 Total Model Subtyping

When a model of type MT_B can be used in every context in which a model of type MT_A is expected, we talk about *total* substitutability. Therefore, a subtyping relation which guarantees *total* substitutability is a *total* subtyping relation.

Definition 7. (Total subtype) *MT_B is a total subtype of MT_A if any model typed by MT_B can be safely used everywhere a model typed by MT_A is expected.*

4.4 Partial Model Subtyping

Conversely, a *partial* subtyping relation enable a model typed by MT_B to be used in a given context (*e.g.*, a given model transformation) where a model typed by MT_A is expected. This notion of *usage context* have been introduced by Kühne in order to define in which cases a specialization relation holds, while it does not hold in the general case [16]. Typically, a *partial* subtyping relation enables a model typed by MT_B to be substituted for a model a of type MT_A in the context of the call `m(a)` if MT_B contains the required features for m, even if MT_B is not a *total* subtype of MT_A.

Definition 8. (Partial subtype) *MT_B is a partial subtype to MT_A wrt. f if models typed by MT_B can be safely used where a model typed by MT_A is expected to use the feature f.*

Here, f can be an attribute or an operation from the model or a model manipulation that takes the model as argument. MT_B is a partial subtype to MT_A wrt. f if MT_B is a total subtype of MT_f, where MT_f is a model type which contains only the necessary information to apply or access f safely and such that MT_A is a total subtype of MT_f. We call MT_f the *effective model type* of f.

Definition 9. (Effective model type) *The effective model type MT_f of a feature f extracted from a model type MT_A is the model type which contains all the required features to access or call f and such that $MT_A \sqsubseteq MT_f$.*

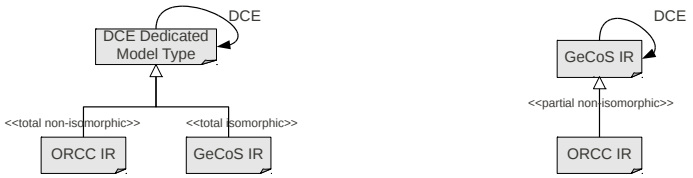
This effective model type can be processed using a function which analyzes the model type and extracts its required subset to access a given feature.

Definition 10. (Effective model type extraction) *The effective model type extraction function is a function $extractEffectiveMT(MT_A, f)$, with MT_A a model type and f a required feature belonging to MT_A, and such that $MT_f = extractEffectiveMT(MT_A, f)$ is the effective model type of f extracted from MT_A.*

One possible way to extract this required subset is to use an approach like the one proposed by Sen *et al.* [5]. They compute a metamodel (called the *effective metamodel*) from a larger metamodel using the *footprint* of a model manipulation, *i.e.*, the set of types and features touched by the manipulation. This footprint can be processed statically, by analyzing the code of the model manipulation or dynamically using a trace of the execution of the operation [24]. The dynamic footprint is more accurate because it contains only the types and features of the objects which have been touched by the operation, whereas the static footprint contains all the types and features which may be touched by the operation. However, the dynamic footprint is also costlier and cannot be used for static type checking (cf. Section 5.2).

4.5 Definition of Subtyping Relations for Model Types

From these two criteria (isomorphism of the structures and totality of the subtyping), we define four model subtyping relations to provide model substitutability. In the following MT_A and MT_B are model types and $ModelType$ is the set of all model types.



(a) Total subtyping (b) Partial non-isomorphic subtyping

Fig. 4. Two different scenarios of the reuse of DCE between ORCC IR and GeCoS IR

The first model subtyping relation is the *total isomorphic* subtyping relation, to which the three others refer. MT_B is a *total isomorphic* subtype of MT_A if it contains one matching object type for every object type of MT_A, *i.e.*, if $MT_B \sqsubseteq MT_A$. For example, such a subtyping relation could hold between GeCoS IR and a model type extracted from GeCoS IR by selecting only the relevant concepts for Dead Code Elimination (DCE). In Figure 4(a), where the DCE arrow represent the DCE model manipulation defined on a dedicated model type, this case is represented by the generalization arrow between GeCoS IR and the DCE dedicated model type.

Definition 11. (Total isomorphic subtyping relation) *The* total isomorphic *subtyping relation is the matching relation, denoted* $MT_B \sqsubseteq MT_A$.

A *partial isomorphic* subtyping relation wrt. feature f holds between MT_B and MT_A if MT_B contains matching object types for every object type belonging to the *effective model type* of f extracted from MT_A, *i.e.*, MT_B is *partial isomorphic* subtype of MT_A wrt. feature f if MT_B is a *total isomorphic* subtype of the *effective model* type of f extracted from MT_A.

Definition 12. (Partial isomorphic subtyping relation) *The* partial isomorphic *subtyping relation wrt. the feature* f *is a binary relation* $\sqsubseteq_f$ *on* $ModelType$ *such that* $(MT_B, MT_A) \in <:_f$ *(also written* $MT_B \sqsubseteq_f MT_A$*) iff* $MT' \sqsubseteq extractEffectiveMT(MT_A, f)$.

MT_B is a *total non-isomorphic* subtype of MT_A if there is a adaptation able to adapt every model typed by MT_B in a model typed by a *total isomorphic* subtype of MT_A. This adaptation must be bidirectional, or it would be impossible to reuse *endogenous* model manipulations from MT_A and the subtyping relation would not be *total*. Figure 4(a) represents such a subtyping relation between ORCC IR and the model type dedicated to DCE mentioned above. Loops from the latter are isomorphic to GeCoS IR ones, thus they cannot be isomorphic to loops from the former. Therefore an adaptation is needed, as the one described earlier (see 3.2).

Definition 13. (Total non-isomorphic subtyping relation) *The* total non-isomorphic *subtyping relation is a binary relation* $\underset{\sim}{\sqsubseteq}$ *on* $ModelType$ *such that* $(MT_B, MT_A) \in \underset{\sim}{\sqsubseteq}$ *(also written* $MT_B \underset{\sim}{\sqsubseteq} MT_A$*) iff* $\exists adapt_{MT_A}$ *a bidirectional adaptation from* MT_B *to* MT_C *such that* $MT_C \sqsubseteq MT_A$.

Finally, model type MT_B is a *partial non-isomorphic* subtype of MT_A wrt. the feature f if there is an adaptation able to adapt a model typed by MT_B in a model typed by a

total isomorphic subtype of the effective model type of f extracted from MT_A. This adaptation must be bidirectional if f is an *endogenous* feature. Such a *partial non-isomorphic* subtyping relation is pictured in Figure 4(b), where ORCC IR is subtype of GeCoS IR through an adaptation to the *effective model type* of DCE extracted from GeCoS IR.

Definition 14. (Partial non-isomorphic subtyping relation) *The* partial non-isomorphic *subtyping relation wrt. the feature f is a binary relation $\sqsubseteq_{\sim f}$ on $ModelType$ such that $(MT_B, MT_A) \in \sqsubseteq_{\sim f}$ (also written $MT_B \sqsubseteq_{\sim f} MT_A$) iff $\exists adapt_{MT_A}$ an adaptation from MT_A to MT_C such that $MT_C \sqsubseteq extractEffectiveMT(MT_A, f)$ and $adapt_{MT_A}$ is a bidirectional adaptation if f is an* endogenous *model manipulation.*

5 Putting Subtyping Relations to Work

Defining model subtyping relations is not sufficient to build a type system. Indeed, a type system implements one or more subtyping relations and provides ways to declare and check them. Thus, we discuss here the ways to declare and check subtyping relations and the respective drawbacks and advantages of these approaches for an implementation of a model-oriented type system.

5.1 Declaration of Subtyping Relations

Subtyping relations can be declared in two ways: *explicitly* and *implicitly*. We call a subtyping relation declaration *explicit* when a syntactic construct is used to state the subtyping relation. Conversely, if the type system infers the subtyping relation from the information it can gather about the types or the use which is done from their instances, the declaration of the subtyping relation is *implicit*. In addition, the declaration of the subtyping relation can take place either *at the definition* of a type or *after the definition* of the subtype and the supertype involved in the subtyping relation.

The way to declare model subtyping relations may affect the possibilities that these relations offer through the type system. For example, a *non-isomorphic* model subtyping relation can be declared *implicitly*. To this end, a tool able to infer adaptations is necessary. Such inference can be done through patterns which are known to be safe or using ontologies to find corresponding class or feature names. However, an *implicit* adaptation mechanism will be more limited in terms of possible adaptations than an *explicit* one, which let the user define its adaptation based on its knowledge of the two model types involved. On the other hand, an *explicit* adaptation mechanism needs appropriate syntactic constructs and analyses to ensure that an adaptation is safe.

Declaration of a subtyping relation *at the definition* of a type is a kind of documentation, letting know what are the subtypes or supertypes of the defined type. Conversely, it is not always possible or desirable to add this information in a type, particularly if the subtyping relation is required for a very specific use (*e.g.*, a *partial* subtyping relation for a single model manipulation) or legacy code where existing model types should be modified. In such cases, declaring the subtyping relation *after the definition* of the involved types may be a solution.

Finally, declaration of a subtyping relation *explicitly at the definition* of a model type could allow inheritance. That is, reuse of the structure of the supertype, with the possibility to redefine or modify it in the subtype without breaking model subtyping. Moreover, if *explicit* declaration *at the definition* is the only way to declare model subtyping relations, it prevents from the type system to use subtyping relations which are unknown from the user, and thus prevents from accidental substitutability.

5.2 Checking of Subtyping Relations

Checking of the subtyping relations is the verification that a subtyping relation holds. Regardless of the way the subtyping relation is declared, this check can be processed either at design time, *i.e.*, during the compilation or interpretation process, or at runtime.

Here again, the way to check model subtyping relations can impact the facilities provided for model manipulations. On the one hand, design time (or static) check enables earlier detection (*i.e.*, than runtime check) of type errors and programming mistakes and thus earlier user feedback. It also enables tools to provide more facilities, such as type-based compiler optimizations, auto-completion or impact analyses. Moreover, compared to runtime checking, design time checking needs significantly fewer tests to achieve the same level of runtime safety.

On the other hand, runtime checking can be processed with more precise type information. When the program is running, the actual type of a variable is known rather than its declared type. Although possibly slower because of the process of the check during the execution of the program, dynamic checking enables valid programs which would be forbidden by a static type checker because of a lack of information. In the context of model types, knowing the actual model would enable the extraction of its model type and would possibly enable subtyping relations forbidden by a static type checker.

6 Discussions

Several approaches have been proposed in the last decade to provide engineering facilities for model manipulation reuse. We show in this section how the fthe different model subtyping mechanisms (*i.e.*, total/partial and isomorphic/non-isomorphic model subtyping, declaration and checking) defined in this paper can be used to classify these approaches through a unified theory. Figure 5 summarizes this classification. The question marks indicate the lack of information about the given mechanism.

6.1 Isomorphic vs. Non-isomorphic Subtyping Relations

To the best of our knowledge, the only approach using an isomorphic subtyping relation is the bidirectional subset of the *adaptation* DSL proposed by Babau *et al.* [9,10]. All other approaches either let class names vary or go further, enabling adaptations such as $n-to-1$ concepts *binding* or navigation and filtering of features. The latter use different mechanisms to *bind* the subtype to its supertype and express the adaptation (*e.g., adaptation* and *binding* DSLs or static introduction). The rarity of *isomorphic* subtyping relations can be explained by the restrictions such relations impose, restrictions which can be safely relaxed in some cases, *e.g.*, class names modification.

Fig. 5. Classification of different model manipulation reuse approaches

	Total / partial	Iso / non-iso	At / after definition	Explicit / implicit	Checking	Legacy tool reuse
Varró *et al.* [1]	Total	Non-iso (Class renaming)	After	Implicit	?	No
Cuccurru *et al.* [2]	Total	Non-iso (Abstract class renaming)	After	Explicit (Genericity and explicit object subtyping)	?	Yes
Steel *et al.* [3]	Total	Non-iso (Class renaming, multiplicities contraction)	After	Implicit	At compile-time, with possible runtime type errors	Yes
Sanchez Cuadrado *et al.* [4]	Total	Non-iso (Class renaming, multiplicities contraction)	After	Explicit (Binding DSL)	?	Yes
Sen *et al.* [5]	Partial (Effective metamodel)	Non-iso (Potentially any adaptation)	After	Explicit (Static introduction)	At compile-time, with possible runtime type errors	Yes
De Lara *et al.* [6,7,8]	Total	Non-iso (Class renaming, navigation and filtering of properties, $n-to-1$ bindings)	After (Binding), At definition (Specialization)	Explicit (Binding DSL)	?	No
Babau *et al.* [9,10]	Total (Bidirectional subset)	Iso (Bidirectional subset)	After	Explicit (Adaptation DSL)	?	Yes

6.2 Total vs. Partial Subtyping Relations

Excepting one approach which allows the extraction of the *effective metamodel* from a model manipulation [5], all existing approaches are *total*. To be *total* a *non-isomorphic* subtyping relation must handle bidirectional adaptation. Bidirectionality is tackled in existing approaches by almost *isomorphic* relations [1,2,3,4,9] or by generating an adapted model manipulation rather than adapting the model [6,7,8].

6.3 Declaration of Subtyping Relations

All the existing approaches declare the subtyping relation or *binding after the definition* of the two model types (or their equivalent). However, de Lara *et al.* authorize specialization of model types (called *concepts* in their terminology) using a mechanism close to inheritance (*i.e.*, *at definition*) [6]. Only two approaches declare subtyping relations *implicitly* [1,3] whereas the others use *explicit* mechanisms mainly through DSLs [4,6,7,8,10,9], with the exception of the approaches from Cuccuru *et al.* [2] and Sen *et al.* [5] which use respectively genericity and static introduction.

6.4 Checking of Subtyping Relations

Little is said about the checking of the subtyping relations, apart from the work of Steel *et al.* [3], in which subtyping relations are checked *at compile time*. De Lara *et al.* [6] mention a notion of *valid* binding, but do not formalize it.

6.5 Legacy Tools Reuse

One group of our examples, abstract interpretation analyses, are implemented in an existing tool (P-Interproc). Among the existing approaches, some need to specifically

define the model manipulation to be reused [1], or to process it in order to generate an adapted model manipulation [6,7,8]. By doing so, they prevent from reusing existing model manipulations which have not been defined using their own mechanisms or which sources are not available. The other approaches, which enable a subtyping relation with a legacy tool, are the ones with the fewest possible adaptations [2,3,4,10,9], or without any guarantee on the bidirectionality of such adaptations [5].

7 Conclusion and Perspective

This paper provides a review of the overall scope of model substitutability through model typing. To this end we analyze the subtyping relation between two model types wrt. both their structure, and the context of the need for such a substitutability.

First, to be substitutable a model must be structurally equivalent to the expected one.Such a structural equivalence can be achieved if the structures of the two model types are *isomorphic*, or thanks to an adaptation making the two structures *isomorphic*.

Second, such an isomorphism can be *total*, *i.e.*, achieved between the whole structures of the two model types, allowing a substitution of the corresponding models in every possible context where the substitution is necessary. Otherwise, *partial* model substitutability can be achieved according to a given context. In other words, the isomorphism can be achieved between a model type and the *effective model type* of the feature to be reused, *i.e.*, the subset of the model type used in a given context.

From these distinctions, we define four model subtyping relations providing different kinds of model substitutability: *total isomorphic*, *partial isomorphic*, *total non-isomorphic* and *partial non-isomorphic*. We review existing approaches to model manipulation reuse wrt. these subtyping relations. It appears that few existing approaches use *partial* subtyping relations or enable adaptations to handle complex structural heterogeneities between model types. Moreover some approaches forbid the reuse of legacy tools. More importantly, most of the approaches lack of a way to forbid erroneous reuse.

In addition to the comparison of existing approaches, the subtyping relations introduced in this paper provide a specification of a family of model type systems that can be implanted in a MDE CASE tool to enable *safe reuse* of model manipulations. In this context, we thus discuss ways to declare a subtyping relation, and how to check it. Depending on the chosen subtyping relation, as well as the way to declare and check it, these type systems enable many facilities that are well known at the programming language level, such as type-based compiler optimizations and auto-completion.

As a direct perspective of this work, we plan to refactor the existing model typing in Kermeta to support the subtyping relations identified in this paper. To this end, we plan to integrate the state-of-the-art of the existing approaches and to study how the work of Vignaga *et al.* [25], focusing on the typing of the relations between models as functions, can be combined with our subtyping relations.

Acknowledgement This work has been partially supported by VaryMDE, a collaboration between Inria and Thales Research and Technology, and by the French ANR BioWIC (ANR-08-SEGI-005). The authors thank the anonymous reviewers for their constructive feedback which helped us to considerably improve the article.

References

1. Varró, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In: UML. (2004)
2. Cuccuru, A., Mraidha, C., Terrier, F., Gérard, S.: Templatable metamodels for semantic variation points. In: ECMDA-FA. (2007)
3. Steel, J., Jézéquel, J.M.: On model typing. SoSyM **6**(4) (2007)
4. Sánchez Cuadrado, J., García Molina, J.: Approaches for model transformation reuse: Factorization and composition. In: ICMT. (2008)
5. Sen, S., Moha, N., Mahé, V., Barais, O., Baudry, B., Jézéquel, J.M.: Reusable model transformations. SoSyM **11**(1) (2010)
6. de Lara, J., Guerra, E.: From types to type requirements: genericity for model-driven engineering. SoSyM (2011)
7. Sánchez Cuadrado, J., Guerra, E., de Lara, J.: Generic model transformations: Write once, reuse everywhere. In: ICMT. (2011)
8. Wimmer, M., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Cuadrado, J., Guerra, E., de Lara, J.: Reusing model transformations across heterogeneous metamodels. In: International Workshop on Multi-Paradigm Modeling. (2011)
9. Kerboeuf, M., Babau, J.P.: A DSML for reversible transformations. In: OOPSLA Workshop on Domain-Specific Modeling. (2011)
10. Babau, J.P., Kerboeuf, M.: Domain Specific Language Modeling Facilities. In: MoDELS Workshop on Models and Evolution. (2011)
11. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley (2006)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. POPL (1977)
13. OMG: Meta Object Facility (MOF) 2.0 Core Specification. (2006)
14. OMG: UML Object Constraint Language (OCL) 2.0 Specification. (2003)
15. Ernst, E.: Family Polymorphism. In: ECOOP. (2001)
16. Kühne, T.: On model compatibility with referees and contexts. SoSyM (2012)
17. Bruce, K.B., Schuett, A., van Gent, R., Fiech, A.: Polytoil: A type-safe polymorphic object-oriented language. ACM TOPLAS **25**(2) (2003)
18. Steel, J.: Typage de modèles. PhD thesis, Université de Rennes 1 (April 2007)
19. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schoenboeck, J., Schwinger, W.: From the heterogeneity jungle to systematic benchmarking. In: MoDELS Workshops. (2010)
20. Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science **152** (2006)
21. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM TOPLAS **29**(3) (2007)
22. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional Transformations: A Cross-Discipline Perspective. In: ICMT. (2009)
23. Hu, Z., Schürr, A., Stevens, P., Terwilliger, J.F.: Bidirectional transformation "bx" (dagstuhl seminar 11031). Dagstuhl Reports **1**(1) (2011)
24. Jeanneret, C., Glinz, M., Baudry, B.: Estimating footprints of model operations. In: ICSE. (2011)
25. Vignaga, A., Jouault, F., Bastarrica, M., Brunelière, H.: Typing artifacts in megamodeling. SoSyM (2011)