Proving Model Transformations

Hung Ledang
ALTRAN ASD
2 rue Paul Vaillant Couturier
92300 Levallois-Perret, FRANCE
hung.ledang@altran.com

Hubert Dubois
CEA, LIST
Point Courrier 94
91191 Gif-sur-Yvette, FRANCE
hubert.dubois@cea.fr

Abstract

Within the MDA context, model transformations (MT) play an important role as they ensure consistency and significant time savings. Several MT frameworks have been deployed and successfully used in practice. Like for any software, the development of MT programs is error prone. However there is limited support for verification and validation in current MDA technologies.
This paper presents an approach to prove model transformations. Model transformations are firstly formalized in B. Then the B provers will be used to analyze and prove the correctness of transformation rules w.r.t. metamodels and transformation invariants. We also analyze and prove the consistency of transformation rules w.r.t. each other.

Keywords: *model transformation, B method, transformation rule, transformation invariant, B abstract machine, substitution*

1 Introduction

The Model Driven Architecture (MDA) framework proposed by OMG has become a central interest of many large industrial groups in designing and implementing their software applications [7, 1, 18]. A set of standards (UML, MOF, QVT) is defined to express models, model relationships, and model-to-model transformations [4]. The MDA considers models as the first-class assets in software development. Models are manipulated, stored, and modified by tools. Expressed in well-defined notations (such as UML), models create a cornerstone to understand systems for large-scale solutions.
Model transformations (MT) encapsulate techniques to manipulate and create models. As shown in Fig. 1, a MT program implements a set of transformation rules which specify how to generate one or several target models (TM) from one or several source models (SM)[10]. The target models must conform to the target metamodels (TMM) provided that the source models conform to the corresponding source metamodels (SMM). The example represented in Fig. 1 is used throughout the paper and further details will be introduced when necessary. The use of metamodels as a formal underpinning is the basis for automation through tools. Metamodels define not only the syntax but also the semantics (operational semantics, axiomatic semantics) of source and target models [17].

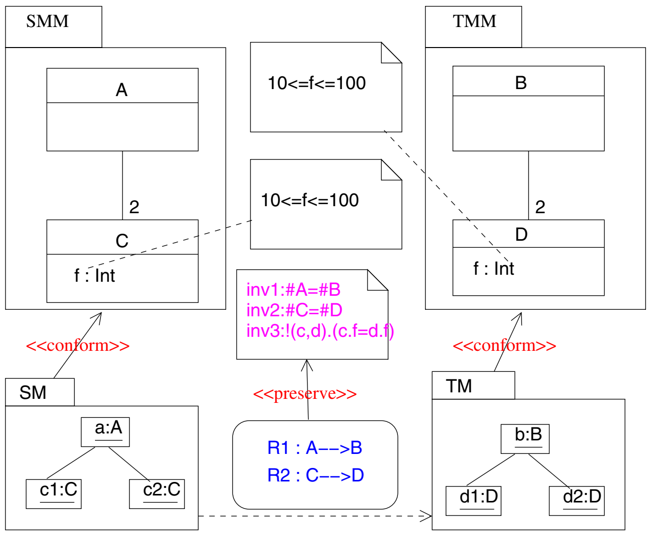

Figure 1. Model Transformation

The automation of non-trivial model transformations ensures consistency and significant time savings [1]. The development of MT programs should be considered as a software development. The metamodels, the transformation invariants, and the transformation rules, which constitute essentially the requirement documents of MT programs, should be clearly understood and checked both semantically and technically (correctness, consistency and termination), then underwent by a careful design and implementation. We also expect to verify the validity of execution of transformation rules even before any design and implementation attempt:

1. we have to validate source models and eventually input parameters of transformation rules;
2. provided that the source models conform to the source metamodels and the eventual preconditions of transformation rules, we have to verify that the execution of transformation rules produces target model elements that respect target metamodels;
3. we also need to ensure that the execution of transformation rules preserves the transformation invariants, which, by definition, represent properties on traceability links between the source-model elements and the target-model elements.

Current MT technologies such as the ATL toolkit [10] and the IBM Rational Rose XDE MDA toolkit [1] provide frameworks to implement, execute, and deploy MT programs but they provide very limited functionality to check the correctness of transformation rules. The IBM Rational Rose XDE MDA toolkit provides "empty" Java methods for verifying and validating the transformation; but such methods should be implemented by MT developers. With the ATL toolkit, we can check the syntax of source-model elements and specify the precondition of transformation rules; but we can not check whether a transformation rule produces a target-model elements respecting target metamodels. None of the current MT technologies propose the way to check the correctness of transformation rules w.r.t. transformation invariants, which, in our opinion, is a very important aspect in the verification of model transformations.

1.1 A solution to prove model transformations

In our opinion, a general solution to deal with three issues mentioned above is to:

1. formalize metamodels in an appropriate formal notation;
2. represent the source models using the proposed formal notation;
3. formalize the transformation rules and prove that they respect the source and target metamodels. The target models are formalized during the formalization of transformation rules. The formalization of transformation rules must conform to their execution semantics, i.e., there is only one transformation rule applied on each source-model element;
4. formalize explicitly transformation invariants and prove that the execution of transformation rules respects all stated transformation invariants;
5. prove the consistency of transformation rules w.r.t. each other.

In the sequel, we will explain in detail this approach through an example. Section 2 briefly presents the B language and method and explains our motivation of using B in proving model transformations. Section 3 describes the formalization of metamodels and source models getting involved in a transformation (points *1* and *2*). The formalization of transformation rules and transformation invariants (points *3* and *4*) is presented in Sect. 4. The consistency among transformation rules (point *5*) is formalized and analyzed in Sect. 5. In each section, we also present the way we use B tools to prove and analyze the MT-related consistency issues (points *3*, *4* and *5*). In Sect. 6, we summarize different contributions related to verification then validation for the MDA and identify similarities and differences between these contributions and our approach. Conclusions summarizing our contributions and sketching the perspectives in Sect. 7 will close our paper.

2 The B language and method

B [2] is a formal software development method that covers the software process from specifications to implementations. The B method has been adopted for use in industry [3]. The B notation is based on Zermelo-Frankel set theory and first order logic. Specifications are composed of abstract machines similar to modules or classes; they consist of variables, invariance properties related to those variables and operations. The state of the system, i.e. the set of variable values, is only modifiable by operations. The means by which B operations specifies state transitions is the *generalised substitution language* whose semantics is defined by means of predicate transformers [9] and the weakest precondition [6]. A generalised substitution is an abstract mathematical programming construct, built up from basic substitution $x := e$, corresponding to assignments to state variables, via a set of operators like No-op ($skip$), bounded choice (**choice** S_1 **or** S_2 ...), preconditioning (**pre** P **then** S **end**), unbounded non-determinism (**var** v **in** S **end**, **any** v **where** P **then** S **end**), guarding, sequential composition, multiple generalised composition and looping.
The abstract machine can be composed in various ways. Thus, large systems can be specified in a modular way, possibly reusing parts of other specifications. B refinement can be seen as an implementation technique but also as a specification technique to progressively augment a specification with more details until an implementation that can then be

translated into a programming language like ADA, C or C++. At every stage of the specification, proof obligations ensure that operations preserve the system invariant. A set of proof obligations that is sufficient for correctness must be discharged when a refinement is postulated between two B components.
These characteristics make B a good choice to formalize MOF-metamodels which, by definition, describe concepts, their attributes, and the relationships among these concepts. In this paper, we will demonstrate that B is also suitable for formalizing transformation rules and invariants thanks to its semantics of generalized substitutions. Our objective is not to develop MT programs using B. Instead, we intend to use B as a tool to analyze and prove the correctness of model transformations. Please refer to the B Book [2] for a complete description. We also try to present a comprehensive description of every B element encountered in this paper.

3 Metamodels and source models

```
MODEL MM
SETS OBJECTS
CONSTANTS SA, SC, TB, TD
PROPERTIES
     SA ⊆ OBJECTS ∧ SC ⊆ OBJECTS ∧
     TB ⊆ OBJECTS ∧ TD ⊆ OBJECTS
END
```

Figure 2. Abstract Machine *MM*

We consider here MOF-compliant metamodels. Such metamodels use class diagrams to describe concepts, their attributes, the relationships among concepts, and the constraints (axiomatic semantics) in those concepts. In Fig. 1, the source metamodel SMM describes two concepts represented by classes A and C. The association between A and C has the multiplicity of 2 at the C-end. The class C has an attribute f defined as an integer with a value, which is limited in 10..100. The target metamodel TMM is composed of two classes B and D. The B-D association has the multiplicity of 2 at the D-end. The class D has an attribute f defined as an integer with a value in the range 10..100. Successfully applied in different ATL transformation examples[1] (BookToPublication, ClassToRelational, UMLToAmble), all expected aspects of our approach, however, are not fully demonstrated by each of these examples. The example in Fig. 1 will be therefore used in the sequel to explain our approach.
Fig. 2 presents ***MM***, a B abstract machine, derived from SMM and TMM metamodels. We define the set *OBJECTS* to formalize the instance space of all classes in metamodels. The constants *SA* and *SC* formalize respectively the instance space of the classes A and C. The constants *TB* and *TD* formalize respectively the instance space of the classes B and D. *SA*, *SC*, *TB*, and *TD* are therefore defined (cf. the clause **PROPERTIES** in Fig. 2) as subsets of *OBJECTS*. Each constant derived from source metamodels is prefixed by *S*; each constant derived from target metamodels is prefixed by *T*.
Given a model transformation, the source models are formalized in ***Input***, a dedicated B abstract machine. We need to formalize all characteristics defining elements in source models as described in the source metamodels: classes, attributes, associations, and related constraints. These formalizations are based on the previous work of Meyer and Souquières [16]. Fig. 3 shows the abstract machine ***Input*** defined for a source model that conforms to the metamodel SMM.

```
MODEL Input
SEES MM
VARIABLES ia, ic, iac, icf
INVARIANT
  ia ⊆ SA ∧ ic ⊆ SC ∧ icf ∈ ic → NAT ∧ ran(icf) ⊆ 10..100 ∧
  iac ∈ ia ↔ ic ∧ iac⁻¹ ∈ ic → ia ∧ dom(iac) = ia ∧
  ∀xx.(xx ∈ ia ⇒ card(iac[{xx}]) = 2)
INITIALISATION
   any sa , sc, sac, scf where
        sa ⊆ SA ∧ sc ⊆ SC ∧ scf ∈ sc → NAT ∧
        ran(scf) ⊆ 10..100 ∧ sac ∈ sa ↔ sc ∧ sac⁻¹ ∈ sc → sa ∧
        dom(sac) = sa ∧ ∀xx.(xx ∈ sa ⇒ card(sac[{xx}]) = 2)
   then ia, ic, iac, icf := sa, sc, sac, scf end
END
```

Figure 3. Abstract Machine *Input*

The variable *ia* represents the set of effective instances of the class A in the source model. It is therefore defined as a subset of the constant *SA* (cf. clause **INVARIANT** in Fig. 3). The reason is that *SA* represents the A-instance space, while *ia* represents the set of effective A-instances. The variable *ic* is defined in the same way. The constant *SA* in the abstract machine ***MM*** is visible in ***Input*** thanks to the **SEES** link between ***Input*** and ***MM***. Please note that the ***MM*** machine is be referenced by every machine created during the formalisation by the **SEES** clause which allows such machines reference to the sets and constants defined in ***MM***.
The attribute C::f is formalized by the variable *icf* which is defined as a binary relation between *ic* and **NAT**, *ic* is derived from the class C, and **NAT** is a B predefined type that corresponds to the type Int. In addition, we add a predicate limiting the range of *icf*, which corresponds to the constraint attached to C::f in SMM.
The association A-C is formalized by the variable *iac* which is defined as a binary relation between *ia* and *ic*. Since the multiplicity at the A-end is 1, the inverse relation of *iac* is defined as a total function from *ic* to *ia*. The multiplicity at the C-end indicates that there are exactly two instances of C

[1] http://www.eclipse.org/m2m/atl/atlTransformations/

for each A-instance; this constraint is expressed by two last predicates in the clause **INVARIANT**.
The variables defined in ***Input*** will be referenced during the formalization of transformation rules (Sect. 4). In ***Input***, the variables are prefixed by *i* to indicate that they formalize the transformation input elements.
The clause **INITIALISATION** in ***Input*** uses a substitution **any** to initialize the variables defined in this abstract machine. In the substitution **any**, for each variable which needs to be initialized, we declare a corresponding variable which respects the same conditions as invariant predicates. The reason is that the transformation can accept the input from any source models respecting the constraints defined in source metamodels.

4 Transformation rules and invariants

We consider that the transformation rules, irrespective to their implementation technologies (ATL, Smart QVT, etc), describe the way to generate target-model elements from source-model elements. In Fig. 1, the rule R_1 indicates that for each A-instance, we produce a B-instance together with two D-instances which are obtained from the C-instances associated with the A-instance in question by applying R_2. The rule R_2 indicates that, for each C-instance, we produce a D-instance and the value of D::f is copied from the corresponding C::f. Thus, abstractly, transformation rules can be formalized as operations in B abstract machines (or B abstract operations for short).

4.1 Formalization requirements

While the formalization of metamodels and source-models is based on the previous works [16, 15], the formalization of transformation rules is one of our new contributions in this paper. The formalization of transformation rules has to satisfy following requirements:

1. **execution semantics faithfulness**: each transformation rule is applied only once to each source-model element matched with its precondition. This execution semantics needs to be preserved during the formalization of transformation rules;

2. **target-metamodel respect**: given source-model elements respecting the source metamodels, each transformation rule must produce target-model elements respecting constraints described in target metamodels. In Fig. 1, a is an A-instance, c1 and c2 are two C-instances associated with a. Assuming that by applying R_1 on the source model SM={a,c1,c2}, we obtain the target model TM={b,d1,d2}, where b is a B-instance, d1 and d2 are D-instances associated with b.

   It is easy to check that TM respects all constraints specified in the current TMM; however, if we replace the multiplicity at the D-end with 0..1, TM would violate TMM.

3. **transformation invariant preservation**: we also need to check the correctness of transformation rules w.r.t. transformation invariants. In Fig. 1, we include following transformation invariants:

   - INV_1 specifies that the number of B-instances produced by the transformation is the same as the number of A-instances to which the transformation has been applied;
   - INV_2 specifies that the number of D-instances generated by the transformation is the same as the number of C-instances to which the transformation has been applied;
   - INV_3 specifies that a D-instance and its corresponding C-instance have the same value in the f attributes.

   Normally, such constraints will be encoded in the transformation rules R_1 and R_2. However, before that, they need to be stated as transformation objectives and then used to validate the transformation. Such transformation invariants, therefore, need to be explicitly represented in the abstract machines where there are abstract operations derived from transformation rules.

4.2 Formalizing transformation rules and invariants in B

For the first and second requirements, we need to create the B variables formalizing the source-model and target-model elements to which the transformation rules are applied. The B abstract operation for a given transformation rule therefore specifies the creation of target-model elements and the update of source-model elements taken by the transformation.
Fig. 4 shows ***Top***, an abstract machine which contains $a2b$, the abstract operation formalizing the rule R_1. We created the variables *sa*, *sc*, *sac* and *scf* to formalize respectively the effective instances of the classes A, C, the association A-C, and the attribute C::f which are manipulated by R_1. These variables are therefore considered respectively as subsets of the variables *ia*, *ic*, *iac*, and *icf* defined in ***Input*** (cf. Fig. 3). In addition, they will have constraints derived from the SMM like *ia*, *ic*, *iac*, and *icf*, but with some *appropriate weakenings* when this is necessary. In the example, concerning the variable *sac* which formalizes the association A-C, two *weakenings* were made: (i) the constraint on its domain was removed; (ii) the constraint on the range was also weakened. Such modifications are necessary to reflect that

```
MODEL Top
SEES MM
INCLUDES Input
VARIABLES sa, sc, sac, scf, tb, td, tbd, tdf, trab, trcd
INVARIANT
     sa ⊆ ia ∧ sc ⊆ ic ∧ scf ∈ sc → NAT ∧ ran(scf) ⊆
     10..100 ∧ scf ⊆ icf ∧ sac ∈ sa ↔ sc ∧ sac⁻¹ ∈ sc →
     sa ∧ sac ⊆ iac ∧ sac⁻¹ ⊆ iac⁻¹ ∧ ∀xx.(xx ∈ sa ⇒
     card(sac[{xx}]) ≤ 2) ∧ tb ⊆ TB ∧ td ⊆ TD ∧ tdf ∈ td →
     NAT ∧ ran(tdf) ⊆ 10..100 ∧ tbd ∈ tb ↔ td ∧ tbd⁻¹ ∈ td →
     tb ∧ ∀xx.(xx ∈ tb ⇒ card(tbd[{xx}]) ≤ 2) ∧ trab ∈ sa ↠
     ↣ tb ∧ trcd ∈ sc ⤖ td ∧ ∀cc.(cc ∈ dom(trcd) ⇒ scf(cc) =
     tdf(trcd(cc)))
INITIALISATION
     sa, sc, sac, scf, tb, td, tbd, tdf, trab, trcd :=
                        ϕ, ϕ, ϕ, ϕ, ϕ, ϕ, ϕ, ϕ, ϕ, ϕ
OPERATIONS
   rb ← a2b (aa) =
   pre aa ∈ ia−sa then
      any bb, sd, strcd where
         bb ∈ TB−tb ∧ sd ⊆ TD−td ∧ sd ≠ ϕ ∧ card(sd) =
         card(iac[{aa}]) ∧ strcd ∈ iac[{aa}] ⤖ sd
      then
         sa := sa∪{aa} ||
         sc := sc∪iac[{aa}] ||
         sac := sac ∪ ({aa}◁iac) ||
         scf := scf∪((iac[{aa}])◁icf) ||
         tb := tb∪{bb} ||
         td := td∪sd ||
         tbd := tbd∪{bb}×sd ||
         tdf := tdf∪{dd, ff|dd ∈ sd∧ff = icf(strcd⁻¹(dd))} ||
         rb := bb
      end
   end
END
```

Figure 4. Abstract Machine *Top*

sometimes during the transformation, while an A-instance is visited, its associated C-instances are not.

In the same manner, we created the variables *tb*, *td*, *tbd* and *tdf* to formalize respectively the effective instances of the classes B, D, the association B-D, and the attribute D::f which are produced by the execution of the rule R_1.

The solution for the third requirement is to create variables which store the *traceability links* of the transformation. On such variables, we specify the expected transformation invariants. In our example, we create two variables: (i) *trab* models the traceability link between A-instances and their corresponding B-instances created by R_1; (ii) *trcd* models the traceability link between C-instances and their corresponding D-instances created by R_2. We define (cf. the clause **INVARIANT** in ***Top***) *trab* as a bijection from *sa* to *tb*; and similarly, *trcd* is defined as a bijection from *sc* to *td*. By this way, two invariants INV_1 and INV_2 are automatically appeared. The predicate $\forall cc.(cc \in \mathbf{dom}(trcd) \Rightarrow scf(cc) = tdf(trcd(cc)))$, which comes after bijections, represents INV_3.

Each above-defined variable is initialized with an empty set. This initialization simulates the fact that before the transformation, no source-models element has been visited and no target-model element has been created. The fact that ***Top*** has a **SEES** link to ***MM*** allows this abstract machine to reference the sets and constants defined in the abstract machine ***MM***. The clause **INCLUDES** in ***Top*** allows ***Top*** to reference all variables defined in ***Input*** in the invariants predicates and in the operation $a2b$ in ***Top***.

Each transformation rule is formalized by a B abstract operation in the following manners: (i) the input elements are formalized by input parameters of the B abstract operation; (ii) the output elements are formalized by output parameters of the B abstract operation; (iii) the precondition is also formalized in the precondition part of the B abstract operation; and (iv) the production of target-model elements is formalized by the B operation body.

Considering the abstract operation $a2b$ in Fig. 4, this operation formalizes the rule R_1. The input parameter *aa* formalizes the A-instance which is the input element of R_1. The output parameter rb formalizes the output element of R_1. The precondition predicate $aa \in ia - sa$ simulates the fact that R_1 will take, each time it is called, a new A-instance in the source models; by this way, our formalization satisfies the first requirement described above. Since R_1 does not have precondition, no extra precondition predicate has been needed in *a2b*.

The body of $a2b$ is a substitution **any**, which declares three local variables: (i) *bb* formalizes the B-instance that corresponds to the A-instance formalized by *aa*; (ii) *sd* formalizes the D-instances that correspond to the C-instances associated with the A-instance of *aa*; (iii) *strcd* represents the traceability link between aforementioned C-instances and D-instances.

The substitution **any** contains assignment substitutions which are considered to be executed in parallel (i.e., their order is not important). Indeed, at the abstract machine level, what we need is to specify the operation effect. The sequential substitutions are not necessary and they will be introduced in refinement or implementation operations. The substitutions enclosed in the substitution **any**: (i) update the list of source-model elements visited by R_1, (ii) create target-model elements as specified by R_1, and (iii) update traceability links.

In our opinion, the substitution **any** is particularly suitable in the formalization of transformation rules since it allows us to simulate the creation of target-model elements.

We also need to consider the execution context of a transformation rule in its formalization. Fig. 5 shows $c2d$, the abstract operation formalizing the rule R_2.

The signature of $c2d$ has three input parameters: (i) *cc* derived from the input C-instance of R_2; (ii) *aa* formalizes the A-instance that the C-instance *cc* is associated with; (iii) *bb* formalizes the B-instance created from A-instance by R_1. The reason is that R_2 will be executed during the execution of R_1, so we need to consider such an execution context in the formalization of R_2. The execution context of a given transformation rule gives rise not only to input parameters

```
rd ← c2d (cc, aa, bb) =
pre
  $cc \in ic - sc \land aa \in sa \land aa \mapsto cc \in iac \land \mathbf{card}(sac[\{aa\}]) \leq$
  $1 \land bb \in tb \land aa \mapsto bb \in trab$
then
   any dd where $dd \in TD - td$ then
      $sc := sc \cup \{cc\}$ ||
      $scf := scf \cup \{cc \mapsto icf(cc)\}$ ||
      $sac := sac \cup \{aa \mapsto cc\}$ ||
      $td := td \cup \{dd\}$ ||
      $tbd := tbd \cup \{bb \mapsto dd\}$ ||
      $tdf := tdf \cup \{dd \mapsto scf(cc)\}$ ||
      $trcd := trcd \cup \{cc \mapsto dd\}$ ||
      $rd := dd$
   end
end
```

Figure 5. Abstract Operation $c2d$

but also to additional precondition predicates in the derived B operation. In Fig. 5, the precondition predicates related to *aa* have the following semantics: (i) the predicate $aa \in sa$ indicates that the A-instance formalized by *aa* is visited; (ii) the predicate $aa \mapsto cc \in iac$ indicates that the C-instance of *cc* is associated with the A-instance of *aa*; (iii) the predicate $\mathbf{card}(sac[\{aa\}]) \leq 1$ indicates that, for the A-instance of *aa*, there are still associated C-instances which have not been visited. For *bb*, the predicates $bb \in tb$, and the predicate $aa \mapsto bb \in trab$ indicate that the B-instance of *bb* was created and is linked to the *aa* A-instance by a traceability link.

The body of $c2d$ is also a substitution **any** which: (i) simulates the creation of the D-instance corresponding to the input C-instance; (ii) updates the list of visited C-instances, the list of created D-instances and the traceability link between C- and D-instances.

4.3 Proving transformation rules

Once we got B formalizations for metamodels, source models, transformation rules, and transformation invariants, the next step is to use B tools to generate *goals* and prove them. These goals are to validate the B abstract operations derived from the transformation rules w.r.t. the invariant predicates derived from the metamodels and the transformation invariants. If all proofs are successful, we conclude that the transformation rules are consistent w.r.t. source metamodels, target metamodels, and transformation invariants. If some goals are un-proved, analyzing them with B tools is a convenient and rigorous way to detect anomalies in metamodels, in transformation rules, and in transformation invariants.

We used **B4free** integrated with **CnP**[2] as the prover for this purpose. For the abstract machines ***Input*** and ***MM***, we have only obvious goals; nothing needs to be interactively proved. For the abstract operation $c2d$, there were 25 goals to be proved; only one of those goals needs to be interactively proved; the others were automatically proved. For the abstract operation $a2b$, there were 31 goals to be proved; 9 of those goals need to be interactively proved. With constraints described in Fig. 1, all non-automatically-proved goals were successfully proved in the interactive mode. Thus, R_1 and R_2 are consistent w.r.t. the metamodels SMM, TMM, and the transformation invariants INV_{1-3}.

We tried to modify properties in metamodels SMM and TMM, as well as in transformation invariants INV1−3 and in the semantics of transformation rules R_{1-2} and we observed that some of generated goals are no longer proved:

1. when we replace the multiplicity at the C-end in the A-C-association with 2..3, the invariant predicate specifying the cardinality of *iac* in ***Input*** will be replaced with $\forall xx.(xx \in ia \Rightarrow \mathbf{card}(iac[\{xx\}] \in 2..3))$. Without any modifications in semantics of R_1, we realize that the following goal generated for $a2b$ is no longer proved.

   $$\mathbf{card}((tbd \cup \{bb\} \times sd)[\{xx\}]) \leq 2 \qquad \text{(g1)}$$

   Goal g1 can not be proved when *xx* takes the value *bb* and the left-hand side becomes $\mathbf{card}(sd)$ which is equal to $\mathbf{card}(iac[\{aa\}])$. The latter one belongs to 2..3 according to invariant predicates related to *iac*. Thus, R_1 is no longer consistent with the modified SMM. So if we modify SMM, we need also make eventual modifications in the transformation rules R_1, R_2 or in the target metamodels TMM.

2. when we replace the constraint on the attribute D::f with $11 \leq$ D::F ≤ 101, the invariant predicates specifying the range of *tdf* in the abstract machines for $a2b$ and $c2d$ will be modified accordingly to become $\mathbf{ran}(tdf) \subseteq 11..101$. Without modifications in SMM, in R_1, and in R_2, we realize that one goal for $a2b$ and one goal for $c2d$ are no longer proved. Thus, the modification in TMM also needs a modification in R1, a modification in R2, and even modifications in SMM to keep all of them to be consistent with TMM.

3. when we modify the transformation invariant INV_3 with D::f < C::f for any C-instance and its corresponding D-instance, the corresponding invariant predicate for INV_3 will be replaced with $\forall cc.(cc \in \mathbf{dom}(trcd) \Rightarrow scf(cc) > tdf(trcd(cc)))$, which is no longer validated by $a2b$ and $c2d$ for the un-changed rules R_1 and R_2. Thus, the modification in INV_3 implies modifications in R_1 and R_2 and eventually in TMM and SMM.

4. when we modify the semantics of the R_2 with D::f:=C::f + 1, the substitution updating *tdf* in the abstract op-

[2]http://www.loria.fr/∼cansell/cnp.html

eration $c2d$ will be replaced with $tdf := tdf \cup \{dd \mapsto icf(cc)+1\}$. Without modifications in SMM, in TMM, and in transformation invariants, 2 goals related to $c2d$ are no longer proved. These goals validate the range of *tdf* and the invariant predicate modeling INV_3. Thus, the modified R_2 violates TMM and INV_3; we have to abort the modification in R_2 or change eventually TMM and INV_3 in order to keep the consistency of R_2 w.r.t. TMM and INV_3.

In the last case, we even noticed that if we do not change the semantics of the rule R_1, there is intuitively a mismatch between the un-changed R_1 and the modified R_2. But this mismatch can not be detected by doing and analyzing proofs on $a2b$ and $c2d$. Sect. 5 will deal with such an inconsistency.

5 Consistency among transformation rules

According to the semantics of the rules R_1 and R_2 (cf. Sect. 4), each execution of R_1 triggers two executions of R_2. Formalizing such a dependency provides a way to check the consistency among transformation rules.

5.1 Dependency of transformation rules

From a programming point-of-view, the fact that R_1 triggers R_2 can be formalized by calling the abstract operation $c2d$ within $a2b$. Using B framework, we need to place $a2b$ and $c2d$ in two separate abstract machines. The reason is that operations in the same abstract machine can not call each other. We create therefore two B abstract machines: ***Top*** for $a2b$ as shown in Fig. 4 and another machine named ***Bottom*** for $c2d$. The variables of ***Top*** and ***Bottom*** are identical as they are all derived from the source metamodel SMM, the target metamodel TMM, and transformation invariants INV_{1-3}. Each of the two abstract machines ***Top*** and ***Bottom*** has an **INCLUDES** link to ***Input***, in order to make use of the variables defined in ***Input*** in the invariant predicates and in the body of the abstract operations $a2b$ and $c2d$. We create ***Top_i***, an implementation module, which refines ***Top***. ***Top_i*** defines an implementation for the abstract operation $a2b$. It is in the implementation operation $a2b$ that the calls to the abstract operation $c2d$ happen.
As shown in Fig. 6, the clause **IMPLEMENTATION** declares the name of the implementation module as ***Top_i***. The clause **REFINES** declares that ***Top_i*** refines ***Top***. The clause **SEES** declares that ***Top_i*** has a **SEES** link to ***MM*** in order to use the constants defined in this abstract machine. The clause **IMPORTS** declares that ***Top_i*** imports ***Bottom*** so that the variables defined in ***Bottom*** become variables of ***Top_i*** and it is allowed to call operations defined in ***Bottom*** in the implementation operation $a2b$.

```
IMPLEMENTATION Top_i
REFINES Top
SEES MM
IMPORTS Bottom
OPERATIONS
    rb ← a2b(aa) =
    var bb, s_c, ii, dd in
        bb ← getB(aa);
        s_c ← getSC(aa);
        ii := 0;
        while s_c ≠ [ ] do
            dd ← c2d(first(s_c), aa, bb);
            s_c := tail(s_c);
            ii := ii+1;
        invariant
            ii ∈ NAT ∧ 0 ≤ ii ∧ ii ≤ 2 ∧ s_c ∈
            iseq(iac[{aa}]) ∧ ii+size(sc) = 2 ∧ iac[{aa}] =
            sac[{aa}]∪ran(s_c) ∧ sac[{aa}]∩ran(s_c) = ϕ ∧ sac ∈
            sa ↔ sc ∧ sac⁻¹ ∈ sc → sa ∧ card(sac[{aa}]) =
            ii ∧ sac = sac$0∪({aa}◁sac) sac⁻¹ =
            sac$0⁻¹∪(sac⁻¹▷{aa}) ∧ ...
        variant size(s_c)
        rb := bb
        end
    end
END
```

Figure 6. Implementation Module *Top_i*

Considering the implementation operation $a2b$, its signature is identical to those defined in the abstract operation $a2b$. Since we need temporary variables during the realization of the implementation operation, we use the substitution **var**. The temporary variables have the following semantics:

- *bb* formalizes the B-instance created by the rule R_1 from the A-instance formalized by *aa*;
- *s_c* formalizes the set of C-instances associated with the A-instance formalized by *aa*;
- *ii* is a counter variable specifying the number of iterations which have been done;
- *dd* formalizes the D-instance created by the R_2 for a C-instance formalized by an element in *s_c*.

In B theory, these variables need to be initialized by an expression or by the return value of an operation call. Thus, ***Bottom*** defines not only the abstract operation $c2d$ but also the auxiliary operations $getB$ and $getSC$ according to the aforementioned semantics of *bb* and *s_c*. The description of these auxiliary operations has been shorten for space reasons.
The substitution **var** contains a substitution **while** which specifies that: for each element in *s_c*, we make a call to $c2d$ in order to produce an element *dd*. By this way, we formalize actually the triggering of R_2 during the execution of R_1. The clause **invariant** in the substitution **while** contains predicates ensuring the correctness of the statements defined in the **do** part of the substitution **while**. In our example, the correctness consists of validating that the successive

calls to $c2d$ in the implementation operation $a2b$ produce the same effects as specified in the abstract operation $a2b$. The invariant predicates in the substitution **while** play an important role in validating the implementation operation $a2b$ w.r.t. the abstract operation $a2b$ (cf. Sect. 5.2). Therefore, they need to be carefully designed. In the example, we need to specify properties not only on local variables *ii* and *s_c* but also on variables *sa*, *sc*, *tb*, *td*, *sac*, *scf*, *tbd*, *tdf*, *trab*, and *trcd* which are modified during the **while** loop. In Fig. 6, the properties of several aforementioned variables are presented:

- *ii* is defined as a non negative integer to specify the number of iterations which were done;
- *s_c* is defined as an injective sequence of $iac[\{aa\}]$ to: (i) indicate that its elements models C-instances associated with the A-instance of aa; (ii) indicate that its elements are distinct; and (iii) facilitate the access to its elements using predefined functions **first** and **tail**.

  In each iteration, the value of *ii* increases and the number of elements in *s_c* decreases but the sum $ii+\mathbf{size}(sc)$ is always equal to 2 which is the number of C-instances associated with the A-instance formalized by aa.

  The predicates $iac[\{aa\}] = sac[\{aa\}] \cup \mathbf{ran}(s\_c)$ and $sac[\{aa\}] \cap \mathbf{ran}(s\_c) = \phi$ indicate that the new elements were added in $sac[aa]$ are elements of $iac[\{aa\}]$ which were removed from *s_c*. Thus, at the end of the loop **while**, we ensure that $sac[\{aa\}] = iac[\{aa\}]$, which is necessary to prove goals validating the substitution $sac := sac \cup (\{aa\} \lhd iac)$ in the abstract operation $a2b$.
- *sac* is defined in ***Bottom*** and affected by the calls of $c2d$ within the loop **while**. The predicates $sac \in sa \leftrightarrow sc$ and $sac^{-1} \in sc \rightarrow sa$ confirm the type of *sac* defined in ***Bottom***. The predicate $\mathbf{card}(sac[\{aa\}]) = ii$ indicates that for each iteration, one C-instance is visited. The predicates $sac = sac\$0 \cup (\{aa\} \lhd sac)$ and $sac^{-1} = sac\$0^{-1} \cup (sac^{-1} \rhd \{aa\})$ relate the current value of *sac* to its value before the execution of $a2b$ (denoted by $sac\$0$). These predicates are particularly important in validating the substitution $sac := sac \cup (\{aa\} \lhd iac)$ in the abstract operation $a2b$.

The clause **variant size**(s_c) in the substitution **while** is used to ensure that the **while** loop will terminate since we remove one element from *s_c* for each iteration.
There are two remarks in the use of the substitution **while** in formalizing the dependency among transformation rules. First, it is the only way in the B theory to specify the loop that happens in almost transformation rules. Second, due to the iterative nature of transformation rules and the fact that the substitution **while** is only allowed in B implementation modules [2], we decided to create ***Top_i*** as an implementation module instead of a refinement one.

5.2 Proving the consistency among transformation rules

When ***Top_i*** imports ***Bottom***, it imports all variables in ***Bottom***. As explained earlier, the variables defined in ***Top*** and ***Bottom*** are identical as they are systematically derived from metamodels and transformation invariants. Thus, ***Top_i***, without explicit invariant constraints, considers implicitly that variables imported from ***Bottom*** and variables defined in ***Top*** having the same name are equal. These implicit invariant predicates need to be validated by goals generated for the implementation operation $a2b$ in ***Top_i***.
Once the **while**-invariant predicates were correctly designed (i.e with necessary invariant predicates), analyzing goals generated for ***Top_i*** provides a convenient and rigorous way to check:

- if the execution context of $c2d$ is ensured for each call to it;
- if the execution of $c2d$ and auxiliary operations in the implementation operation $a2b$ preserves the semantics specified by the abstract operation $a2b$.

Thus, the consistency between R_1 and R_2 is checked, validated, and proved.
With the given semantics of R_1 and R_2, using **B4free** and **CnP**, we generate for ***Top_i*** 101 goals, in which, 16 goals need to be interactively proved and we needed one working day to prove them all. The rules R_1 and R_2 are therefore consistent with each other.
When we modify R_2 by affecting to D::f the value of C::f+1 instead of C::f, not only the constraints in TMM and transformation invariant INV_3 are violated (cf. Sect. 4.3) but the following goal generated for ***Top_i*** is no longer proved.

$$\begin{array}{l} tdf\$1 \\ \cup\{dd\$1, ff\$1 \mid dd\$1 \in \mathbf{ran}(trcd\$2 - trcd\$1) \ \wedge \\ \qquad\qquad ff\$1 = icf\$1(trcd\$2^{-1}(dd\$1))\} \\ \cup\{dd\$1 \mapsto icf\$1(\mathbf{first}(s\_c\$0))+1\} \\ = \\ tdf\$1\{dd\$0, ff\$0 \mid dd\$0 \in \mathbf{ran}((trcd\$2 \cup \{\mathbf{first}(s\_c\$0) \mapsto \\ dd\$1\}) - trcd\$1) \wedge ff\$0 = icf\$1((trcd\$2 \cup \{\mathbf{first}(s\_c\$0 \mapsto \\ dd\$1\})^{-1})(dd\$0))\} \end{array}$$

Figure 7. Modified R_2 Rule is no longer consistent with R_1

The left-hand side of goal in Fig. 7 corresponds to the change to *tdf* in the loop **while** each time $c2d$ is called, while the right-hand side corresponds to the effect specified in the abstract operation $a2b$ for tdf when **first**$(s\_c\$(0))$ is added. The left-hand and right-hand sides of goal in Fig. 7 are not equal due to the presence of the set $\{dd\$1 \mapsto icf\$1(\textbf{first}(s\_c\$0))+1\}$ in the left-hand side. Thus, the fact that goal 7 was un-proved corresponds to the mismatch between $a2b$ and $c2d$ and therefore also corresponds to the mismatch between the un-changed R_1 and the modified R_2.

6 Related work

There have been many works related to the formalization of metamodels and UML models using formal notations [15, 16, 5]. However, discussing such approaches here is out of the paper scope as we are interested in the verification and validation of model transformations.
Our approach can be considered as an answer for verification and validation aspects of the MDA from lessons identified by Brown and Conallen [1]: (i) any automation of model transformations is possible once the semantics of source and target models are well defined; (ii) the mapping rules must be semantically and technically checked; (iii) writing MT programs should be treated as a software development project itself.
Anastasakis et al. [12] used the Alloy language as the formal tool to formally analyze model transformations. The source metamodels, the target metamodels, and the transformation rules are formalized in Alloy. The derived Alloy model is then analyzed and simulated using Alloy Analyzer in order to detect flaws in the transformation rules. This approach is very similar to some aspects in our contributions, namely, formalizing and analyzing the correctness of transformation rules w.r.t. metamodels, however, we go much further by: (i) formalizing the dependency between transformation rules and analyzing the consistency of transformation rules w.r.t. each other; (ii) formalizing transformation invariants and analyzing the correctness of transformation rules w.r.t. transformation invariants. In addition, using B as the formal technique in analyzing model transformations allows us to deal with more complex model transformations thanks to the powerful B provers. You may argue that Alloy is fully automated while B is not. It is true but, Alloy handle smaller state spaces than B provers. In addition, we can improve the prove automation with B provers by providing appropriate mathematic rules.
Izerrouken et al. [17] experimented with the Coq proof-assistant in developing certified automated code generators (ACG). The source language of their ACG is MatLab/Simulink and the target language is a C language specialized for real-time systems. Both are then formalized in Coq. All needed properties of generated code are also expressed using Coq, then Coq Toolkit is used to verify the correctness of these properties on the source and target models. Such an approach can ensure the correctness of code generation at a posteriori but it does not consider explicitly the model transformations.
Moutou et al. [11] proposed a test-based approach to improve the reliability of model transformation programs which have three facets: specification and implementation, and test cases. The process starts with an initial test cases set, a specification, and an implementation. The improvement is realized in several rounds using mutation analysis; in each round, one improves successively test cases, the implementation and specification. This approach can help to improve an operational MT program. However, it can not be applied to the development of a new MT program from requirements. It may elaborate a given model transformation specification but it can not certify such a specification like ours. In addition, the fact that three facets (specification, implementation, and test cases) are getting evolved simultaneously, may make the result of this approach divergent.
Another approach for verifying model transformations comes from graph transformation community. Karsai and Narayan [8] advocated that it is possible to use of bisimulation techniques to verify the preservation of behavior semantics during a model transformation. Their idea is to capture errors in the target models, irrespective of faults in the model transformer. It does not solve the verification problem in general but answers the question for a specific target model, and for a selected set of safety properties. In comparison with this approach, our proposition to formalize transformation invariants may provide a solution to ensure the behavior preservation in a generalized manner. This approach can also be considered as being complementary to ours since it addressed the operational semantics of source and target models while our approach considered only the axiomatic semantics.

7 Conclusions

This paper presented an approach to prove model transformations using B. The B language based on set theory and first order logic, is suitable to formalize metamodels, invariants and models getting involved in model transformations. The operational semantics of the B language based on the generalized substitutions is also suitable to formalize transformation rules and their dependency. Using B provers on the derived B specifications provides a rigorous and convenient way to prove the correctness of model transformations. If all proofs are successful, we conclude that the model transformations are correct; otherwise, when some proofs fail, analyzing such proofs with B provers is also a rigorous and convenient way to detect the anomalies inside the metamodels, transformation rules, or transformation invariants.
Our approach is still generic but already very well structred

and can be used with any formal techniques. We chose B because we have worked with it for a long time in an industrial context. Using a simple example, we have shown the feasibility and different steps of our approach to prove model transformations. This paper is the first step in the definition of a verification approach capable to be scaled in an industrial context of the Model Driven Enginneering. Our intention is to prove the correctness of the code generators used in the development of real-time embedded software. For this purpose, we are going to investigate in the following directions :

1. The previous works on systematic formalisation in B of UML models [15, 16] provide us a solid background to automate the formalisation of metamodels and input models. However, we need to investigate further in formalising transformation rules so that such a formalisation can be automated. For this purpose, we are going to studying the formalisation of transformation rule in the ATL language, an OCL-like transformation language. Such a choice is motivated by our previous work on the formalisation of OCL constraints in B [14].

2. By definition, the dependency between transformation rules show the way the rules call each other during their execution. Such a relationship can be automatically identified. It is used to build the architecture of the derived B model. Such kind of work is similar to our previous work [13] which define a translation in B of UML class and collaboration diagrams into B.

3. We introduced *Transformation Invariant*, which represents properties on traceability links between the source-model elements and the target-model elements. In our future work, we will investigate the way *Transformation Invariant* is used to describe properties of source models which have to be preserved in target models.

References

[1] A. Brown, J. Conallen. An Introduction to Model-Driven Architecture (MDA) : Lessons from the Design and Use of an MDA Toolkit, April 2005. IBM Technical Library.

[2] J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.

[3] P. Behm, P. Desforges, and J.-M. Meynadier. MÉTÉOR: An Industrial Success in Formal Development, April 1998. An invited talk at the 2nd Int. B conference, LNCS 1939.

[4] A. Brown. An Introduction to Model-Driven Architecture (MDA) : MDA and Today's Systems, February 2004. IBM Technical Library.

[5] J.M. Bruel, J. Lilius, A. Moreira, and R.B. France. Defining Precise Semantics for UML. In *Object-Oriented Technology*, LNCS 1964, pages 113–122, Sophia Antipolis and Cannes (F), June 12-16, 2000. ECOOP 2000 Workshop Reader.

[6] E.W.D. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, 1976.

[7] E.Poupart et al. Offline Interoperability, Cost Reduction and Safer Operational Procedures using Meta-Modeling Technology. In *SPACEOPS2008*, Heidelberg, Germany, May 12-16 2008.

[8] G. Karsai, A. Narayanan. Towards Verification of Model Transformations Via Goal-Directed Certification. In *Model-Driven Development of Reliable Automotive Services*, pages 67–83. 2008.

[9] D. Gries. *The Science of Programming*. Springer Verlag, New York (USA), 1981. 350 pages.

[10] ATLAS group. *ATL User Manual*. INRIA Nantes, February 2006.

[11] J.-M. Moutou, B. Baudry, Y. Le Traon. Reusable MDA Components: A Testing-for-Trust Approach. In *MoDELS'06*, pages 83–97, Genova, Italy, October 2006.

[12] K. Anastasakis, B. Bordbar, J.-M. Kuster. Analysis of Model Transformations via Alloy. In *MoDeVVa07 : Model-Driven Engineering, Verification and Validation*, Nashville, USA, October 2 2007.

[13] H. Ledang and J. Souquières. Modeling Class Operations in B: Application to UML Behavioural Diagrams. In *ASE2001: the 16th IEEE International Conference on Automated Software Engineering*, pages 289–296, Loews Coronado Bay, San Diego (USA), November 26-29, 2001. IEEE Computer Society.

[14] H. Ledang and J. Souquières. Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B. In *APSEC 2002: the 9th Asia Pacific Software Engineering Conference*, Gold Coast, Queensland (AU), December 4-6, 2002. IEEE Computer Society.

[15] William E. McUmber and Betty H.C. Cheng. A General Framework for Formalizing UML with Formal Languages. In *ICSE01 : 23rd International Conference on Software Engineering*, pages 433–442, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[16] E. Meyer and J. Souquières. A systematic Approach to Transform OMT Diagrams to a B Specification. In *FM'99 : World Congress on Formal Methods in the Development of Computing Systems*, LNCS 1708, Toulouse (F), September 1999. Springer-Verlag.

[17] N. Izerrouken, X. Thirioux, M. Pantel, M. Strecker. Certifying an Automated Code Generator Using Formal Tools : Premilinary experiments in the GeneAuto project. In SIA, editor, *ERTS 2008 : 4th European Congress on Embedded Real Time Software*, January 29-31, February 1 2008.

[18] F. Terrier and S. Gérard. MDE Benefits for Distributed, Real Time and Embedded Systems. In *DIPES 2006 : IFIP TC 10 Working Conference on Distributed and Parallel Embedded Systems*, pages 15–24, 2006.