Automated Verification of Model Transformations in the Automotive Industry*

Gehan M. K. Selim[1], Fabian Büttner[2], James R. Cordy[1], Juergen Dingel[1], and Shige Wang[3]

[1] School of Computing, Queen's University, Kingston, Ontario, Canada
[2] AtlanMod, École des Mines de Nantes - INRIA, LINA, Nantes, France
[3] Electrical and Controls Integration Lab, General Motors Research and Development, Warren, Michigan, USA

Abstract. Many companies have adopted MDD for developing their software systems. Several studies have reported on such industrial experiences by discussing the effects of MDD and the issues that still need to be addressed. However, only a few studies have discussed using automated verification of industrial model transformations. We previously demonstrated how transformations can be used to migrate GM legacy models to AUTOSAR models. In this study, we investigate using automated verification for such industrial transformations. We report on applying an automated verification approach to the GM-to-AUTOSAR transformation that is based on checking the satisfiability of a relational transformation representation, or a transformation model, with respect to well-formedness OCL constraints. An implementation of this approach is available as a prototype for the ATL language. We present the verification results of this transformation and discuss the practicality of using such tools on industrial size problems.

Keywords: Model Transformation, Automated Verification, Automotive Industry

1 Introduction

Model Driven Development (MDD) has been increasingly used in the last decade for software development and, in many cases, has replaced traditional, code-centric approaches. In MDD, *models* or software abstractions are the basic building blocks in the software development life cycle and *model transformations* are the technology used to map between models conforming to different metamodels. Transformations are used for different purposes in MDD, e.g., refactoring, migration, and code generation. Since transformations are essential in MDD, transformation testing and verification is essential to the success of MDD.

* This work was partially funded by the Nouvelles Équipes Program of the Pays de la Loire Region (France), and by NSERC (Canada), as part of the NECSIS Automotive Partnership with General Motors, IBM Canada and Malina Software Corp.

Several studies have reported on industrial experiences in adopting MDD [13, 25]. However, only a few of them have specifically discussed using model transformations in industry. Daghsen *et al.* [14] used transformations to map AUTOSAR timing models to classical scheduling models to perform timing analysis. Giese *et al.* [15] used triple graph grammars to synchronize SysML system engineering models with AUTOSAR software engineering models. Studies reporting on automated verification of industrial transformations have also been limited.

In this study, we report on using a light-weight, automated verification prototype to reason about the correctness of an ATL [22] transformation developed for the automotive industry [29]. More specifically, we check the correctness of the transformation with respect to OCL well-formedness constraints after translating the ATL transformation into a logical satisfiability problem. The basic approach has been presented in previous work [10] but to our knowledge we are the first reporting on its application to an industrial-sized verification problem.

While the transformation itself is not exceptionally large (in the number of transformation rules), the corresponding metamodels are. Together, they comprise 1586 classes, 897 associations, and 371 multiplicity constraints. Since even types not directly touched by the transformation are relevant for the verification (due to constraints that relate them), we have to deal with large potential instances. To verify our transformation, we have successfully checked models of up to 20000 potential elements with reasonable runtimes (although all counter examples found contained much fewer elements and were found quite quickly). Hence we claim that the verification approach is applicable to realistic verification scenarios.

The rest of this paper is organized as follows: Section 2 gives an overview of the GM-to-AUTOSAR transformation previously presented in [29]; Section 3 introduces the applied verification approach and prototype; Section 4 describes the case study conducted to verify the GM-to-AUTOSAR transformation using the aforementioned prototype; Section 5 summarizes the results of the case study and investigates the performance of the used approach; Section 6 discusses its strengths and limitations; Section 7 summarizes related work in the literature and Section 8 concludes and discusses future work.

2 Background: Model Transformation in the Automotive Industry

We now review the GM-to-AUTOSAR transformation presented in [29] which was used to migrate GM legacy models to the AUTOSAR standard.

2.1 Overview of the Model Transformation Problem

As one of the leading automotive companies, General Motors has been adopting MDD for the develoment of automotive software. GM engineers have been using a domain-specific metamodel for the development of vehicle control software (VCS). We refer to their domain-specific metamodel as the *GM metamodel*.

AUTOSAR (the AUTomotive Open System ARchitecture) [2] has been developed and adopted by many organizations as an automotive industry standard that is meant to facilitate the development and integration of software components from different vendors. AUTOSAR specifies requirements for software that is meant to conform to the standard. Further, AUTOSAR has its own metamodel with a well-defined architecture and interfaces.

Since the majority of organizations in the automotive industry are migrating to AUTOSAR, transforming models conforming to the GM metamodel to their equivalent AUTOSAR models is an important goal. Thus, we have previously developed and reported on a transformation that maps between subsets of the GM metamodel and the AUTOSAR metamodel as its source and target metamodels. In that work, we focused on subsets of the two metamodels that represent the deployment and interaction of software components.

2.2 The GM Metamodel

Fig. 1 illustrates the subset of the GM metamodel that we manipulated in our transformation in [29][1]. The *PhysicalNode* models a physical node on which soft-

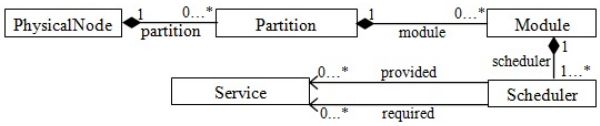


Fig. 1. Subset of the GM metamodel directly used by our transformation in [29].

ware is deployed. A *PhysicalNode* may contain multiple *Partition*s (i.e., processing units or memory partitions) on which software is deployed. Multiple *Module*s can be deployed on a single *Partition*. A *Module* is an atomic, deployable, and reusable element in a product line and can contain multiple *Scheduler*s. A *Scheduler* is the basic unit for software scheduling. It contains behavior-encapsulating entities, and is responsible for managing services provided or required by the behavior-encapsulating entities. Each *Scheduler* may provide and/or require *Service*s, which model the services provided or required by the *Scheduler*.

2.3 The AUTOSAR Metamodel

The AUTOSAR metamodel is defined as a set of templates. Each template specifies an AUTOSAR artifact such as software components. Among the defined templates, the *System* template [1] models the configuration of a system or an Electronic Component Unit (ECU). An ECU is a physical unit on which software is deployed. When used for modeling the configuration of an ECU, the System template is referred to as the *ECU Extract*. Fig. 2 shows the subset of the ECU Extract manipulated by our transformation. The ECU extract is modeled using the *System* type that aggregates *SoftwareComposition* and *SystemMapping*

[1] In this study, we follow the same obfuscated naming conventions that we used for the GM metamodel in [29] for reasons of confidentiality.

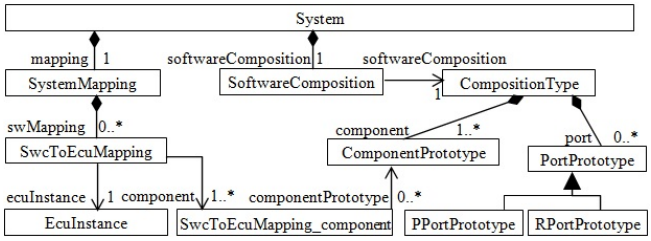


Fig. 2. Subset of the AUTOSAR System Template directly used by our transformation.

elements. The *SoftwareComposition* type points to the *CompositionType* type which eliminates any nested software components in a *SoftwareComposition*. The *SoftwareComposition* type models the architecture of the software components deployed on an ECU, the ports of these software components and the ports' connectors. Each Software component is modeled using the *ComponentPrototype* type, which defines the structure and attributes of a software component; each port is modeled using the *PortPrototype* type (i.e., a *PPortPrototype*or a *RPortPrototype*) for providing or requiring data and services.

The *SystemMapping* type binds the software components to ECUs and the data elements to signals and frames (not shown). The *SystemMapping* type aggregates the *SwcToEcuMapping* type, which assigns *SwcToEcuMapping_component*s to an *EcuInstance*. *SwcToEcuMapping_component*s in turn, refer to *ComponentPrototype* elements. According to AUTOSAR, only one *SwcToEcuMapping* should be created for each processing unit or memory partition in an ECU.

3 Verification Methodology

We apply the automated verification approach presented in [10] to the GM-to-AUTOSAR transformation. In short, we translate the ATL transformation T, its source metamodel MM_{src}, and its target metamodel MM_{tar} into a combined model, or a *transformation model*, consisting of MM_{src} and MM_{tar} and additional model elements that represent the transformation rules. Additionally, a set *Sem* of OCL constraints is generated for the combined model that characterizes the execution semantics of the ATL rules. For declarative ATL rules without recursion, the constraints describe the ATL semantics one-to-one, i.e., each valid instance of the transformation model corresponds to an execution of the transformation and vice versa.

Using this representation we can check partial correctness of the transformation with respect to properties specified as OCL constraints over the source and/or the target model, by checking if there exists a counterexample within a specific scope (i.e., maximum number of objects per class). More specifically, for a set of transformation preconditions (or assumptions) $Pre_1, \ldots, Pre_n$ and a set of postconditions (or assertions) $Post_1, \ldots, Post_m$, we want to show that for

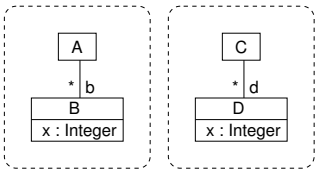
each instance M of the transformation model,

$$\begin{aligned}&\big(Sem_1 \text{ and } Sem_2 \text{ and } \ldots \text{ and } Sem_k \text{ and}\\ &\ Pre_1 \text{ and } Pre_2 \text{ and } \ldots \text{ and } Pre_n\big) \text{ implies}\\ &\big(Post_1 \text{ and } Post_2 \text{ and } \ldots \text{ and } Post_m\big)\end{aligned} \tag{1}$$

holds. This can be expressed equivalently as follows: For each postcondition $Post_i$ $(1 \leq i \leq m)$, the following formula must be unsatisfiable (i.e., there is no model M under which the formula is true):

$$Sem_1 \text{ and } \ldots \text{ and } Sem_k \text{ and } Pre_1 \text{ and } \ldots \text{ and } Pre_n \text{ and not}(Post_i) \tag{2}$$

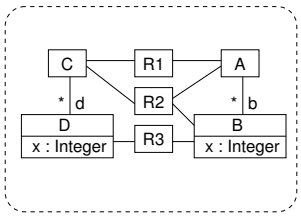
Fig. 3 illustrates this using a simple example. In the upper part we have an ATL transformation (c) over the shown source and target metamodels (a) and (b). The transformation copies the A-B structure to the C-D structure, but creates an additional D object when copying an 'empty' A object. The middle part shows the transformation model of this transformation. In the class diagram (d), each of the three rules is translated into a trace class and connected to the source and target classes according to the *from* and *to* patterns of the rule. The OCL constraints (e) capture the execution semantics of the transformation such as the matching of rule R1, the binding of primitive and object-typed properties, and the controlled creation of target objects. Some pre-/post- conditions are shown in (f) and (g), respectively.



(a) Source MM (b) Target MM

```
create OUT : Tar from IN : Src
rule R1 { from a : Src!A ( a.b->notEmpty() )
          to   c : Tar!C ( d <- a.b ) }
rule R2 { from a : Src!A ( a.b->isEmpty() )
          to   c : Tar!C ( d <- Set{d1} ),
               d : TargetMM!D ( x <- 0 ) }
rule R3 { from b : Src!B  to d : Tar!D ( x <- b.x ) }
```

(c) ATL transformation



(d) Transformation model

```
context a : A inv Sem_R1_match: a.b->notEmpty() implies
  R1. allInstances ()->one(r1|r1.a = a)
context R1 inv Sem_R1_cond: self.a.b->notEmpty()
context R1 inv Sem_R1_bind_c:
   self.d->forAll(d |  self.a.b->exists(b | b.r2 = d) and
   self.a.b->forAll(b |  self.d->exists(b | b.r2 = d)
context R3 inv Sem_R3_bind_d: self.d.x = self.b.x
context C inv Sem_C_create:
   self.r1->size() + self.r2->size() = 1
```

(e) OCL constraints for ATL semantics (excerpt)

```
context A inv Pre1:  self.b.x->sum() >= 0
context A inv Pre2:  self.b->size() >= 1
```

(f) Preconditions

```
context C inv Post1:  self.d->size() >= 1
context C inv Post2:  self.b.x->sum() >= 0
```

(g) Postconditions

Fig. 3. Transformation model example

To verify that, for example, postcondition $Post_i$ is implied by the transformation (given the preconditions), we have to check that Eq. (2) is unsatisfiable. This can be tested using metamodel satisfiability checkers, or *model finders*, such as the USE Validator [23] which is publicly available [35]. The USE Validator translates the UML model and the OCL constraints into a relational logic formula and employs the SAT-based solver Kodkod [33] to check the unsatisfiability of Eq. (2) for each of the post-conditions $Post_i$ within a given scope. Thus, we have four different representations of the problem space, (i) ATL + OCL, (ii) OCL, (iii) relational logic, and (iv) propositional logic (for the SAT solver).

We have implemented the whole chain as an verification prototype (Fig. 4). We have implemented the ATL-to-OCL transformation [10] as a higher-order ATL transformation [32], i.e., a transformation from Ecore and ATL metamodels to Ecore metamodels (where the Ecore model can contain OCL constraints as annotations). Our implementation automatically generates the *Sem* constraints from the ATL transformation as well as *Pre* and *Post* constraints from the structural constraints in the source and target metamodels (further constraints to be verified can be added manually). Since the USE validator has a proprietary metamodel syntax, we have created a converter from Ecore to generate a *USE specification*. We also generate a default search space configuration, which is a file specifying the scopes and ranges for the attribute values. In the search configuration, we can disable or negate individual invariants or constraints.

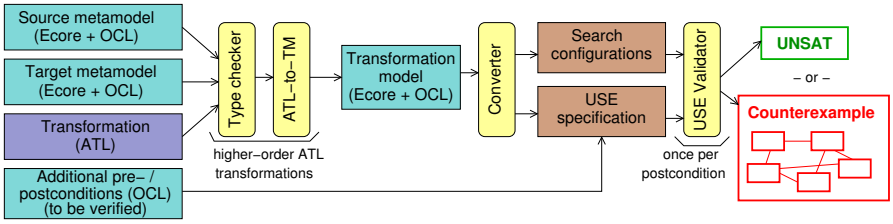


Fig. 4. The tool chain used to perform the transformation verification.

Steps to verify a postcondition using the prototype: To check Eq. (2) for a postcondition, we have to negate the respective postcondition and disable all other postconditions in the generated *search configuration* (Fig. 4) and then run USE. If USE reports 'unsat', this implies that there is no input model in the search space for which the transformation can produce an output model that violates the postcondition. If there exists a counterexample, USE provides the object diagram of the counterexample which can be analyzed using many browsing features of the tool. Although the implementation is a prototype, it is not specific to the GM-to-AUTOSAR transformation.

4 Case Study: Evaluating Transformations in the Automotive Industry Using Automated Verification

We use the prototype described in Section 3 to verify our GM-to-AUTOSAR transformation. However, the verification prototype can only verify ATL transformations composed of declarative matched rules and non-recursive lazy rules.

Thus we have changed the implementation described in [29] to be completely declarative and compatible with the format required by the prototype. The final reimplementation is intended to achieve the same mapping as the original implementation described in [29].

In this section, we describe the constructs used to re-implement our transformation and the different kinds of constraints formulated for verification.

4.1 Reimplementation of the GM-to-AUTOSAR Model Transformation

In the first implementation of the GM-to-AUTOSAR transformation, we used two ATL matched rules, 9 functional helpers and 6 attribute helpers to implement the required mapping between the two metamodels. After reimplementing the transformation to be completely declarative, the new transformation was composed of three matched rules and two lazy rules. Although we had to reimplement the transformation to use the verification prototype, we point out that the new declarative implementation is simpler and more readable. The rules implemented are listed in Table 1 together with the types of the rules, the input element matched by the rule, and the output elements generated by the rule.

Rule Type	**Rule Name**	**Input Types**	**Output Types**
Matched Rule	`createComponent`	*Module*	*SwCompToEcuMapping_component*, *ComponentPrototype*
Matched Rule	`initSysTemp`	*PhysicalNode*	*System*, *SystemMapping*, *SoftwareComposition*, *CompositionType*, *EcuInstance*
Matched Rule	`initSingleSwc2EcuMapping`	*Partition*	*SwcToEcuMapping*
Lazy Rule	`createPPort`	*Scheduler*	*PPortPrototype*
Lazy Rule	`createRPort`	*Scheduler*	*RPortPrototype*

Table 1. The types of ATL constructs used to reimplement the transformation, their designated names, and their input and output element types.

As described in [29], the relationships between the outputs of the matched rules are built using the ATL predefined function `resolveTemp`. The `resolveTemp` function allows a rule to reference the elements that are yet to be generated by another rule at runtime. For example, the `resolveTemp` function was used to connect the *SwcToEcuMapping* elements created by the `initSingleSwc2EcuMapping` matched rule to the *SystemMapping* element created by the `initSysTemp` matched rule. Further, the matched rule `initSysTemp` calls the two lazy rules and assigns the union of the lazy rules' outputs to the *ports* of the *CompositionType* produced by the `initSysTemp` rule.

4.2 Formulation of OCL Pre- and Postconditions

In general, the OCL postconditions in our approach can be either defined on elements of the target metamodel only (then we call them *target invariants*), or they can relate the elements of the source and target metamodels (then we call them *transformation contracts*). Usually, a transformation contract specifies an

implication 'when an input has a property then it's corresponding output has a property'. The OCL preconditions are propositions about the input that we assume to always hold.

In our case study, the preconditions were given by the multiplicity and composition constraints automatically extracted from the GM metamodel as OCL constraints. The formulated OCL postconditions are summarized in Table 2. We divide the formulated postconditions into four categories: *Multiplicity Invariants*, *Uniqueness Contracts*, *Security Invariants*, and *Pattern Contracts*. For each constraint in Table 2, we add to the beginning of its formulation an abbreviation (e.g., $(M1)$, $(U2)$) that will be used in the rest of the paper to refer to the constraint. The Multiplicity Invariants were automatically generated by the prototype. All the other postconditions were manually formulated.

Multiplicity Invariants:

- $(M1)$ **Context** CompositionType **inv** CompositionType_component: self.component→**size()** ≥ 1
- $(M2)$ **Context** SoftwareComposition **inv** SoftwareComposition_softwareComposition: self.softwareComposition $\neq$ **null**
- $(M3)$ **Context** SwcToEcuMapping **inv** SwcToEcuMapping_component: self.component→**size()** ≥ 1
- $(M4)$ **Context** SwcToEcuMapping **inv** SwcToEcuMapping_ecuInstance: self.ecuInstance $\neq$ **null**
- $(M5)$ **Context** System **inv** System_softwareComposition: self.softwareComposition $\neq$ **null**
- $(M6)$ **Context** System **inv** System_mapping: self.mapping $\neq$ **null**

Uniqueness Contracts: Let Unique (invName, X, Y) be
Context Global **inv** invName: (X.**allInstances()**→**forAll**(x1:X, x2:X| x1.Name=x2.Name **implies** x1=x2)) **implies** (Y.**allInstances()**→ **forAll**(y1:Y, y2:Y| y1.shortName = y2.shortName **implies** y1=y2))

- $(U1)$ UnqCompName= Unique (UNQCOMPNAME, Module, ComponentPrototype)
- $(U2)$ UnqSysMName= Unique (UNQSYSMNAME, PhysicalNode, SystemMapping)
- $(U3)$ UnqSysName= Unique (UNQSYSNAME, PhysicalNode, System)
- $(U4)$ UnqSwcmpsName= Unique (UNQSWCMPSNAME, PhysicalNode, SoftwareComposition)
- $(U5)$ UnqCmpstyName= Unique (UNQCMPSTYNAME, PhysicalNode, CompositionType)
- $(U6)$ UnqEcuiName= Unique (UNQECUINAME, PhysicalNode, EcuInstance)
- $(U7)$ UnqS2EName= Unique (UNQS2ENAME, Partition, SwcToEcuMapping)
- $(U8)$ UnqPpName= Unique (UNQPPNAME, Scheduler, PPortPrototype)
- $(U9)$ UnqRpName= Unique (UNQRPNAME, Scheduler, RPortPrototype)

Security Invariant:

- $(S1)$ **Context** System **inv** Self_Cont: mapping.swMapping→**forAll**(swc2ecumap: SwcToEcuMapping| swc2ecumap.component → **forAll**(mapcomp : SwCompToEcuMapping_component| mapcomp.componentPrototype→**forAll**(compro: ComponentPrototype| softwareComposition.softwareComposition.component→ **exists**(c: ComponentPrototype| c=comppro))))

Pattern Contracts:

- $(P1)$ **Context** Global **inv** Sig2P: PhysicalNode.**allInstances()**→ **forAll**(e1:PhysicalNode| e1.partition→ **forAll**(vd: Partition| vd.module→ **forAll**(di: Module| di.scheduler→ **forAll**(ef:Scheduler| (ef.provided→**notEmpty()**) **implies** (System.**allInstances()**→**one**(sy:System| (sy.shortName=e1.Name) **and** (sy.softwareComposition.softwareComposition.port→ **one**(pp:PortPrototype| (pp.shortName=ef.Name) **and** (pp.**oclIsTypeOf**(PPortPrototype))))))))))
- $(P2)$ **Context** Global **inv** Sig2R: PhysicalNode.**allInstances()**→ **forAll**(e1:PhysicalNode| e1.partition→ **forAll**(vd:Partition| vd.module→ **forAll**(di: Module| di.scheduler→ **forAll**(ef:Scheduler| (ef.required→**notEmpty()**) **implies** (System.**allInstances()**→ **one**(sy:System| (sy.shortName=e1.Name) **and** (sy.softwareComposition.softwareComposition.port→ **one**(rp:PortPrototype| (rp.shortName=ef.Name) **and** (rp.**oclIsTypeOf**(RPortPrototype))))))))))

Table 2. Formulated OCL Constraints

Multiplicity Invariants ensure that the transformation does not produce an output that violates the multiplicities in the AUTOSAR metamodel (Fig. 2). As described in Section 3, the prototype generates a USE specification with a multiplicity invariant for each multiplicity in the AUTOSAR metamodel. Ideally, we would check the satisfiability of all the multiplicity invariants generated for the AUTOSAR metamodel. Since our transformation manipulates a subset of the metamodels, we only check multiplicity invariants for output elements affected by our transformation. We have identified six of the generated multiplicity invariants that are affected by our transformation. ($M1$) ensures that each *CompositionType* is associated to more than one *ComponentPrototype* through the *component* association. ($M2$) ensures that each *SoftwareComposition* is associated with one *CompositionType* through the *softwareComposition* association. The rest of the multiplicity invariants can be interpreted in a similar way.

Uniqueness Contracts require the output element (of type Y) generated by a rule to be uniquely named (by the *shortName* attribute) within its respective scope if the corresponding input element (of type X) matched by the rule is uniquely named (by the *Name* attribute) within its scope too. For example, in Section 4.1, we discussed that the matched rule `createComponent` maps *Module*s to *ComponentPrototype*s. Thus, *U1* mandates that the *ComponentPrototype*s generated by the transformation are uniquely named, if the corresponding *Module*s are uniquely named too. The rest of the uniqueness contracts are similar and ensure uniqueness of the output elements of each rule described in Section 4.1 if their corresponding input elements are unique too.

The only security invariant defined, *S1*, mandates that within any *System* element, all its composite *SwcToEcuMapping*s must refer to *ComponentPrototype*s that are contained within the *CompositionType* lying under the same *System* element (refer to Fig. 1). Thus, this invariant assures that any ECU configuration (modeled by a *System* element) is self contained and does not refer to any *ComponentPrototype* that is not allocated in that ECU configuration.

Pattern contracts require that if a certain pattern of elements is found in the input model, then a corresponding pattern of elements must be found in the output model. Pattern contracts also mandate that corresponding elements in the input and output patterns must have the same name. *P1* mandates that if a *PhysicalNode* is connected to a *Service* through the *provided* association (in the input model), then the corresponding *System* element will eventually be connected to a *PPortPrototype*. *P1* also ensures that the names of the *PhysicalNode* and the *System* are equivalent and that the names of the *Scheduler* (containing the *Service*) and the *PPortPrototype* are equivalent. The contract *P2* is similar to *P1* but manipulates *required Service*s and *RPortPrototype*s instead.

Since invariants are constraints on target metamodel elements, the Multiplicity and Security invariants are specified within the *context* of their respective AUTOSAR elements. Since contracts are constraints on the relationships between the source and target metamodel elements, they do not relate to an AUTOSAR element per se. Thus, we add a class to the USE specification file, *Global*, which is used as the context of the Uniqueness and Pattern contracts.

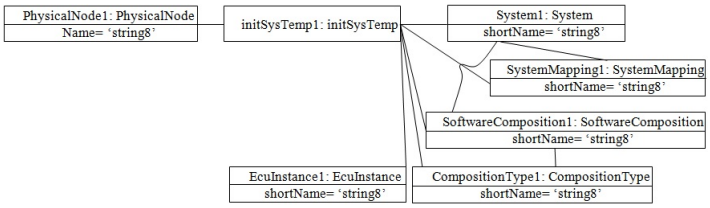


Fig. 5. Counterexample generated for the mult. inv. `CompositionType_component`.

5 Results

In this section, we discuss the results of verifying the OCL contraints defined in Section 4.2 using the verification prototype. We show how the verification prototype was able to uncover bugs in the GM-to-AUTOSAR transformation that were fixed and re-verified. We also describe the results of a study to determine the performance of the used verification approach.

5.1 Verifying the Formulated OCL Constraints

Using the verification prototype, we generated a USE specification and a search configuration as shown in Fig. 4. After adding the constraints (Table 2) to the USE specification, we ran the USE tool once for each constraint.

Out of the 18 constraints defined in Table 2, two multiplicity invariants were found to be violated by the transformation: `CompositionType_component` and `SwcToEcuMapping_component`. In other words, our transformation can generate a *CompositionType* with no *ComponentPrototype*s and/or a *SwcToEcuMapping* with no *ComponentPrototype*s. Both of these possible outputs violate the multiplicities defined in the AUTOSAR metamodel (Fig. 2). The counterexamples were found by USE even within a scope of just one object per concrete class.

Due to the page limit, we only show an excerpt of the counterexample generated for the invariant `CompositionType_component` in Fig. 5. The counterexample shows that the rule `initSysTemp` maps a *PhysicalNode* to five elements, one of which is *CompositionType*. Since the rule does not have any restrictions on the generated *CompositionType*, it was created without associating it to any *ComponentPrototype* through the *component* association. The counterexample for the `SwcToEcuMapping_component` invariant was similar showing that the `initSingleSwc2EcuMapping` rule creates a *SwcToEcuMapping* element without mandating that it is associated to any *SwCompToEcuMapping_component* element through the *component* association.

After examining the two counterexamples generated by USE for the two violated multiplicity invariants, we identified two bugs in two rules shown in Table 3: `initSysTemp` and `initSingleSwc2EcuMapping`. The bold, underlined text are the updates to the rules that fix the two bugs. `initSysTemp` initially mapped a *PhysicalNode* to many elements, including a *CompositionType* that must contain at least one *ComponentPrototype*. If the *PhysicalNode* did not have any *Module* in any of its *Partition*s, then the created *CompositionType* will not contain any *ComponentPrototype*s. Thus we added a matching constraint

```
rule initSysTemp{
from ph: GM!PhysicalNode (ph.partition→exists(p|p.Module→notEmpty()))
to
      ...
      compostype:autosar!CompositionType(
      ...
      component ←ph.partition→collect(p|p.Module)→flatten()→collect(m|
      thisModule.resolveTemp(m, 'comp'))) }
```

```
rule initSingleSwc2EcuMapping {
from p:GM!Partition((GM!PhysicalNode.allInstances()→one(ph|ph.partition→includes(p)))
      and(p.module→notEmpty()))
to
      mapping:autosar!SwcToEcuMapping (
      shortName ← p.Name,
      component ← p.Module→collect(m|thisModule.resolveTemp(m, 'mapComp')),
      ecuInstance ←thisModule.resolveTemp((GM!PhysicalNode.allInstances()→select(ph|
      ph.partition→includes(p)))→first(),'EcuInst'))}
```

Table 3. The two rules that required updates to address the two violations of multiplicity invariants.

to the *PhysicalNode* matched by the rule to ensure that any of its *Partition*s must contain at least one *Module*. Similarly, `initSingleSwc2EcuMapping` initially mapped a *Partition* to a *SwcToEcuMapping* that must contain at least one *SwCompToEcuMapping_component*. If the *Partition* did not have any *Module*, then the created *SwcToEcuMapping* will not contain any *SwCompToEcuMapping_component*. Thus we added a matching constraint to the *Partition* matched by the rule to ensure that it must contain at least one *Module*.

The 18 constraints were reverified on the updated transformation, and were all found to be satisfied.

5.2 Performance of the Verification Approach

To explore the performance of our approach, we used the verification prototype to verify the 18 constraints (Table 2) for different scopes. We ran the verification with scopes between one and 12. We only show the results for scopes 6, 8, 10, and 12 due to the page limit. The scope determines the maximum number of objects per concrete class in the search space. In our tests, we used the same scope for all classes, although it could be set individually. Since our transformation model has 1586 classes, a scope of n generates a model with $1586n$ potential elements (and their corresponding links and attribute values). All experiments where run on a standard laptop at 2.50 GHz and 16 GB of memory, using Java 7, Kodkod 2.0, and Glucose 2.1.

For each combination of constraint and scope, the prototype generates two time values: the time the prototype takes to translate the relational logic formula into a propositional formula (i.e., *translation time*) and the time the SAT solver takes to solve the formula (i.e., *constraint solving time*).

We show these two time values (in seconds) in Table 4. Each column represents the time intervals for each of the 18 constraints, where the *Constraint Abbreviation* is the abbreviation given to each constraint in Table 2 (e.g., (M1) and (U5)). Each row represents the time intervals for a different scope. Thus,

each cell within the table shows the translation time and the constraint solving time of a certain constraint at a specific scope.

		Constraint Abbreviation (from Table 2)								
		U1	**U2**	**U3**	**U4**	**U5**	**U6**	**U7**	**U8**	**U9**
Scope	**6**	76 \ 25	76 \ 19	76 \ 22	76 \ 7	77 \ 19	76 \ 24	76 \ 7	76 \ 7	74 \ 5
	8	169 \ 74	165 \ 79	168 \ 106	165 \ 37	168 \ 85	171 \ 68	167 \ 38	166 \ 57	169 \ 45
	10	279 \ 165	280 \ 188	279 \ 210	281 \ 114	277 \ 211	280 \ 207	281 \ 147	282 \ 170	279 \ 206
	12	455 \ 976	434 \ 643	431 \ 623	428 \ 322	426 \ 827	428 \ 616	425 \ 584	427 \ 604	430 \ 501

		Constraint Abbreviation (from Table 2)								
		M1	**M2**	**M3**	**M4**	**M5**	**M6**	**S1**	**P1**	**P2**
Scope	**6**	74 \ 2	73 \ 0.4	74 \ 1	74 \ 1	75 \ 0.5	74 \ 0.5	74 \ 40	242 \ 14	244 \ 7
	8	162 \ 2	162 \ 1	164 \ 2	163 \ 2	164 \ 1	166 \ 1	168 \ 429	1453 \ 37	1422 \ 65
	10	280 \ 12	281 \ 1	277 \ 6	281 \ 3	275 \ 1	274 \ 1	277 \ 3619	6225 \ 80	6178 \ 249
	12	426 \ 18	425 \ 1	421 \ 25	424 \ 4	422 \ 1	425 \ 1	*	21312 \ 710	21092 \ 814

Table 4. Translation\Constraint Solving times (seconds) for the 18 constraints on different scopes. For a scope of 12, the verification of S1 did not terminate in a week.

Two observations can be made from Table 4. First, despite the exponential complexity of checking boolean satisfiability, we could verify the postconditions for scopes up to 12 in most of the cases; only the analysis of S1 did not finish for scope 12; the constraint solving time of S1 in scope 10 was the longest (just over an hour). Although we have no proof that no bugs will appear for bigger scopes, we are confident that a scope of 12 was sufficient to uncover any bugs in our transformation with respect to the defined constraints. In fact, the two bugs that were uncovered and fixed were found at a scope of one.

Second, the translation times are larger than expected and grow mostly polynomially. This can be attributed to the approach used by Kodkod to unfold a first-order relational formula into a set of clauses in conjunctive normal form (CNF), given an upper bound for the relation extents [33]. While transforming a formula into CNF grows exponentially with the length of the formula, it only grows polynomially with the scope in our case (as the formula's length does not change significantly). For example, each pair of nested quantifiers will generate a number of clauses that grows quadratically with the scope. The relational logic constraints generated implicitly by USE for all associations expand similarly. This justifies why the two pattern contracts (i.e., P1 and P2) show the highest translation times; they have the most quantifiers of the 18 constraints.

Using an incremental SAT solver would improve the performance of the prototype. Since most of the generated Boolean formula is the same for all the 18 constraints (i.e., the encoding of classes, associations, multiplicities, and preconditions), we expect that the translation (i.e., the first number in each cell of Table 4) can be done once for the entire verification process; except for $P1$ and $P2$ which differ in their high number of nested quantifiers.

6 Discussion

6.1 Strengths of the Verification Approach

We claim that the verification approach is practical to use for two reasons. First, the used approach provides a fully automated translation from ATL transformations and their constrained metamodels to OCL and relational logic. The

approach further provides a fully automated verification of the generated translation. Even when applied to a realistic case study, the approach scaled to a scope that was large enough to strongly suggest that the analysis did not overlook a bug in the transformation due to the boundedness of the underlying satisfiability solving approach. If we wanted to perform the same verification on a Java implementation of the transformation, we would require equally rich class and operation contracts for, say, Ecore in JML [21]. To the best of our knowledge, no research has explored automatically inferring such contracts. Even then, we expect that the user would have to explicitly specify loop invariants as soon as the transformation contains non-trivial loops, like the loops in our transformation.

Second, the study translates a substantial subset of ATL for verification, i.e., all rules except for imperative blocks, recursive lazy rules and recursive query operations other than relational closures. Thus, the approach takes advantage of the ways declarative, rule-based transformation languages (e.g., ATL) provide to iterate over the input model without requiring recursion or looping. This simplifies verification by, for instance, obviating the need for loop invariants. Although this subset of ATL is not Turing-complete, it can be used to implement many non-trivial transformations. We have statically checked the 131 transformations (comprising 2825 individual rules) in the ATL transformation zoo [36], and 83 of them fall into the described fragment, i.e., neither use recursive rules nor imperative features. Of the remaining 48 transformations, 24 of them that use imperative blocks but no recursion could be expressed declaratively, too.

We conclude that our verification approach greatly benefited from the conceptual simplicity of the declarative fragment of ATL compared to, e.g., a general-purpose programming language such as Java.

6.2 Limitations of the Verification Approach

We identify two limitations of the verification approach.

Correctness of ATL-to-relational-logic translation: Extensive testing and inspection was used to ensure that all steps involved in the translation of ATL and OCL to first-order relational logic are correct. However, in the absence of a formal semantics of ATL and OCL, a formal correctness proof is impossible and the possibility of a bug in the translation remains. This should be taken into account before our approach is used in the context of safety-critical systems.

Bounded search approach: All verification approaches based on a bounded search space cannot guarantee correctness of a transformation because the scopes experimented with may have been too small. The maximum scope sufficient to show bugs in a transformation is transformation-dependent. For example, a transformation with a multiplicity invariant that requires a multiplicity to be 10, will require a scope of 11 to generate a counterexample for that invariant, if any. With respect to our case study, we are confident that a scope of 5 is sufficient to detect violations of the given constraints; we ran analyses with scopes up to 12, because we wanted to study the performance of the approach. Real proofs of unsatisfiability can be created using SMT solvers and quantifier reasoning [9], but the problem is generally undecidable (i.e., the SAT solver does

not terminate on all transformations), and the mapping presented in [9] does not yet cover all language features used in our case study. Further, we have not yet applied any a priori optimizations of the search problem, e.g., metamodel pruning [30], which we plan to apply for future work.

7 Related Work

There are several approaches that translate declarative model transformations into some logic or logic-like language to perform automated verification. Anastasakis *et al.* [3] and Baresi and Spoletini [5] use relational logic and the Alloy analyzer to check for inconsistencies in a transformation. Inaba *et al.* [19] verify the typing of transformations with respect to a metamodel using second-order monadic logic and the MONA solver. Troya and Vallecillo [34] define an encoding of ATL in rewriting logic, that can be used to check the possible executions of a transformation in Maude. Cabot *et al.* [11] translated QVT-R and triple graph grammar transformations into OCL contracts, requiring an OCL model finder to conduct the counterexample checking. Our translation of ATL into OCL (based on [10]) closely resembles this approach. In another previous work [9], we have presented a mapping of ATL directly into first-order logic, using quantifier reasoning to prove transformation properties with SMT solvers.

Asztalos *et al.* [4] formulated transformations and their properties as assertions in first-order logic. A deduction system was implemented to deduce the properties from the rules. Lucio *et al.* [24] verified correctness constraints for transformations in DSLTrans language using a model checker implemented in Prolog. Rensink [28] checked first-order linear temporal properties for graph transformation systems. Becker *et al.* [6] verified a metamodel refactoring implemented as a graph rewriting system by extending the metamodel with predicate structures which were used to specify well-formedness graph constraints. Stenzel *et al.* [31] implemented an algebraic formalization of a subset of operational QVT in the KIV theorem prover.

There are also several approaches that use OCL constraints to specify contracts for model transformations. Guerra *et al.* [18], Gogolla and Vallecillo [16], Braga *et al.* [7], and Cariou *et al.* [12] discussed testing transformations against contracts. In the same vein, Narayanan *et al.* [26] discuss a methodology to specify structural correspondence rules between source and target. Our constraints presented in Sect. 4.2 can be considered a transformation contract in this sense, although we do not use the contracts to test the actual transformation implementation but use them to verify the transformation independent of any input.

Regarding the used approach to check the satisfiability of OCL-constrained models, there are several potential alternatives to the USE Model Validator [23] that we employed. Gonzalez *et al.* [17] implemented the EMFtoCSP model finder that encodes metamodels and OCL constraints as constraint-logic programs (performing bounded verification). Queralt and Teniente [27] implemented a symbolic reasoning procedure for OCL constraints, based on predicate calculus. Brucker *et al.* [8] implemented the HOL-OCL theorem prover to interactively

prove correctness constraints. Jackson *et al.* [20] used the FORMULA tool to reason about metamodels, but they did not support OCL.

The novel aspect of our study is two-fold: First, we have applied an automated verification methodology to an industrial model transformation implemented in the ATL transformation language. Second, we have shown the applicability of this approach to realistic search spaces and discussed the performance of our approach. Most of the referenced research papers evaluate their verification approach on small examples and do not address the performance aspect.

8 Conclusion and Future Work

In this study, we demonstrated how automated verification can be useful in verifying industrial transformations. First, we described the GM-to-AUTOSAR transformation that we have developed for General Motors [29]. We also discussed an automated transformation verification prototype that works on the declarative, non-recursive subset of ATL and its application to our transformation. The prototype was able to uncover two bugs in the transformation that violated two multiplicities in the AUTOSAR metamodel. We further discussed the performance of the verification prototype by showing the translation and constraint solving times for all the constraints over different scopes. The numbers showed that both the Translation times and the Constraint Solving times grow exponentially with the scope. Nonetheless, analysis of the transformation in sufficiently large scopes (up to 12) was possible. We conclude that the application of our verification approach to the case study was successful and provides evidence for its practicality, even in industrial contexts.

For future work, this study can be extended in several ways. First, other industrial transformations should be incorporated in the case study to have a better idea of the practicality of using automated verification on such transformations. Our case study explored a transformation that manipulates metamodels that are considered large on an industrial scale. The transformation, although far from being trivial, does not fully manipulate the two metamodels. We conducted a couple of experiments that show that the verification problem scales almost linearly when more independent rules are added. However, we still need to investigate the performance on larger and more complex transformations. As a result of our demonstration of the effectiveness of our approach in migrating a subset of the GM metamodel to its AUTOSAR equivalent, engineers at General Motors have expressed interest in extending the transformation to the full scope of the GM metamodel. Second, incremental SAT solvers can be used in the bounded search approach to improve the performance and the execution time of the approach, as suggested in Section 5.2. Third, pruning of the manipulated metamodels or the transformation model can be applied before executing the bounded search, as suggested in Section 6.2.

References

1. AUTOSAR Consortium. AUTOSAR System Template, http://AUTOSAR.org/index.php?p=3&up=1&uup=3& uuup=3&uuuup=0& uuuuup=0/AUTOSAR_TPS_SystemTemplate.pdf, 2007.
2. AUTOSAR Consortium. AUTOSAR, http://AUTOSAR.org/, 2007.
3. K. Anastasakis, B. Bordbar, and J. Küster. Analysis of Model Transformations via Alloy. *MoDeVVa*, pages 47–56, 2007.
4. M. Asztalos, L. Lengyel, and T. Levendovszky. Towards Automated, Formal Verification of Model Transformations. In *ICST*, pages 15–24, Paris, France, 2010.
5. L. Baresi and P. Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In *ICGT*, volume 4178 of *LNCS*, pages 306–320, 2006.
6. B. Becker, L. Lambers, J. Dyck, S. Birth, and H. Giese. Iterative Development of Consistency-Preserving Rule-Based Refactorings. *ICMT*, pages 123–137, 2011.
7. C. Braga, R. Menezes, T. Comicio, C. Santos, and E. Landim. On the Specification, Verification and Implementation of Model Transformations with Transformation Contracts. In *SBMF*, volume 7021 of *LNCS*, pages 108–123, 2011.
8. A. D. Brucker and B. Wolff. Semantics, Calculi, and Analysis for Object-Oriented Specifications. *Acta Informatica*, 46(4):255–284, 2009.
9. F. Büttner, M. Egea, and J. Cabot. On Verifying ATL Transformations Using Off-the-Shelf SMT Solvers. In *MODELS*, volume 7590 of *LNCS*, pages 432–448, 2012.
10. F. Büttner, M. Egea, J. Cabot, and M. Gogolla. Verification of ATL Transformations Using Transformation Models and Model Finders. In *ICFEM*, volume 7635 of *LNCS*, pages 198–213, 2012.
11. J. Cabot, R. Clarisó, E. Guerra, and J. de Lara. Verification and Validation of Declarative Model-to-Model Transformations Through Invariants. *Systems and Software*, 83(2):283–302, 2010.
12. E. Cariou, N. Belloir, F. Barbier, and N. Djemam. OCL Contracts for the Verification of Model Transformations. *EASST*, 24, 2009.
13. T. Cottenier, A. Van Den Berg, and T. Elrad. The Motorola WEAVR: Model Weaving in a Large Industrial Context. In *AOSD*, volume 32, Vancouver, Canada, 2007.
14. A. Daghsen, K. Chaaban, S. Saudrais, and P. Leserf. Applying Holistic Distributed Scheduling to AUTOSAR Methodology. In *ERTSS*, Toulouse, France, 2010.
15. H. Giese, S. Hildebrandt, and S. Neumann. Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. *Graph Transformations and Model-Driven Engineering*, 5765:555–579, 2010.
16. M. Gogolla and A. Vallecillo. Tractable Model Transformation Testing. In *ECMFA*, pages 221–236, Birmingham, UK, 2011.
17. C. A. González Pérez, F. Büttner, R. Clarisó, and J. Cabot. EMFtoCSP: A Tool for the Lightweight Verification of EMF Models. In *FormSERA*, pages 44–50, Zurich, Switzerland, 2012.
18. E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck, and W. Schwinger. Automated Verification of Model Transformations Based on Visual Contracts. *Automated Software Engineering*, 20(1):5–46, 2013.
19. K. Inaba, S. Hidaka, Z. Hu, H. Kato, and K. Nakano. Graph-Transformation Verification Using Monadic Second-Order Logic. In *PPDP*, pages 17–28, 2011.

20. E. Jackson, T. Levendovszky, and D. Balasubramanian. Automatically reasoning about metamodeling. *SoSyM*, pages 1–15, 2013.
21. B. Jacobs and E. Poll. A Logic for the Java Modeling Language JML. In *FASE*, volume 2029 of *LNCS*, pages 284–299, 2001.
22. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.
23. M. Kuhlmann, L. Hamann, and M. Gogolla. Extensive Validation of OCL Models by Integrating SAT Solving into USE. In *TOOLS*, volume 6705 of *LNCS*, pages 290–306, 2011.
24. L. Lúcio, B. Barroca, and V. Amaral. A Technique for Automatic Validation of Model Transformations. *MODELS*, pages 136–150, 2010.
25. P. Mohagheghi and V. Dehlen. Where is the Proof?-A Review of Experiences from Applying MDE in Industry. In *ECMDA–FA*, pages 432–443, 2008.
26. A. Narayanan and G. Karsai. Verifying Model Transformations by Structural Correspondence. *EASST*, 10(0), 2008.
27. A. Queralt and E. Teniente. Verification and Validation of UML Conceptual Schemas with OCL Constraints. *TOSEM*, 21(2):13, 2012.
28. A. Rensink. Explicit State Model Checking for Graph Grammars. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 114–132, 2008.
29. G. Selim, S. Wang, J. Cordy, and J. Dingel. Model Transformations for Migrating Legacy Models: An Industrial Case Study. *ECMFA*, pages 90–101, 2012.
30. S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel. Meta-model Pruning. In *MODELS*, volume 5795 of *LNCS*, pages 32–46, 2009.
31. K. Stenzel, N. Moebius, and W. Reif. Formal Verification of QVT Transformations for Code Generation. *MODELS*, pages 533–547, 2011.
32. M. Tisi, F. Jouault, P. Fraternali, S. Ceri, and J. Bézivin. On the Use of Higher-Order Model Transformations. In *ECMDA-FA*, volume 5562 of *LNCS*, 2009.
33. E. Torlak and D. Jackson. Kodkod: A Relational Model Finder. In *TACAS*, volume 4424 of *LNCS*, 2007.
34. J. Troya and A. Vallecillo. A Rewriting Logic Semantics for ATL. *Journal of Object Technology*, 10:5: 1–29, 2011.
35. The USE Validator. available online, http://sourceforge.net/projects/useocl/files/Plugins/ModelValidator/.
36. The ATL Transformation Zoo. available online, http://www.eclipse.org/atl/atlTransformations/.