Temporal Logic Specification and Analysis for Model Transformations

S. Yassipour-Tehrani[1], K. Lano[1]

[1]Dept of Informatics, King's College London, Strand, London, UK

Abstract. In this paper we outline an approach for using temporal logic specifications and model-checking tools to express and verify model transformation properties. Linear Temporal Logic (LTL) is used to express transformation semantics, and the SMV formalism is used to encode this semantics and to perform model checking.

1 Introduction

Model transformations may be defined in a large number of different languages such as ATL [6], QVT-R [13], QVT-O, ETL [7], and UML-RSDS [9]. In [11, 10] we have described work on the verification of transformations based on translations from MT languages into a semantic representation or intermediate language, and formal analysis upon this representation. This translation explicitly encodes some aspects of the computational model of the MT languages (such as the two-phase execution of ATL transformations), however fine-grain temporal orderings (eg., that QVT-R relations quoted in the *when* clause of another relation should execute prior to the quoting relation [13]) are only implicitly expressed.

Most declarative and hybrid transformation languages have a common execution semantics which consists of applying individual transformation rules to specific source model elements. The order of rule applications may be only partly constrained by the specification. This execution model fits well with the concepts of linear time temporal logic, which defines properties over sequences of states using the operators $\bigcirc$ (in the next state), $\diamond$ (in the current or some future state), $\Box$ (in the current and all future states), $\blacklozenge$ (strictly in the past). For example, to express that some properties *Inv* are invariant throughout the transformation execution, given a precondition Asm, we could write:

$$Asm \Rightarrow \Box Inv$$

Thus LTL is an appropriate formalism to enhance the approach of [11] with detailed computation semantics.

Temporal logic is also useful to define transformation requirements in a formal but language-independent manner. This use of LTL accords with the KAOS requirements engineering method [15]. In KAOS, system goals are formalised according to the pattern of behaviour they require. Four generic goal patterns are identified by KAOS:

1. *Achieve*: $G \Rightarrow \diamond Q$ – Q holds in some future state.
2. *Cease*: $G \Rightarrow \diamond \neg\ Q$ – There will be some point in the future where Q does not hold
3. *Maintain*: $G \Rightarrow \Box Q$ – Q holds in all future states
4. *Avoid*: $G \Rightarrow \Box \neg\ Q$ – Q will never hold in the future.

Transformation postconditions Q are a special case of the Achieve pattern, where Q remains true after it has been established: $G \Rightarrow \diamond \Box Q$. The Cease pattern applies particularly for quality improvement requirements of refactoring transformations: that at some stage in the transformation, an example of poor structure/low quality in the model will be eliminated. The Maintain pattern applies for transformation invariants. The Avoid pattern applies for safety requirements, where the transformation (eg., for a safety-critical control functionality) must never enter some hazardous states [3, 4].

In Section 2 we identify how transformation semantics can be expressed in LTL. In Section 3 we describe how this semantics can be represented in the SMV/nuSMV formalism. In Section 4 we give a specific example of the approach applied to the VOLT UML to RDB case study.

2 Transformation semantics and temporal logic

A model transformation τ is typically specified by a finite collection of *transformation rules*, r_1, ..., r_n which apply a source-target mapping or in-place source model rewrite for source model elements which satisfy the enabling condition $en(r_i)$ of the rule[1]. The application of a rule r to specific elements p is written as $r(p)$. Logically, $r(p)$ can be interpreted as an LTL proposition that the application occurs at the present time. For simplicity in this paper we will assume there is only one parameter or 'pivot element' $s : S_i$ for each rule r_i. An execution history of τ is then a sequence of rule applications, each of which can only occur within its enabling condition (*WithinEnabling*):

$$\Box(\forall\, s : S_i \cdot r_i(s) \ \Rightarrow\ en(r_i(s)))$$

For separate-models transformations where enabled rules remain enabled until they are applied, and are then disabled, there are the axioms:

$$\Box(\forall\, s : S_i \cdot en(r(s)) \Rightarrow \diamond r(s))$$

For some languages (such as QVT-R and ATL) there is a distinction between top-level and subordinate rules: subordinate rules r can only occur if they are invoked by another rule (*InvokedRules*):

$$\Box(r(s') \ \Rightarrow\ \bigvee\nolimits_i \exists\, s : S_i \cdot r_i(s))$$

[1] The enabling condition may include both positive and negative application conditions.

where the disjunction ranges over all other rules r_i which directly invoke r, this disjunction is *false* if there are no such rules, so that r cannot occur in such a case. At each step only one top-level rule application can occur (*RuleExclusion*):

$$\Box(\forall\, s, s' : S_i \cdot r_i(s) \wedge s' \neq s \;\Rightarrow\; \neg\, r_i(s'))$$

for each top-level r_i, and $\Box(\forall\, s : S_i \cdot r_i(s) \;\Rightarrow\; \neg\, (\exists\, s' : S_j \cdot r_j(s')))$ for each top-level rule r_j, $i \neq j$.

In QVT-R (separate-models mode) and ATL, a top relation cannot be re-applied to the same model element (*NoReapplication*):

$$\Box(\forall\, s : S_i \cdot r_i(s) \;\Rightarrow\; \bigcirc\, \Box \neg\, r_i(s))$$

for all top-relations r_i.

To express that one rule r' always has priority over another rule r, we can write (*Priority*):

$$\Box(en(r'(s)) \wedge en(r(s)) \;\Rightarrow\; \neg\, r(s))$$

The following conditions may be checked in order to verify a transformation or to identify errors in its specification:

- Any required invariant *Inv* is maintained throughout the transformation execution: $Asm \;\Rightarrow\; \Box Inv$
- Any required overall postcondition must eventually be true (and remain true): $Asm \;\Rightarrow\; \diamond\, \Box Post$
- Non-triviality: that each top-level rule will eventually be applied: $\diamond r(p)$

Such properties can be checked for specific transformations using a LTL theorem prover such as SPASS, or a model checker, such as SMV/nuSMV [12].

3 Encoding transformation semantics in SMV

We utilise the SMV/nuSMV language and model checker to analyse temporal properties of transformations. We select SMV because it is an established formalism which supports LTL, and because there is already a mapping from UML to SMV implemented in the UML-RSDS tools [9]. SMV models systems in *modules*, which may contain variables ranging over finite domains such as subranges a..b of natural numbers and enumerated sets. The initial values of variables are set by initialisation statements $init(v) := e$. A $next(v) := e$ statement identifies how the value of variable v changes in execution steps of the module. In the UML to SMV encoding of [9], objects are modelled by integer object identities, attributes and associations are represented as variables, and classes are represented by modules parameterised by the object identity values (so that separate copies of the variables exist for each object of the class). A specific numeric upper bound must be given for the number of possible objects of each class. Models of arbitrary size can be represented by varying this bound. A transformation

is modelled by modelling transformation execution steps as module execution steps: the application of a particular rule to particular pivot instance(s). The temporal logic operators are denoted in SMV by **G** (for $\Box$), **F** (for $\diamond$), **X** (for $\bigcirc$) and **O** (for $\blacklozenge$).

The structure of an SMV specification of a class diagram is as follows:

```
MODULE main
VAR
  C : Controller;
  MEntity1 : Entity(C,1);
  .... other object instances ....

MODULE Controller
VAR
  Entityid : 1..n;
  event : { createEntity, killEntity, event1, ..., eventn };

MODULE Entity(C, id)
VAR
  alive : boolean;
  ... attributes of Entity ...
DEFINE
  TcreateEntity := C.event = createEntity & C.Entityid = id;
  TkillEntity := C.event = killEntity & C.Entityid = id;
  Tevent1 := C.event = event1 & C.Entityid = id & alive = TRUE;
  ... transitions for each event ...
ASSIGN
  init(alive) := FALSE;

  next(alive) :=
    case
      TcreateEntity : TRUE;
      TkillEntity : FALSE;
      TRUE : alive;
    esac;
```

Each class *Entity* is represented in a separate module, and each instance is listed as a module instance in the main module. System events are listed in the Controller, and the effect of these events on specific objects are defined in the module specific to the class of that object. The value of the *Controller* variable *event* in each execution step identifies which event occurs, and the value of the appropriate *Eid* variable indicates which object of class E the event occurs on. Associations $r : A \rightarrow B$ of 1-multiplicity are represented as attributes

```
r : 0..bcard
```

where *bcard* is the maximum permitted number of objects of B. Events to set and unset this role are included. 0 represents the null object (for 0..1-multiplicity roles). For *-multiplicity unordered roles $r : A \rightarrow Set(B)$, an array representation is used instead:

```
r : array 1..bcard of 0..1
```

with the presence/absence of a B element with identity value i being indicated by $r[i] = 1$ or $r[i] = 0$. Operations to add and remove elements are provided. The Controller Aid and Bid variables identify which links are being added/removed.

For separate-model (mapping) model transformations, this representation is extended as follows:

- No *killE* or *createE* events for source model entities are included: the source model is assumed to exist initially and its elements are read-only.
- Both source model and target model entity types are represented by SMV modules. Source entity attributes are defined as frozen variables (they cannot be changed after initialisation). The transformation steps are defined as controller events, with the effect of these steps being defined in one or more target entity modules. Target entity modules have the modules of the source entities they depend upon as parameters.

In this encoding the axioms (*RuleExclusion*) and (*InvokedRules*) are true by construction of the model, whilst (*WithinEnabling*) is a property which should be assumed. (*NoReapplication*) is added as a logical assumption if the transformation satisfies this semantics.

4 Example specification and analysis

We consider the execution semantics and analysis of QVT-R, and specifically analyse the case study transformation, from UML to relational databases, as given in [13]:

```
transformation uml2rdb(uml1 : SimpleUML, rdb1 : SimpleRDMS) {
  top relation Package2Schema
  { checkonly domain uml1 p : Package { name = pn }
    enforce domain rdb1 s : Schema { name = pn }
  }

  top relation Class2Table
  { checkonly domain uml1 c : Class { namespace = p : Package {}, name = n };
    enforce domain rdb1 t : Table { schema = s : Schema {}, name = n };
    when { Package2Schema(p,s); }
    where { Attribute2Column(c,t); }
  }

  relation Attribute2Column
  { checkonly domain uml1 c : Class { attribute =
        a : Attribute { name = an, type = typ : Type { name = tn } } };
    enforce domain rdb1 t : Table { column =
        col : Column {name = an, coltype = tn} };
  }
}
```

A computation step of a QVT-R transformation consists of the application of a top relation to specific model elements matching its source domains. The application of non-top relations invoked (in a *where* clause) directly or indirectly from this top relation application are included in the computation step.

In contrast, a relation application in a *when* clause indicates that the application must have occurred strictly prior to the current relation application: indeed the previous application of the relation is part of the enabling condition of the current relation (*WhenCond*):

$$en(Class2Table(c)) \Rightarrow \blacklozenge Package2Schema(c.namespace)$$

An individual application of *Package2Schema* must achieve its specific post-condition (*PostDef*):

$$Package2Schema(p) \Rightarrow \bigcirc \exists\, s : Schema \cdot s.name = p.name$$

Likewise for *Class2Table* and *Attribute2Column*. A desirable invariant of the transformation is that the only schemas which exist are those that correspond to a package: $\forall\, s : Schema \cdot \exists\, p : Package \cdot s.name = p.name$. There are similar properties for tables and columns.

Model-checking can identify errors in such specifications, using the LTL semantics. For example, if the specifier had mistakenly written *Attribute2Column* in the *when* clause of *Class2Table*, and *Package2Schema* in the *where* clause. This would mean that *Attribute2Column* cannot occur (by axiom *InvokedRules*), and consequently that *Class2Table* cannot occur (by *WhenCond*). This error would be identified by the generation of a counter-example to the non-triviality property (Section 2).

In QVT-R, priorities between rules are typically enforced using flag variables and specific conditions depending upon these in *when* clauses. Model-checking can verify if such a scheme correctly achieves the required priority ordering.

To map QVT-R to SMV, we first map QVT-R to UML-RSDS, using the UML-RSDS tools [9], and then use the UML to SMV translator to define the basic SMV structure for the transformation. If a transformation rule r iterates over $s : S_i$ and has the UML-RSDS form

$$Ante \Rightarrow T_j \rightarrow exists(t \mid TCond \ \&\ P)$$

then the SMV module for T_j has a parameter of module type S_i, and r is an event of the *Controller*, and $S_i id$ is a Controller variable. An axiom asserts that $r(s)$ can only take place if *Ante* holds.

The T_j module has a transition defined as

```
Tr := C.event = r & C.Siid = id
```

identifying that the application of r takes place on the S_i object with id equal to the current T_j object. The updates to features f of t defined in $TCond$, P are

then specified by $next(f)$ statements, using Tr as a condition, and $next(alive)$ is set to $TRUE$ under condition Tr.

The restrictions of SMV/nuSMV imply that only attributes and expressions of the following kinds can be represented in SMV:

- Booleans and boolean operations.
- Integers and operations on integers within a bounded range.
- Strings represented as elements of an enumerated type. String operations cannot be represented.

Currently the details of transformation steps are manually encoded in SMV.

Part of the SMV specification for the example transformation, in a configuration with one package and two classes, is:

```
MODULE main
VAR
  C : Controller;
  MPackage1 : Package(C,1);
  MClass1 : Class(C,1);
  MClass2 : Class(C,2);
  MSchema1 : Schema(C,MPackage1,1);
  MTable1 : Table(C,MClass1,1);
  MTable2 : Table(C,MClass2,2);

MODULE Controller
VAR
  Packageid : 1..1;
  Classid : 1..2;
  Schemaid : 1..1;
  Tableid : 1..2;
  event : { Package2Schema, Class2Table };

MODULE Package(C, id)
FROZENVAR
  name : { string1, string2, string3 };

MODULE Schema(C, P, id)
VAR
  alive : boolean;
  name : { empty, string1, string2, string3 };
DEFINE
  TPackage2Schema := C.event = Package2Schema & C.Packageid = id;
ASSIGN
  init(alive) := FALSE;

  next(alive) :=
    case
      TPackage2Schema : TRUE;
      TRUE : alive;
    esac;
```

```
init(name) := empty;

next(name) :=
  case
    TPackage2Schema : P.name;
    TRUE : name;
  esac;
```

This represents the effect of the $Package2Schema(p)$ event, as the creation of a Schema instance and the setting of its name. The axioms ($PostDef$) are ensured by construction of this model, whilst the axioms ($WhenCond$) need to be assumed as conditions. For this transformation these axioms would be expressed as assumptions:

$$\Box(C.event = Class2Table \wedge C.Classid = c \Rightarrow \blacklozenge(C.event = Package2Schema \wedge C.Packageid = MClassc.namespace))$$

asserted for each integer c from 1 up to the number of Class instances in the source model. ($NoReapplication$) is expressed by assumptions

$$\Box(C.event = Class2Table \wedge C.Classid = c \Rightarrow \bigcirc\Box(C.event = Class2Table \Rightarrow C.Classid \neq c))$$

for each class identity c, and similarly for the other top relations.

The effect of the *where* clause of $Class2Table$ is expressed in the *Column* module, as the (simultaneous) creation and update of columns for each attribute linked to the transformed class. The invariant that each existing schema corresponds to a package can be verified in nuSMV as:

```
LTLSPEC G(MSchema1.alive = TRUE  ->  MSchema1.name = MPackage1.name)
```

This could alternatively be expressed by an INVAR assertion.

The results of counter-example analysis are presented as sequences of states which directly correspond to traces of the transformation execution. These can be readily understood in terms of the original model and transformation rules, or could be processed into a graphical form. An example counterexample trace is:

```
-- specification  G ((C.event = Class2Table & C.Classid = 1) ->
      O(C.event = Package2Schema & C.Packageid = MClass1.namespace))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  C.Classid = 1
  C.event = Class2Table
  C.Tableid = 1
  MPackage1.name = string1
  MClass1.name = string1
```

```
  .... details omitted ....
  MTable1.alive = FALSE
  MTable1.name = empty
  MSchema1.TPackage2Schema = FALSE
  MTable1.TClass2Table = TRUE
  MTable2.TClass2Table = FALSE
-> State: 1.2 <-
  C.event = Package2Schema
  MTable1.alive = TRUE
  MTable1.name = string1
  MSchema1.TPackage2Schema = TRUE
  MTable1.TClass2Table = FALSE
...
```

Validated properties are reported as true, for example:

```
NuSMV > read_model -i volt.smv
NuSMV > go
-- specification  G (MSchema1.alive = TRUE -> MSchema1.name = MPackage1.name)  is true
NuSMV >
```

We tested the efficiency of the nuSMV analysis using source models of sizes 3 (1 package and 2 classes), 6 (1 package and 5 classes), 11 (1 package and 10 classes) and 21 (1 package and 20 classes), with the invariant property `G (MSchemai.alive = TRUE -> MSchemai.name = MPackagei.name)` being checked for all classes and packages i. Table 1 shows the results.

Source model size	*Formula size*	*Time taken*
3	2 conjuncts	less than 1s
6	5 conjuncts	1s
11	10 conjuncts	5 minutes
21	20 conjuncts	Out of memory

Table 1. Invariant verification times

5 Related work

Model checkers and satisfaction checkers such as Alloy and Z3 have been applied to identify counter-examples to required transformation outcomes [1, 8]. Some temporal logic analysis (in ASK-CTL) is supported by the translation from QVT-R to coloured Petri Nets in [5]. A manual translation from graph transformations (in AGG) to the Bogor model checker is described in [2]. Model-checking techniques for graph transformations are compared in [14]: a translation into SPIN/Promela is compared with the GROOVE checker, which operates directly upon graph transformations. The new contribution of this paper is to define a general temporal logic semantics for MT languages, which can be specialised

to reflect the semantics of individual languages. The temporal logic semantics enhances the existing translations from MT languages to $\mathcal{TMM}$/UML-RSDS by defining the permitted computation sequences of transformations. Analysis in nuSMV provides an additional semantic analysis enabling proof and counter-example detection for transformations. The nuSMV checker supports expression of invariants, variants, time bounds between events and regular expression constraints on transformation execution histories via our encoding [12].

6 Conclusions

We have described an approach for the specification and analysis of model transformations using linear temporal logic and the nuSMV model checker. This approach has been implemented by a translation from UML-RSDS to nuSMV.

References

1. K. Anastasakis, B. Bordbar, J. Kuster, *Analysis of Model Transformations via Alloy*, Modevva 2007.
2. L. Baresi, V. Rafe, A. Rahmani, P. Spoletini, *An efficient solution for model checking graph transformation systems*, ENTCS, May 2008.
3. B. Becker, D. Beyer, H. Giese, F. Klein, D. Schilling, *Symbolic invariant verification for systems with dynamic structural adaption*, ICSE 2006, ACM Press.
4. B. Becker, L. Lambers, J. Dyck, S. Birth, H. Giese, *Iterative development of consistency-preserving rule-based refactorings*, ICMT 2011, LNCS vol. 6707, 2011.
5. E. Guerra, J. de Lara, *Colouring: execution, debug and analysis of QVT-Relations transformations through coloured Petri Nets*, SoSyM vol. 13, no. 4, Oct 2014.
6. Eclipsepedia, *ATL User Guide*, http://wiki.eclipse.org/ATL/ User_Guide_-_The_ATL_Language, 2014.
7. D. Kolovos, R. Paige, F. Polack, *The Epsilon Transformation Language*, in ICMT 2008, LNCS Vol. 5063, pp. 46–60, Springer-Verlag, 2008.
8. K. Lano, S. Kolahdouz-Rahimi, T. Clark, *Comparing verification techniques for model transformations*, Modevva workshop, MODELS 2012.
9. K. Lano, *The UML-RSDS Manual*, www.dcs.kcl.ac.uk/staff/kcl/uml2web/umlrsds.pdf, 2014.
10. K. Lano, S. Kolahdouz-Rahimi, T. Clark, *Language-independent Model Transformation Verification*, VOLT 2014, York, July 2014.
11. K. Lano, S. Kolahdouz-Rahimi, T. Clark, *A framework for model transformation verification*, BCS FACS journal, 2014.
12. NuSMV, http://nusmv.fbk.eu, 2015.
13. OMG, *MOF 2.0 Query/View/Transformation Specification v1.1*, 2011.
14. A. Rensink, A. Schmidt, D. Varro, *Model checking graph transformations: A comparison of two approaches*, ICGT 2004, LNCS 3256, pp. 226-241, 2004.
15. A. Tang, *A rationale-based model for architecture design reasoning*, Faculty of ICT, Swinburne University of Technology, 2007.