Using Model Types to Support Contract-Aware Model Substitutability

Wuliang Sun[1], Benoit Combemale[2], Steven Derrien[2], Robert B. France[1]

[1] Colorado State University, Fort Collins, USA
[2] University of Rennes 1, IRISA, France

Abstract. Model typing brings the benefit associated with well-defined type systems to model-driven development (MDD) through the assignment of specific types to models. In particular, model type systems enable reuse of model manipulation operations (e.g., model transformations), where manipulations defined for models typed by a supertype can be used to manipulate models typed by subtypes. Existing model typing approaches are limited to structural typing defined in terms of object-oriented metamodels (e.g., MOF), in which the only structural (well-formedness) constraints are those that can be expressed directly in metamodeling notations (e.g., multiplicity and element containment constraints). In this paper we describe an extension to model typing that takes into consideration structural invariants, other than those that can be expressed directly in a metamodeling notation, and specifications of behaviors associated with model types. The approach supports contract-aware substitutability, where contracts are defined in terms of invariants and pre-/post-conditions expressed using OCL. Support for behavioral typing paves the way for behavioral substitutability. We also describe a technique to rigorously reason about model type substitutability as supported by contracts, and apply the technique in a usage scenario from the optimizing compiler community.
Key words:SLE, Modeling Languages, Model Typing, Contract Matching, Model Substitutability

1 Introduction

In Model Driven Engineering (MDE), developers of complex software systems create and transform models using model authoring and transformation technologies. The rise in the number of new modeling languages, however, presents a challenge because it requires software engineers to create complex transformations that manipulate models expressed in the new languages. Building these transformations from scratch requires significant effort. To address this problem, various approaches [1][2][3][4][5] have recently been proposed to facilitate the reuse of model transformation across different languages.

Model substitutability rules that are based on model typing [1] can be used to support model transformation reuse. For example, a subtyping relation that supports model substitutability allows a model typed by A to be safely used where a model typed by B is expected, where B is the supertype of A. The transformation used for models typed by B can thus be reused on models typed by A.

Current approaches to model type definition and implementation, however, only consider MOF-based metamodels as model types. In MOF, contracts (e.g., pre-conditions, post-conditions and invariants) are externally defined, that is, they are defined in another language, for example, the Object Constraint Language (OCL) [6]. Neither the original paper on model typing [1] nor the follow-up paper [7] considers externally defined contracts in subtyping relations. This limits the utility of model subtyping in model-based software development approaches that are contract based (e.g., design by contract [8]). There is thus a need for model typing that provides support for typing models with contracts.

In this paper we propose a form of model typing that supports contract-aware substitutability, where contracts are defined in terms of invariants and pre-/post-conditions expressed using OCL. We add invariants to model types that specify additional structural properties, and use operation pre-/post-conditions to specify the transformation rules on model types. We also describe a technique for rigorously reasoning about the substitutability of models with contracts.

The rest of the paper is organized as follows. Section 2 illustrates the need for contract-aware substitutability using motivating examples from the high-performance embedded system design domain. Section 3 presents background material needed to understand the work described in this paper. Section 4 presents a formal definition of the subtyping relation between two model types that include contracts, and describes tool support for reasoning about substitutability on model types. Section 5 describes limitations of the approach. Section 6 discusses related work, and Section 7 concludes the paper with a discussion of planned future work.

2 Motivating Examples

In this section we describe two motivating examples from the high-performance embedded system design domain. Modern heterogeneous embedded hardware platforms are notoriously difficult to design and to program. In this context, tool-supported model based approaches (e.g., Simulink, Ptolemy) are now widely acknowledged as some of the most effective approaches to designing embedded systems.

Typically, these model-based approaches use tool chains that manipulate many different types of models. For example, structural platform description models range from system level models that abstract over processing and storage resource with their interconnections, to very low level Register-to-Logic level circuit models that are used to describe the structure of hardware accelerators within the platforms.

Similarly, behavioral description models range from application level modeling of the application using Models of Computation such as Synchronous Data Flow Graphs or Kahn Process Networks, to fine grain scalar operation level representations such as the basic-block level instruction dependence graph used in an optimizing compiler back-end.

Most of these tool chains share a common goal: They aim to produce highly optimized implementations. This requires the use of advanced algorithms that implement very complex model manipulations. It is also the case that these manipulations often

have similar algorithmic patterns. These patterns can be used as the basis for developing reusable model transformations.

2.1 Example 1: Using Model Types to Support Structural Substitutability

In the optimizing compiler domain, a variety of models describing different aspects of languages are manipulated (i.e., analyzed and transformed) at different stages of the compilation process. While the analyses and transformations are different, they also share many common characteristics. For example, consider algorithms for schedule optimization. Obtaining an optimized implementation of an application on a target platform involves performing several static scheduling optimizations. Many of these algorithms have common characteristics, for example, they are often expressed as an acyclic graph resource constrained scheduling problem, for which many techniques (heuristics or MILP-Mixed Integer Linear Programming-solver based) have been proposed. Because these scheduling algorithms involve very sophisticated algorithms, reusable algorithms that can be tailored to the different types of representations (models) are highly desirable. For example, it would be useful to have a reusable scheduling algorithm that can be used to derive a schedule for an Application level Synchronous Data-flow graph on a multi-processor based implementation, as well as for generating efficient code for a customized VLIW (Very Long Instruction Word) embedded processor.

However in this case, structural substitutability based only on constraints that can be expressed directly in a metamodel (e.g., multiplicity or element containment constraint) is not sufficient; other structural constraints need to be specified. For example, a classical static scheduling toolset can only operate on acyclic dependence graphs and the acyclicity property cannot be expressed directly in a metamodel. A language such as the Object Constraint Language (OCL) is needed to specify properties of acyclic graphs. In this case, model substitutability requires that a substitute model enforces the acyclicity constraint expressed in OCL. Model typing based on metamodels with OCL constraints can be used to enable such structural substitutability.

2.2 Example 2: Using Model Types to Support Contract-based Behavioral Substitutability

Behavioral substitutability for model transformations: A consistent scheduling transformation must ensure that every node in the dependence graph is scheduled *at least* once. This property can be expressed as a post-condition on the scheduling transformation and thus any scheduler implementation should enforce this post-condition. The effective post-condition could even be stricter; in our case we could consider a post-condition that restricts a node to be scheduled *exactly* once.

The same holds for the pre-condition. For example, most schedulers operate on acyclic graphs and this can be translated as a pre-condition for the transformation. However, there also exists a class of pipelined schedulers that operate on cyclic graphs, in which cycles implement delays to preserve causality. For such pipelined schedulers, the pre-condition would not forbid cycles in the dependence graph. That would, however, prevent a pipelined scheduler from being used to schedule acyclic graphs in a design flow.

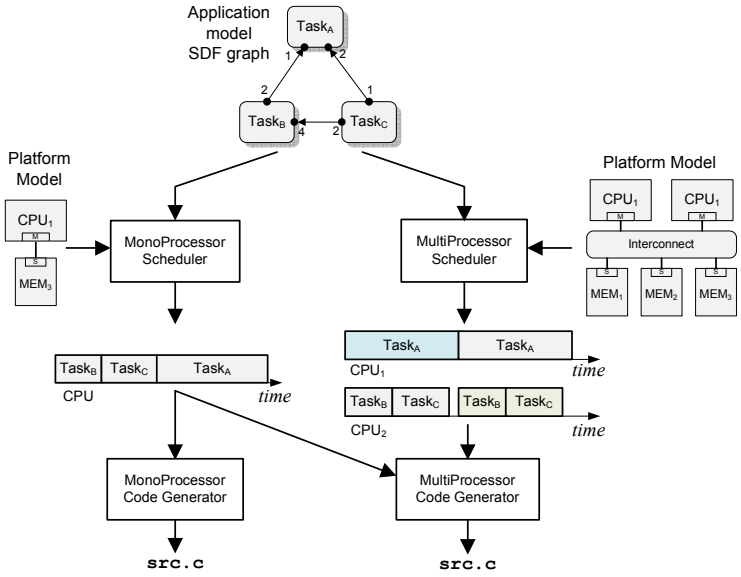

Fig. 1. A Model based compiler tool chain for embedded multiprocessors.

Contract based tool chain validation: An optimizing compiler custom tool chain consists of a sequence of analyses and transformations (called compiler passes) executed in a very carefully chosen order. They can hence be seen as a model transformation chain. Compiler passes cannot be combined arbitrarily, as each pass usually assumes that the program representation at hand has very specific properties.

For example, consider a compiler tool chain for generating software code from Synchronous Data Flow Graph (SDF) model specifications on an embedded platform. Such a tool chain is illustrated in Figure 1. Before any code can be produced, the SDF first needs to be scheduled on this platform. Depending on whether the target system consists of a single or several processors, it is likely that different scheduling algorithms will be used. Similarly, different code generators (i.e. pretty printers) will have to be used depending on whether we target a mono-processor or multiprocessor. Two back-end code generators are shown in Figure 1. The mono-processor code generator can only be used after a mono-processor scheduling stage, whereas the second back-end is more general and can be used for both types of scheduling.

These constraints – that is targeting one or several processing resources – apply to the result of the scheduling stage and to the input on the code generation stage. They can hence be modeled as pre-conditions (resp. post-conditions) expressed using OCL. When chaining a given scheduling and code generation pass, we can ensure the consistency of the flow by checking if the pre-/post-conditions of two chained transformation are satisfiable.

3 Background

In this section, we describe the concepts underlying our use of model types to support model substitutability. We first present the MOF (Meta-Object Facility) meta-language, the basis for metamodels, and thus model manipulation operators. We then give an overview of model types as currently defined and implemented [1,7], and describe the limitations addressed by the approach presented in this paper.

3.1 Metamodeling

The Meta-Object Facility (MOF) [9] 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 modeling languages and tools. A metamodel defines a set of models on which it is possible to apply common operators. The model substitutability approach presented in this paper is applicable to models expressed in languages with MOF metamodels.

MOF supports the definition of metamodels using `Classes` and `Properties`. `Classes` can be abstract (i.e., they cannot be instantiated) and have `Properties` and `Operations`, which respectively declare attributes and references, and the signatures of methods available to the modeled concept. 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 `Properties`) and read-only (i.e., cannot be modified). A `Property` can also have an opposite `Property` with which it forms a bidirectional association.

Metamodels can be viewed as class diagrams in which each metamodel element can be instantiated to obtain objects representing model elements. However, metamodel elements are themeslves 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., `Class` or `Property` classes).

3.2 Model Typing

Model Types were introduced by Steel *et al.* [1], as an extension of object typing to provide abstractions about the object type level and enable the reuse of model manipulation operators. Informally, a model type is a substructure (referred to as a *type group*) of the metamodel's class diagram. It is important to distinguish the usage of the term metamodel from model type. We use the term metamodel to refer to the class diagram used to define a language, and when the same class diagram is used to define the type of a model it is called an *exact type*. It is also important to note that a model has one and only one metamodel to which it must conform, but the same model can have several model types, where each model type is a substructure of the metamodel. Because model types and metamodels share the same structure, it is possible to extract the exact type of a model from its metamodel. Figure 2 represents a model m_1 that conforms to a metamodel MM_1 and is typed by model types MT_A and MT_B, where MT_B is the *exact type* of m_1 that is extracted from MM_1. Both metamodels and model types conform to MOF. Given the above, a model type can be defined as follows:

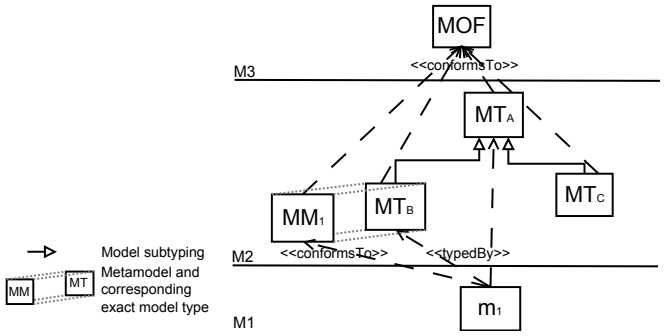

Fig. 2. Conformance, model typing and model subtyping relations

Definition 1. (Model type) *A model type is a substructure of a metamodel's class structure. A model does not have to include instantiations of each class in an associated model type, that is, the set of classes of elements in a model can be smaller than the classes in its model type.*

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 specializations of model substructures (or *type groups*)[1]. One way to safely reuse a model manipulation operation created for a model typed by MT_A on a model typed by MT_B is to ensure that MT_A contains elements that can be substituted by elements defined by MT_B. However, it is not possible to achieve model type substitutability through object subtyping. Thus, model typing uses an extended definition of *object type matching* introduced by Bruce *et al.* [11], namely *MOF Class Matching*.

Definition 2. (MOF class matching) *MOF class T' matches T (written $T' <\# T$) iff their names are equal, and for each property (respectively method) in T there is a corresponding property (respectively method) in T'.*

The *MOF class matching* relation can be seen as a kind of *object type matching* relation that is tailored to MOF concepts. Based on the *MOF class matching* relation, we can achieve model type substitutability by defining a subtyping relation as follows:

Definition 3. (Subtyping relationship for model types) *The model type* subtyping *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 $T_B <\# T_A$.*

We recently introduced four extended subtyping relations between model types that take into account two additional criteria: The presence of heterogeneities between two model types (using adaptation) and the considered subset of the model types (using model type pruning) [7].

[1] For further information on type groups see Ernst's paper [10].

The *subtyping* relation as currently defined has shortcomings. In particular, the current model typing definition and implementation only considers MOF-based metamodels as model types (through the *MOF class matching* relation). Unfortunately, MOF delegates the definitions of contracts (*e.g.*, pre and post-conditions or invariants) to other languages (*e.g.*, OCL, the Object Constraint Language [6]). This limits the applicability of model typing for safely reusing model manipulations where OCL contracts are needed to precisely specify the applicability of the model transformation or the structure on which the model transformation can be applied (see motivating examples in Section 2). The approach described in this paper addresses this limitation.

4 Contribution

In this paper we extend the subtyping relation described in [7] by taking into account OCL contracts for a safe substitutability of models conforming to metamodels including contracts. This provides a safe reuse of model transformations expressed on metamodels that include contracts. Specifically, we extend the MOF class matching (cf. Def. 2 of Section 3) by considering contracts matching (Section 4.1) and provide a technique for analyzing the matching of OCL contracts associated with two classes with different model types (Section 4.2). In this section we describe the contract matching technique we developed to support contract-aware model substitutability. We also describe an Alloy-based prototype tool that supports contract matching (Section 4.3), and illustrate the use of contract-aware substitutability using the motivating examples (Section 4.4).

4.1 Contract-aware MOF Class Matching

We consider the use of OCL invariants added to MOF classes to specify additional structural properties, and OCL pre-/post-conditions defined in the context of MOF class operations to specify the model manipulation rules (e.g., transformation) associated with model types. The MOF class matching relation is thus determined by two aspects: the structural features specified using MOF (e.g., classes, properties, operation signatures, etc.) and the contracts expressed using OCL (e.g., invariants and pre-/post-conditions).

The substitutability through model subtyping is a specialization of the Liskov Substitution Principle [12] on the model type system. Specifically the contract matching relation that enables contract-aware model substitutability must abide by the following rules: (1) invariants of the supermodel type cannot be weakened in a sub model type, (2) pre-conditions cannot be strengthened in a sub model type, and (3) post-conditions cannot be weakened in a sub model type. The extended MOF class matching relation is formalized as follows:

Definition 4 (Contract-aware MOF Class Matching). *Class T' matches T (written T' <# T) iff their structures match (cf. Def. 3 of [7]), their invariants match and their operation pre-/post-conditions match, where*

1 ***Invariants Match*** *is defined as follows:*
let T.ownedInvariant = $\{inv_{T1}, inv_{T2}, ..., inv_{Tk}\}$ be the invariants defined for T;
let $result_T = inv_{T1} \wedge inv_{T2} \wedge ... \wedge inv_{Tk}$;

let SuperClass(T) = $\{cls_1, cls_2, ..., cls_n\}$ where cls_i is a superclass of T;
let cls_i.ownedInvariant = $\{ inv_{i1}, inv_{i2}, ..., inv_{ik} \}$ be the invariants defined for cls_i, for i = 1, .., n;
let $result_i = inv_{i1} \wedge inv_{i2} \wedge ... \wedge inv_{ik}$, for i = 1, .., n;
let $invs = result_1 \wedge result_2 \wedge ... \wedge result_n \wedge result_T$;

let T'.ownedInvariant = $\{inv'_{T1}, inv'_{T2}, ..., inv'_{Tk} \}$ be the invariants defined for T';
let $result'_T = inv'_{T1} \wedge inv'_{T2} \wedge ... \wedge inv'_{Tk}$;
let SuperClass(T') = $\{cls'_1, cls'_2, ..., cls'_n\}$ where cls'_i is a superclass of T';
let cls'_i.ownedInvariant = $\{ inv'_{i1}, inv'_{i2}, ... , inv'_{ik} \}$ be the invariants defined for cls'_i, for i = 1, .., n;
let $result'_i = inv'_{i1} \wedge inv'_{i2} \wedge ... \wedge inv'_{ik}$, for i = 1, .., n;
let $invs' = result'_1 \wedge result'_2 \wedge ... \wedge result'_n \wedge result'_T$;

The invariants of T and T' match if $Models(invs) \supseteq Models(invs')$, where $Models(invs)$ returns all models that satisfy invs and $Models(invs')$ returns all models that satisfy $invs'$.

2 ***Pre-/post-conditions Match*** *is defined as follows:*
$\forall$ op $\in$ T.ownedOperation, $\exists$ S' $\in$ SuperClasses(T') such that $\exists$ op' $\in$ S'.ownedOperation and:

2.1 *let op.ownedPrecondition = $\{pre_1, pre_2, ..., pre_k \}$ be the pre-conditions defined for op;*
let $pres = pre_1 \wedge pre_2 \wedge ... \wedge pre_k$;
let op'.ownedPrecondition = $\{pre'_1, pre'_2, ..., pre'_k \}$ be the pre-conditions defined for op';
let $pres' = pre'_1 \wedge pre'_2 \wedge ... \wedge pre'_k$;

2.2 *let op.ownedPostcondition = $\{post_1, post_2, ..., post_k \}$ be the post-conditions defined for op;*
let $posts = post_1 \wedge post_2 \wedge ... \wedge post_k$;
let op'.ownedPostcondition = $\{post'_1, post'_2, ..., post'_k \}$ be the post-conditions defined for op';
let $posts' = post'_1 \wedge post'_2 \wedge ... \wedge post'_k$;

The operation specifications of T and T' match if $Models(pres') \supseteq Models(pres)$ and $Models(posts) \supseteq Models(posts')$

4.2 Analyzing the Matching of Contracts

Definition 4 can be used to formally reason about the matching relation between two MOF classes with contracts. The MOF class matching relation in Definition 4 includes the matching of the contracts from classes of two model types. Consequently, analyzing

such relations requires one to formally analyze the relation between contracts (e.g., to check if the models satisfying one contract includes the models satisfying the other). To do this, a query function $Models(MT, C)$ is used to compute all models that both conform to a model type MT and satisfy an OCL contract C defined in MT. Thus given contract C_1 in a candidate supermodel type MT_1 and contract C_2 in a candidate sub model type MT_2, C_1 matches C_2 iff (1) C_1, C_2 are invariants, and $Models(MT_1, C_1) \supseteq Models(MT_2, C_2)$, (2) C_1, C_2 are pre-conditions, and $Models(MT_2, C_2) \supseteq Models(MT_1, C_1)$, and (3) C_1, C_2 are post-conditions, and $Models(MT_1, C_1) \supseteq Models(MT_2, C_2)$.

Checking the contract matching requires a tool to implement the functionality of the query function $Models(MT, C)$. We use the Alloy Analyzer [13] for this purpose. The Alloy Analyzer is used to analyze Alloy specifications. It is supported by a SAT-based model finder. The Alloy Analyzer can generate models that conform to a model type expressed in Alloy in terms of signatures and fields that specify the model type structure and a predicate that expresses the contracts. In this paper we use the Alloy Analyzer at the back-end to check whether two contracts match.

For example, given a candidate supermodel type MT_1 and a candidate sub model type MT_2, with two OCL invariants respectively, C_1 and C_2, the procedure below can be used to check if C_1 matches C_2.

1. (preprocess) Since model subtyping requires each element in the supermodel type to be matched by an element in the sub model type (see Definition 4), the contract defined in the supermodel type refers to elements that also exist in the sub model type. Thus we can move C_1 to MT_2, and use only the sub model type (i.e., MT_2) to check whether C_1 and C_2 match.
2. Transform MT_2 to an Alloy model using the technique described in [14]. Convert C_1 and C_2 into two Alloy predicates, P_1 and P_2, respectively.
3. Run an empty predicate in the Alloy Analyzer to search for a model conforming to the model type MT_2. If the Analyzer returns no model satisfying the empty predicate (i.e., $Models(MT_2, \emptyset) = \emptyset$), $Models(MT_2, C_1) = \emptyset$ and $Models(MT_2, C_2) = \emptyset$. In this case C_1 matches C_2 since $\emptyset$ is a subset of $\emptyset$; otherwise, continue to the next step.
4. Run P_1 and P_2 respectively. If the Alloy Analyzer returns no model for each predicate (i.e., $Models(MT_2, C_1) = \emptyset$ and $Models(MT_2, C_2) = \emptyset$), then C_1 matches C_2; if the Alloy Analyzer returns a model (or models) for only P_1, then C_1 matches C_2; if the Alloy Analyzer returns a model (or models) for only P_2, then C_1 does not match C_2; otherwise, continue to the next step.
5. Run a predicate to search for a model satisfying both P_1 and P_2. If the Alloy Analyzer returns a model satisfying the predicate, continue to the next step; otherwise, C_1 does not match C_2.
6. Run a predicate P_3 to search for a model satisfying both P_1 and $\neg P_2$ (i.e., the negation of P_2), and another predicate P_4 to search for a model satisfying both P_2 and $\neg P_1$. If the Alloy Analyzer returns no model for both P_3 and P_4 (i.e., $Models(MT_2, C_1) = Models(MT_2, C_2)$), C_1 matches C_2; if the Alloy Analyzer returns a model (or models) satisfying only P_3, $Models(MT_2, C_1) \supset Models(MT_2, C_2)$ and C_1 matches C_2; otherwise, C_1 does not match C_2.

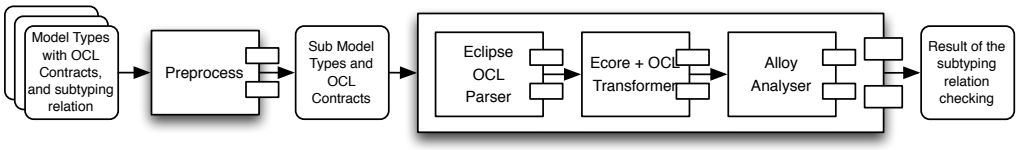

Fig. 3. Contract Matching Checking Tool Overview

The approach uses the Alloy Analyzer at the back-end to analyze the relation between two contracts, and it thus requires a translation from OCL expressions to Alloy specifications. The OCL to Alloy translation used in the prototype tool we developed is based on translation rules described in work by Bordbar et al. [15].

4.3 Contract Matching Checking Tool

The contract matching approach described in the previous subsection has been implemented in a prototype tool. Figure 3 shows an overview of the prototype tool. It consists of an OCL parser, an Ecore[2]/OCL transformer and the use of the Alloy Analyzer. The Ecore/OCL transformer is developed using Kermeta [17], an aspect-oriented metamodeling tool. The inputs of the prototype are (1) an Ecore file that specifies two model types, and (2) a textual OCL file that specifies the contracts from each model type. The model types and contracts are automatically transformed to an Alloy model consisting of signatures and predicates.

The prototype provides several interfaces to check contract matching. For example, *matchInv(inv1: Constraint, inv2: Constraint)* is used to check whether *inv*1 matches *inv*2. In addition, *matchInvs(cls1: Class, cls2: Class)* can be used to check whether the invariants defined in *cls*1 and the invariants defined in *cls*2 match.

4.4 Case Study

In this section we illustrate how to use our approach to define model types and subtyping relations between them to ensure a safe reuse of model transformations.

A Simple Case Study of Structural Substitutability Let us reconsider the scheduling example described in Section 2.1. A model transformation performs a static scheduling on an acyclic dependence graph. The model transformation needs a metamodel for "Acyclic Graph" (due to space limitation, the metamodel is not shown in the paper). The model type *AcyclicGraph* (see Figure 4) shows a simple example of model type definition for the dependency graph used in the example. Its definition consists of metaclasses that specify a graph structure, an invariant that specifies the acyclicity property, and a model transformation that takes as input an acyclic graph.

[2] Ecore is an implementation aligned with MOF included in the Eclipse Modeling Framework [16].

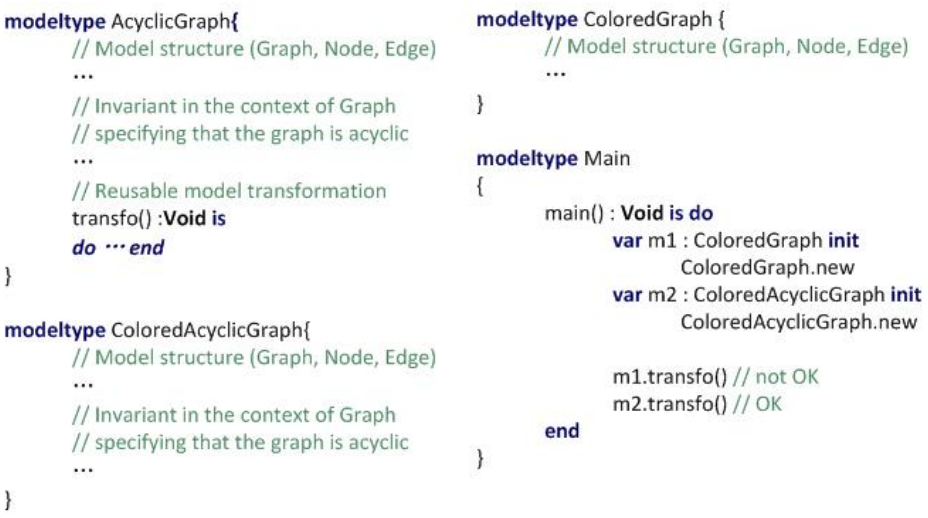

Fig. 4. A Simple Example of Structural Substitutability in Kermeta

Suppose that in another context a colored graph is used as an intermediate representation and it extends the concept of nodes by introducing additional information. To reuse the transformation defined in *AcyclicGraph*, a colored graph must be a subtype of *AcyclicGraph*. The model type *ColoredAcyclicGraph* ensues the subtyping relation by adding an acyclicity invariant in its definition. However, the model type *ColoredGraph* does not specify any invariants. A compilation error will thus show that the $transfo$ operation cannot take as input an instance of *ColoredGraph* because *ColoredGraph* is not a subtype of *AcyclicGraph*.

A Simple Case Study of Behavioral Substitutability In the optimizing compiler community, the daily task for software engineers is to design compilation chains in the right partial order, that is, scheduling the various passes (i.e., optimization, translation, code generation, analysis, etc.). Designing compilation chains would benefit from behavioral substitutability by opening the way to describe "abstract" compilation chains, capitalizing a given knowledge in terms of constraints (pre-/post-conditions) to schedule a set of passes for a given purpose, where each pass would be then implemented in various ways, but conforming to the pre-/post-conditions defined in the abstract compilation chain.

Figure 5 shows a simple example of model types used for the compilation chain. Suppose that the abstract model transformation *transfo* defined in *MT* is used for optimization purpose and define a post condition stating that the model must conform to the Static Single Assignment (SSA) form. *MT* also contains *transfo2* as the next pass of the compilation chain and states as precondition that the model must conform to the SSA form. The two model types $subMT1$ and $subMT2$ implement the model transformation *transfo* but only $subMT1$ ensures as postcondition the SSA form. While $subMT2$ is not

```
modeltype MT{
        // Model structure and invariant
        ...
        // Abstract transformation
        transfo() : MT is abstract
                post:   SSA // OCL expression
        // Reusable model transformation
        transfo2() : MT is do ... end
                pre:    SSA
}

modeltype subMT1{
        // Model structure and invariant
        ...
        // Transformation specialization
        transfo() : subMT1 Is
        do ... end
                post:   SSA
}
```

```
modeltype subMT2{
        // Model structure and invariant
        ...
        // Transformation specialization
        transfo() : subMT2 Is
        do ... end
}

modeltype Main
{
        main() : Void is do
                var m1 : subMT1 init
                        subMT1.new
                var m2 : subMT2 init
                        subMT2.new

                m1.transfo().transfo2() // OK
                m2.transfo().transfo2() // not OK
        end
}
```

Fig. 5. A Simple Example of Behavioral Substitutability in Kermeta

in this case a sub model type to MT, a compilation error for $m2.transfo().transfo2()$ will be returned. This shows that the model returned by *transfo* in $subMT2$ (typed by $subMT2$) is not of type MT, and can not reuse *transfo2*.

5 Discussion

In this section we discuss limitations of our work, and its scope of application. We first discuss the supported contracts in the subtyping relation of model typing (Section 5.1), and the corresponding model substitutability provided by our approach (Section 5.2).

5.1 On the Support of Contracts in Model Typing

In this paper we consider contracts in addition to the object oriented structure described in a metamodel. The object-oriented structure is usually defined using Ecore, an implementation aligned with OMG MOF. Contracts can then be invariants expressed in the context of the concepts (i.e., classes) defined in the MOF metamodel, and pre-/post-conditions expressed in the context of operations specified in concepts. While invariants restrict the structure of conforming models and their possible structural substitutability, pre- and post-conditions specify the behavior of the conforming models (i.e. manipulation by model operations) and their possible behavioral substitutability.

In our approach, we assume that the first order logic is used to express contracts in metamodels, and we have chosen OCL to express them. We rely on the provided binding

between MOF and OCL as defined by OMG to link OCL expressions to a given MOF metamodel.

To test the feasibility of our approach, we implemented a prototype tool that is integrated into the Kermeta workbench. The tool checks OCL-based contract-aware subtyping relations between model types. While the substitutability related to the MOF structure is computed directly using Kermeta, the one related to the contracts is computed using Alloy through a translation from OCL expressions to Alloy specifications, and then an analysis of the output provided by the Alloy Analyzer. The tool only provides support for translating a subset of OCL to Alloy.

Most of the OCL operators have corresponding Alloy constructs. For example, OCL operator $forAll$ corresponds to Alloy construct *all*, *exists* corresponds to *some*, *includes* corresponds to *in*, *excludes* corresponds to !*in*, *sum* corresponds to *sum*, and *closure* corresponds to $*$. OCL contracts that involves such operators can be directly transformed into Alloy specifications.

However, as pointed out by Anastasakis et al. [15], the translation from OCL to Alloy is not seamless. There are some OCL operators that do not have corresponding Alloy constructs, and thus OCL contracts including such operators cannot be easily transformed into Alloy specifications. Some of them can be partially supported by the tool using the Alloy libraries. For instance, OCL operators like *select* and *collect* are translated by the tool described in the paper using Alloy functions that implement their semantics. Consequently, the operator $iterate$ is partially supported by the transformation tool. The tool provides support for OCL contracts including $iterate$ expressions that can be rewritten as $forall$ with $select/collect$ operators. However, the tool cannot be used to deal with $iterate$ expressions that involve arithmetic accumulation since Alloy is a purely declarative language that does not provide support for imperative accumulators. Finally, the translation cannot deal with OCL casting operators like *oclAsType* since Alloy has a very simple type system that has little support for type casting.

5.2 On the Support of Modeling Language Substitutability

The research work described in the paper builds upon our previous work in [7], and paves the way for reasoning about the subtyping relation between two model types that include contracts. Specifically it can be used to reason about the contract-aware subtyping relation that involves structural subtyping (including not only MOF-based Object-Oriented structure but also OCL-based first order invariants) and behavioral subtyping (including a behavioral semantics in an axiomatic way using pre-/post-conditions on operations).

We implement our approach in a (Kermeta-based) tool included in the Kermeta language workbench to check advanced (i.e., including contracts) subtyping relations between modeling languages based on Ecore and OCL. These two meta-languages are supported by the Kermeta language workbench and are used for describing the abstract syntax and the static semantics respectively.

This approach and its corresponding implementation addresses the need illustrated by the motivating examples from the high-performance embedded system community used throughout the paper. The scope of the structural substitutability we offer

is bounded by OCL and its translation to Alloy, and its applicability is well founded, e.g., in model transformation reuse in model-driven development [7].

The actual scope of behavioral substitutability is more difficult to define. The difficulty is twofold: while we support the motivating examples described in the paper, complex situations of OCL based typing could be considered, such as type propagation in model transformation chains. Such challenges will be addressed in future work. Moreover, the scope itself of behavioral substitutability is more difficult to delimit.

6 Related Work

The technical contribution of this paper is the integration of contract matching in the subtyping relation of model typing to enhance the substitutability supported between modeling languages. As discuss in the previous section, we rely for that on the most established translation to Alloy. Then, the work related to our contribution discussed in this paper is the applicability of the substitutability as illustrated in the motivating examples, namely on model transformation reuse.

Substitutability is supported through subtyping in object-oriented languages, including the support of contracts (*e.g.*, Eiffel [18]). However, object subtyping does not handle type group specialization (*i.e.*, the possibility to specialize relations between several objects and thus groups of types)[3]. Such type group specialization have been explored by Kühne in the context of MDE [19]. 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.

Several approaches have been proposed during the last decade for model transformation reuse. Strict substitutability relation, such as the first version of model type matching presented in [1], offers the possibility to reuse model transformation through isomorphic metamodels, i.e., metamodels with MOF-based equivalent structures. Such possibility was first proposed in [2] where the authors introduce *variable entities* in patterns for declarative transformation rules. These entities express only the needed concepts (*e.g.*, types, attributes, etc.) to apply the rule, allowing any tokens with these concepts to match the pattern and thus to be processed by the rule. Latter, Cuccuru *et al.* introduced the notion of semantic variation points in metamodels [3]. Variation points are specified through abstract classes, defining a *template*, and metamodels can fix these variation points by binding them to classes extending the abstract classes. Patterns containing *variable entities* and *templates* can be seen as kinds of model types where the variability has to be explicitly expressed and thus anticipated. Sanchez Cuadrado *et al.* propose in [4] a notion of substitutability based on model typing and model type matching, but rather to use an automatic algorithm to check the matching between two model types, they propose a DSL that allows users to declare the matching by hand. Finally, De Lara *et al.* present in [5] the *concept* mechanism, along with *model templates* and *mixin layers* leveraged from generic programming to MDE. *Concepts* are really close to

[3] We refer the reader interested in the type group specialization problem to the Ernst's paper [10].

model types as they define the requirements that a metamodel must fulfill for its models to be processed by a transformation, under the form of a set of classes. The authors also propose a DSL to bind a metamodel to a *concept* and a mechanism to generate a specific transformation from the binding and the generic transformation defined on the *concept*.

In the context of model transformation chains, existing approaches deal with explicit relationships between model transformations. Vanhooff et al. [20] proposed a domain specific language to model and execute a transformation chain. Aranega et al. [21] used feature models to classify model transformations involved a transformation chain and specified the constraints between them. The user thus can design a new transformation chain by reusing the classified transformations. Yie et al. [22] advocated the use of several independently developed model transformation chains to convert a high-level model into a low-level model. The interoperability among model transformation chains is achieved by deriving correspondence relationships between the final models generated by each model transformation chain.

Unlike the above approaches, contract-aware model subtyping offers a unified and formal type theory to facilitate the safe reuse of model transformations involved in a transformation chain. It follows a declarative fashion to specify a model transformation chain in an abstract way using pre-/post-conditions on abstract model types. This promotes then the reuse of various implementations that match the conditions for a safe execution of the model transformation chain.

7 Conclusion and Perspective

We propose in this paper a model typing theory where model types include contracts. This includes a formally defined subtyping relation between model types, and a tool-supported approach supporting a safe contract-aware substitutability of models conforming to metamodels including contracts. This ensures a safe reuse of model transformations expressed on metamodels including contracts.

Contracts are defined in terms of invariants and pre-/post-conditions expressed using OCL on MOF-based metamodels. The invariants are added on the classes of a metamodel to specify additional structural properties of the metamodel, and pre-/post-conditions are added on the operations of classes to specify model transformations. Consequently, the support of invariants in the subtyping relation ensures a safe reuse of model transformations where OCL contracts are needed to precisely specify the structure on which the model transformation can be applied. The support of pre-/post-conditions paves the way for behavioral substitutability to safely reuse model transformations where OCL contracts are needed to precisely specify the applicability of the model transformation.

The subtyping relation is based on a matching relation between two MOF classes that include OCL contracts, and is checked thanks to a technique based on Alloy. The actual scope of the provided contract-aware substitutability is mainly determined by the OCL-to-Alloy translation.

We are currently extending the prototype by providing support for model types and contracts expressed using the Kermeta language workbench. We also explore how we

can extend the approach by using SMT solvers at back-end to analyze the OCL contracts that include more complex arithmetic calculation.

Acknowledgment This work was supported by the National Science Foundation grant (CCF-1018711), the ANR INS Project GEMOC (ANR-12-INSE-0011), and the CNRS PICS Project MBSAR.

References

1. Steel, J., Jézéquel, J.M.: On model typing. SoSyM **6**(4) (2007)
2. Varró, D., Pataricza, A.: Generic and meta-transformations for model transformation engineering. In: « UML» 2004-The Unified Modeling Language. Modelling Languages and Applications. Springer (2004) 290–304
3. Cuccuru, A., Mraidha, C., Terrier, F., Gérard, S.: Templatable metamodels for semantic variation points. In: ECMDA-FA. (2007)
4. Sánchez Cuadrado, J., García Molina, J.: Approaches for model transformation reuse: Factorization and composition. In: ICMT. (2008)
5. de Lara, J., Guerra, E.: Generic meta-modelling with concepts, templates and mixin layers. In: MODELS: Part I. MODELS (2010)
6. OMG: UML Object Constraint Language (OCL) 2.0 Specification. (2003)
7. Guy, C., Combemale, B., Derrien, S., Steel, J., J'ez'equel, J.M.: On Model Subtyping. In: ECMFA. Number 7349 in LNCS, Springer (July 2012) 400–415
8. Meyer, B.: Applying'design by contract'. Computer **25**(10) (1992) 40–51
9. OMG: Meta Object Facility (MOF) 2.0 Core Specification. (2006)
10. Ernst, E.: Family polymorphism. In: ECOOP 2001 Object Oriented Programming. Springer (2001) 303–326
11. Bruce, K.B., Schuett, A., van Gent, R., Fiech, A.: Polytoil: A type-safe polymorphic object-oriented language. ACM TOPLAS **25**(2) (2003)
12. Liskov, B., Wing, J.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(6) (1994) 1811–1841
13. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology (TOSEM) **11**(2) (2002) 256–290
14. Sun, W., France, R., Ray, I.: Rigorous analysis of uml access control policy models. In: IEEE POLICY. (2011) 9–16
15. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from uml to alloy. Software and Systems Modeling **9**(1) (2010) 69–86
16. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Addison-Wesley Professional (2008)
17. Muller, P., Fleurey, F., Jézéquel, J.: Weaving executability into object-oriented meta-languages. Model Driven Engineering Languages and Systems (2005) 264–278
18. Meyer, B.: Design by contract. the eiffel method. In: Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings. (1998) 446–446
19. Kühne, T.: On model compatibility with referees and contexts. Software & Systems Modeling (2012) 1–14
20. Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., Berbers, Y.: Uniti: A unified transformation infrastructure. Model Driven Engineering Languages and Systems (2007) 31–45
21. Aranega, V., Etien, A., Mosser, S.: Using feature model to build model transformation chains. Model Driven Engineering Languages and Systems (2012) 562–578
22. Yie, A., Casallas, R., Deridder, D., Wagelaar, D.: Realizing model transformation chain interoperability. Software and Systems Modeling (2012) 1–21