Transformation of UML and OCL Models into Filmstrip Models[⋆]

Frank Hilken, Lars Hamann, and Martin Gogolla

University of Bremen
{fhilken,lhamann,gogolla}@informatik.uni-bremen.de

Abstract. This contribution presents an automatic transformation from UML and OCL models into enriched UML and OCL models, so-called filmstrip models, which embody temporal information when employing OCL while maintaining the same functionality as the original model. The approach uses a combination of object and sequence diagrams that allows for a wide range of possible OCL constraints about sequences of operation calls and their temporal properties. The modeler does not need to account for such properties while creating the original model. Errors found by constraints for the filmstrip model can easily be related back to the original model, as the elements of the filmstrip model are synchronized with the original model and the backwards calculation is generally simple. The approach is implemented in a UML and OCL modeling tool.

1 Introduction

In recent years, the Unified Modeling Language (UML) has become the standard language for modeling IT systems. Among the various UML diagram forms, UML class diagrams are the most frequently used ones. One way (among other possibilities) to completely specify structure and behavior of an application is to enrich class diagrams with class invariants and operation pre- and postconditions expressed in the Object Constraint Language (OCL). The starting point for this contribution is an application model solely described by a class diagram and OCL constraints. In the development process, it is essential to validate and verify that such an application model meets the informal and formal postulated requirements.

For structural models with class diagrams and invariants, a number of efficient validation and verification techniques [2, 13, 4, 11, 17] are available. These techniques partly transform UML models including OCL invariants into validation and verification platforms (like SAT or SMT solvers or relational logic) allowing an efficient check of relevant structural properties of the UML model in terms of the target platform. However, less attention has been paid to behavioral model properties, in particular to operation pre- and postconditions.

This contribution proposes a transformation from a UML and OCL application model with pre- and postconditions and invariants into a UML model with

[⋆] This work was partially funded by the German Research Foundation (DFG) under grant GO 454/19-1.

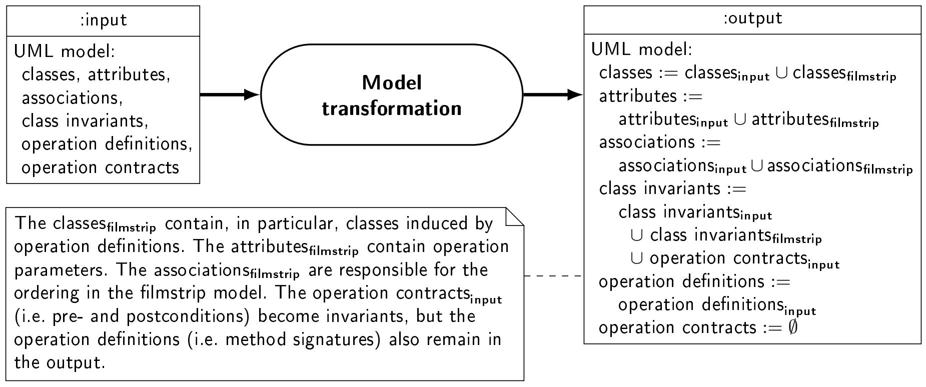
:input
UML model:
classes, attributes,
associations,
class invariants,
operation definitions,
operation contracts
Model transformation
:output
UML model:
classes := $\text{classes}_{\text{input}} \cup \text{classes}_{\text{filmstrip}}$
attributes := $\text{attributes}_{\text{input}} \cup \text{attributes}_{\text{filmstrip}}$
associations := $\text{associations}_{\text{input}} \cup \text{associations}_{\text{filmstrip}}$
class invariants := $\text{class invariants}_{\text{input}} \cup \text{class invariants}_{\text{filmstrip}} \cup \text{operation contracts}_{\text{input}}$
operation definitions := $\text{operation definitions}_{\text{input}}$
operation contracts := $\emptyset$
The $\text{classes}_{\text{filmstrip}}$ contain, in particular, classes induced by operation definitions. The $\text{attributes}_{\text{filmstrip}}$ contain operation parameters. The $\text{associations}_{\text{filmstrip}}$ are responsible for the ordering in the filmstrip model. The operation $\text{contracts}_{\text{input}}$ (i.e. pre- and postconditions) become invariants, but the operation definitions (i.e. method signatures) also remain in the output.

Fig. 1. Inputs and outputs of the filmstrip transformation

OCL invariants only (thus without pre- and postconditions). The intention is that this filmstrip model can then be handled by one of the efficient techniques available for structural models. All behavioral aspects of the original application model are equivalently expressed in a so-called filmstrip model in form of structural constraints, i.e., invariants. Figure 1 gives an overview of the inputs and outputs of the transformation.

There are a number of reasons for us to study the proposed transformation. Alloy [13], for example, has to model temporal system development with explicit relations for objects representing points in time, and these relations have to be described by the developer. Our approach comprises an automatic way to handle temporal system development on the basis of pre- and postconditions. On the other hand, Alloy nicely demonstrates that design flaws concerning dynamics can be successfully detected by structural techniques. A further motivation for us to study the current transformation is a fundamental question about the relationship between structure and behavior and to find out to what extent structural techniques can encode dynamic problems. We also expect that structural automatic validation and verification techniques will show major advances in coming years, as they have shown in recent years.

The challenge of building the filmstrip model is to create a model that does not change the behavior and expressiveness of the application model, but offers more possibilities for validation and verification by employing OCL for checking behavioral properties on the filmstrip model and to automatically translate the detected properties back to the application model: The filmstrip model captures several application model states in one object diagram; it keeps information about successive operation calls and changes between the application model states; pre- and postconditions are transformed into invariants and make behavioral properties from application model sequence diagrams detectable in a single filmstrip model object diagram. The approach allows to give feedback on the application model in form of scenarios and test cases that are directly understandable and analyzable by the application model developer. The filmstrip model also enables the use of temporal logic properties formulated using an extension of standard OCL [20, 3, 18].

Another feature of our approach is that it can be used for checking properties of model transformations themselves. Let us assume that a model transformation consists of separate operations described with pre- and postconditions (for example, given a graph transformation system, each rule becomes an operation), the filmstripped model transformation can be checked for confluence of rules: within a finite search space our approach can build scenarios for rule applications.

Our work is related to several other papers using filmstrip models for various different tasks. The first known notion of the idea is in [8]. The authors of [10] take the filmstrip idea and employ it as part of three-dimensional visualizations within software design. [19, 1] define a different approach for a filmstrip model (called snapshot model or snapshot transition model), which changes more of the original model elements instead of using abstract interface elements. In [5] filmstrips are used as a device for functional testing. [12] shows a less generic approach, with less separation between application model and filmstrip model.

Multiple approaches of an extension of OCL with temporal logic exist in order to verify temporal properties in UML and OCL models, but only a few keep the verification task on the UML and OCL layer. [18] and [14] give a comparison of the different approaches. [6] concentrated on temporal business rules without giving a full semantic definition. [20] gives a semantic definition of linear temporal logic operators. [9] focused on the integration of time bounds in connection with temporal constructs. In [1] temporal OCL expressions are evaluated in state transition systems – a similar form of filmstrip models using a more relational database-like approach.

The rest of this paper is structured as follows. In Sect. 2 the example model for this paper is described, its properties are explained and an example system state is shown. Section 3 covers the transformation of the UML part of the model transformation and Sect. 4 covers the OCL part respectively. Section 5 completes the example and shows further examples of use for the filmstrip model. Section 6 describes the implementation of the transformation and Section 7 finishes the paper with a conclusion and discusses future work.

2 Running Example

The input for the transformation is a UML and OCL model consisting of a class diagram describing an application completely with classes, attributes, associations, operations and invariants. The operations – with their pre- and postconditions – describe the model dynamics, which can be visualized in sequence diagrams. The other characteristics of the model, e.g. invariants and multiplicities describe the allowed system states, which are represented by object diagrams. We call this model the application model.

As an example for this paper, the transformation of a classic process scheduler application model [16, 7] into its filmstrip counterpart is demonstrated. Figure 2 shows the class diagram of the transformed model. The original application

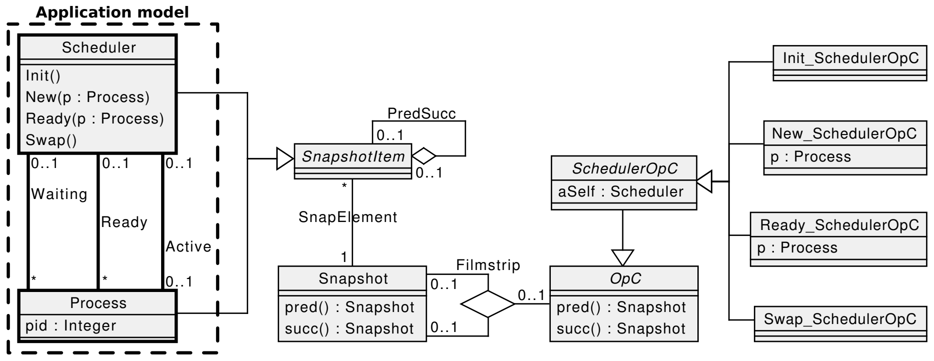

Fig. 2. Scheduler filmstrip model with the contained application model highlighted

model, consisting of the two classes `Scheduler` and `Process` and three associations `Active`, `Ready` and `Waiting`, is completely contained in the filmstrip model and displayed in the dashed box in the left part of the picture. The structure of such model is unchanged.

The class `Process` represents the processes of the system and has one attribute `pid` to distinguish them. The class `Scheduler` represents a scheduler which is connected to the processes via three associations. They link the currently *active* process, which may be none, the ones that are *ready* to be scheduled and the ones *waiting* for an action to become ready again.

Additional constraints, not expressible in UML, are specified using OCL. The OCL constraints marks the sets of ready and waiting processes of a scheduler to always be disjoint, the active process is not simultaneously ready or waiting and when there is no active process, there may not be a ready process. Lastly the process identifiers (`pid`) of the class `Process` must be unique.

The scheduler class has all the functionality of the system. The first operation *initializes* the scheduler into a defined start state. The second operation *New* registers a process to the scheduler and puts it in the list of waiting processes. The third operation moves a process from the list of waiting processes into the list of *ready* processes, unless there is no active process, in which case the process will immediately become the active one. The fourth and last operation *swaps* the active process, putting it into the list of waiting processes and schedules another ready process, if there is any. The general flow of a process therefore is as follows:

$$\text{Unassigned} \xrightarrow{\text{New}} \Big(\text{Waiting} \xrightarrow{\text{Ready}} \text{Ready} \xrightarrow{\text{Swap}^*} \text{Active} \xrightarrow{\text{Swap}} \text{Waiting}\Big)^*$$

The states represent how a process is connected to the scheduler and the arrows describe operation calls on the scheduler between the state changes. The sequence is focusing on one process. An *Unassigned* process is not connected to any scheduler, yet. It gets assigned to a scheduler by a *New* operation call and is then permanently assigned to this scheduler, where it continues *Waiting*. Here it waits for a *Ready* operation call to get into a *Ready* state. When the scheduler now issues a *Swap* operation call and this process is chosen, it will become the

Active process. The next *Swap* operation call will then bring the process back into the *Waiting* state and the cycle repeats. The notation "Swap*" suggests, that several swap operation calls might be necessary before a specific process becomes the active one. Also the flow of a process might vary slightly depending on the number of ready processes, e.g. a process can become the active process as soon as it is ready, if there is no other active process.

The rest of Fig. 2 shows the model parts specific to the filmstrip model. The classes of the application model are modified to inherit from the abstract class `SnapshotItem`. This abstract class provides the connection to the class `Snapshot` to link each object to a certain snapshot and the aggregation `PredSucc` to describe a temporal connection between two object instances. To represent progression of objects during operation calls, multiple objects are used in the filmstrip model with the delta being the changes applied in the course of an operation invocation. Thus an association is required to guarantee that every object of an application model class is linked to a unique snapshot. The association `PredSucc` connects objects that represent one instance.

The next class added to the model is the `Snapshot` class which represents a reference point for a system state in the application model. With the abstract class for representing operation calls (`OpC`), the snapshot is also linked to its predecessor and successor in the same way as the application model classes are. This ternary association is called `Filmstrip` and links two snapshot objects and an operation call object together, representing one operation call. The resulting object diagrams of this structure involve a sequence of snapshots (system states) with operation calls linked in between them, like a filmstrip consists of many consecutive pictures that change from frame to frame.

The possible operation calls of the application model are added to the filmstrip model as classes derived from the interface `OpC`. In the example in Fig. 2 the abstract class `SchedulerOpC` has an attribute `aSelf` which saves the object, this operation is invoked on. This is the base class for every concrete class representing an operation of the class `Scheduler`. These classes store the dynamic information, e.g. parameter values, that occur during an operation call.

An example system state of the filmstrip model is shown in Fig. 3. A scenario in the application model can be represented with an object diagram sequence to show the different states and a sequence diagram to represent the operation calls. The filmstrip model combines this information into a single system state. The main problem is to find a transformation that can reproduce the complete behavior of the application model and nothing more.

Further challenges of the transformation include the consistent handling of: (1) the insertion of new root elements for filmstrip models into the existing model; (2) the change of model classes and operations; and (3) the correct adaptation of OCL constraints.

3 UML Transformation

The process of a filmstrip transformation is an endogenous model transformation. The changes take place solely in the class diagram. This section explains the

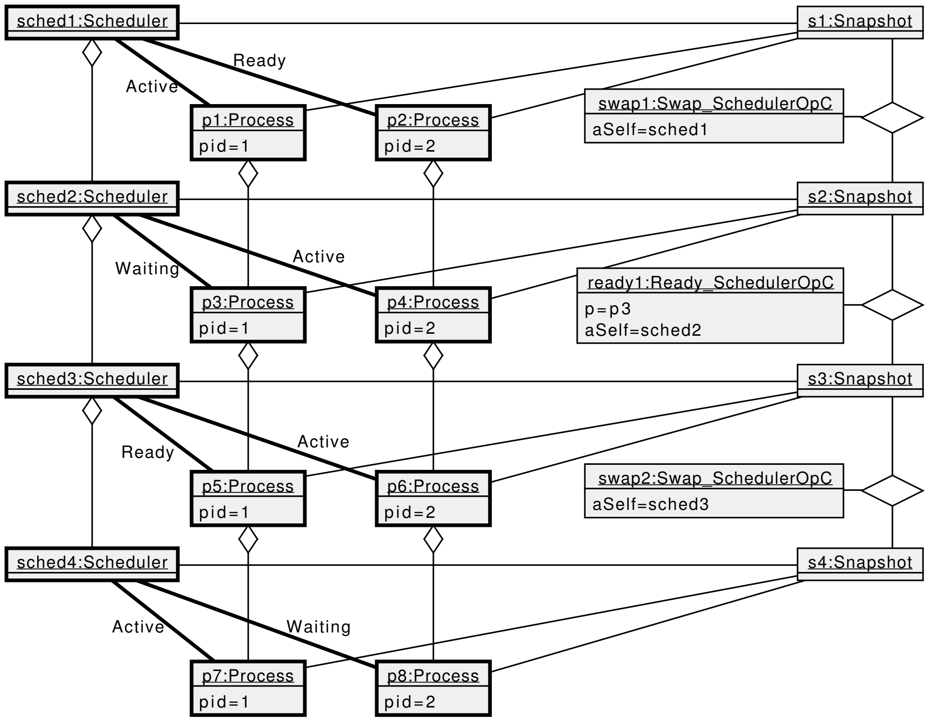

Fig. 3. Example system state of the scheduler filmstrip model with elements from the application model highlighted

steps required to transform an application model into a filmstrip model regarding the UML elements. Further constraints on these elements that are required for a correct behavior, but are not expressible in UML, i.e. OCL invariants, are discussed in Section 4. UML model elements that are not mentioned, e.g. associations and operations, remain the same in the filmstrip model. Figure 4 gives an overview for the steps of the whole transformation process.

3.1 Filmstrip Core Elements

First the core of the filmstrip model is included into the application model. These elements are shown in Fig. 5 and are the same in every filmstrip model. They consist of three classes and three associations and define the functionality of the filmstrip model. They also provide an interface for elements of the application model classes to enable interaction with them (`SnapshotItem`).

The class `Snapshot` represents a system state of the application model where any object linked to a snapshot belongs to the system state represented by it. To represent multiple system states in one object diagram multiple snapshot instances are used. An object diagram may contain several snapshots that represent the same system state, i.e. the properties of the linked objects are equal and the system state would be identical in the application model. This is required

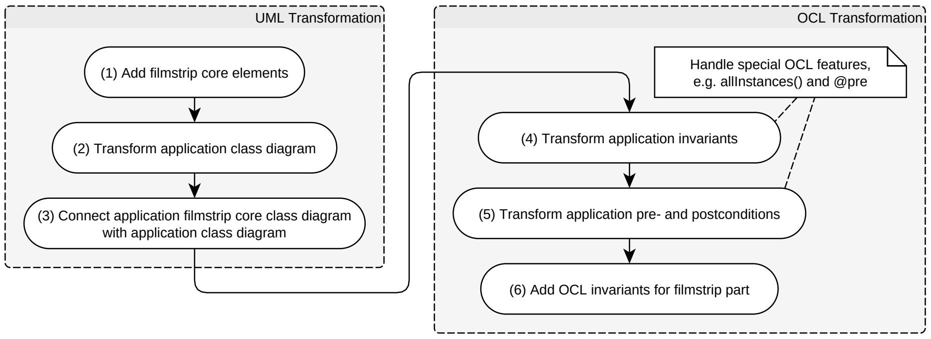
UML Transformation
(1) Add filmstrip core elements
(2) Transform application class diagram
(3) Connect application filmstrip core class diagram with application class diagram
OCL Transformation
Handle special OCL features, e.g. allInstances() and @pre
(4) Transform application invariants
(5) Transform application pre- and postconditions
(6) Add OCL invariants for filmstrip part

Fig. 4. Activity diagram of the filmstrip transformation process

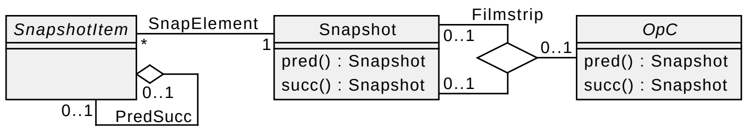

Fig. 5. Static elements of the filmstrip model which are added to the application model

as the filmstrip model shows a linear filmstrip and each snapshot has at most one predecessor and one successor. The intention of the class `SnapshotItem` is similar to the class `LocalSnapshot` from the OCL standard [15], however the handling of the ordering is different there.

Another core element is the abstract class `OpC`. This interface represents the operation calls that occur between two snapshots and is later extended with the specific information from the application model operations. This information includes the object that the operation is called on, the name of the operation, which is called and the parameters. The operations `pred()` and `succ()` of the classes `Snapshot` and `OpC` are query operations navigating to the predecessor snapshot or successor snapshot respectively, returning a single object instead of a `Set`, which the association end for this navigation indicates. For the class `OpC` these are the pre and post states of an operation call.

The next element of the filmstrip core is the abstract class `SnapshotItem`. It is an interface for the classes of the application model and lists functions that every class has to provide, so the filmstrip elements can work with them. The first functionality is specified by the association `SnapElement`, being a connection to the snapshot to assign objects to it. The second functionality is specified by the aggregation `PredSucc`, which connects two objects of the same type with each other. An aggregation is used to keep the connections cycle free. It defines the successors and predecessors of each object to easily navigate between different incarnations. These incarnations describe one object from the application model that can change its state during operation calls, whereas in the filmstrip model each incarnation is a new object in the object diagram. Without an explicit connection between these objects, another identifier would be necessary to navigate between incarnations, e.g. a key attribute. In contrast with the alternatives the association provides easier access, which is in particular useful when transforming the OCL expression `@pre`.

Finally, the ternary association `Filmstrip` connects two `Snapshot` objects and an `OpC` object, to represent the predecessor state and the successor state of an operation call. A ternary association is chosen to provide direct access between the objects and still keep a maximum level of compatibility. An alternative is replacing the abstract class `OpC` with an abstract association class between two `Snapshot` classes, which would make the query operations `pred()` and `succ()` unnecessary. For an even better compatibility, especially with validation and verification tools in mind, the ternary association `OpC` can be replaced with two binary associations. One leading from the `Snapshot` class to the `OpC` class and one association back to the `Snapshot` class. These associations can also be represented by aggregations or compositions to inherit their traits, i.e. cycle freeness. All options are interchangeable with minor differences in their usage which affects the transformation process. The constraints on the filmstrip association also need to be adapted. This work concentrates on the transformation using a ternary association, as shown in the class diagram in Fig. 5.

3.2 Application Model Classes

The next step in the transformation process handles the application model classes. These classes remain mostly the same, i.e. the name, attributes and operation definitions are kept. The classes are modified to inherit from the abstract class `SnapshotItem` to define a connection to the filmstrip core elements. Since the associations of the interface are defined on the abstract class `SnapshotItem`, the inherited type of the association ends is `SnapshotItem`. To replace these with the actual type of the transformed class, both associations are refined using the `redefines` keyword. With this UML feature, association ends can override other existing association ends of the class hierarchy, e.g. it is possible to specify a more precise end type for the navigation. The results are a type-safe access of the properties and another advantageous side-effect, which prevents links between objects of different types, e.g. between `Scheduler` and `Process`. In addition, the properties become well-defined even when using multiple inheritance. The refinement of the association `SnapElement` creates a property to access all objects of a specific class from the snapshot object, instead of all objects that inherit from `SnapshotItem`, which will be useful when transforming the OCL expression `allInstances()`. An example of a transformed class with all UML features visible is shown in Fig. 6. Association classes of the application model are included in this transformation step. These refinements were omitted in the class diagram in Fig. 2 for better clarity.

Lastly for every class that has operations with side-effects, a new abstract class inheriting from the abstract class `OpC` is created. This new class represents the base class for all concrete operation classes of this class and has an attribute `aSelf` of the type of the application model class to represent an object, that an operation call is invoked on. In the example from Sect. 2 this class is called `SchedulerOpC`.

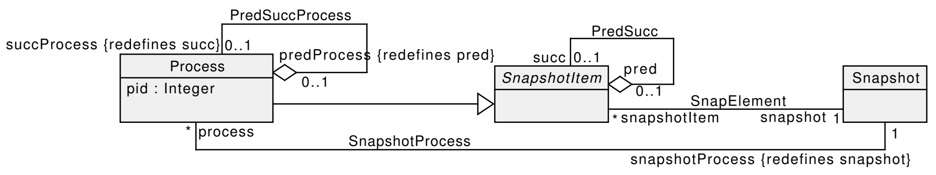

Fig. 6. Redefined association ends for class `Process`

3.3 Application Model Operations

The expressions of query operations remain with the application model class and can be used in the filmstrip model as well, since other elements probably depend on it. Only some OCL expressions, e.g. `allInstances()`, are transformed according to Sect. 4, since their effect is different in the filmstrip model.

Operations with side-effects are transformed differently. In the application model, the parameter values of these operation calls is only required at the time when the operation is invoked. On the contrary, in the filmstrip model, these operation calls are modelled statically with the class `OpC` and it is desired to validate the operation calls statically as well. Thus a new class is created for each operation with side-effects. It inherits the abstract operation call class of the operations owner class introduced earlier. The operation parameters are replicated as attributes of this class. The only variable left is `self` which is saved in the attribute `aSelf` inherited in the class (see class `SchedulerOpC` in Fig. 2). As a result, all variables required for the pre- and postconditions are provided by the concrete operation call class. These attribute values must point to the predecessor snapshot. Successor values can be accessed with the association `PredSucc`.

The pre- and postconditions are transformed into invariants and assigned to the concrete operation call class as well. The class is only instantiated when such operation call occurs. Therefore the invariants representing the operation pre- and postconditions only trigger once for every operation call invocation. This matches the exact behavior of the pre- and postconditions.

The OCL expressions of the pre- and postconditions need to be adjusted, when transforming them into invariants, because the variables inherently available in such expressions, e.g. `self` and parameters, have become class attributes. In addition, postconditions may contain unique expressions that are not available in invariants, i.e. `@pre` and `oclIsNew()`. These special expressions need to be transformed along with the other expressions that change their behavior in the filmstrip model. The details on these transformations are described in the next section. Finally the pre- and postconditions are removed from the operation as they are fully covered by the invariants and no longer needed.

4 OCL Transformation

In the filmstrip model, a clear separation exists between the filmstrip core elements and the application model elements. Therefore both parts are mostly functioning on their own after the transformation. As a result, most OCL expressions of the application model can be reused after the transformation. How the remaining aspects of the OCL elements are transformed, is described in this section.

4.1 Variables

Certain OCL expressions like operation pre- and postconditions have predefined variables, i.e. `self` and operation parameters, accessible in the expression. When transforming these pre- and postconditions into invariants, the parameters are lost. Also the variable `self` has a different value, since the OCL expression moves from the owner class of the operation to the operation call class of the filmstrip model. Thus each access of a variable is changed to point to the proper attribute of the operation call class. This includes the variable `self`, which is replaced with the expression `self.aSelf`. For postconditions, the values of the post state are required, which are accessed using the association `PredSucc`. For the application model invariants this is not necessary, as they remain at their corresponding class and the value of `self` does not change.

4.2 Expression Transformation

When transforming OCL expressions for the filmstrip model most of the OCL elements can be kept. As stated before, the expressions of the application model do not include filmstrip elements. Therefore, they will not use elements from outside of its originating snapshot.

However, the OCL expression `allInstances()` with its global property represents an exception to this rule. In the application model it is used to access all objects from one state, i.e. all objects in a single object diagram. In the filmstrip model a whole state is represented as one snapshot and multiple snapshots may be part of a single object diagram. Therefore the expression `allInstances()` needs a special treatment when being transformed. To replicate the functionality of the expression all objects of the requested type, that are assigned to one snapshot need to be accessed. The refinements of the association `SnapElement` are used for this task. To determine the correct snapshot the value of the variable `self` (`self.aSelf` for transformed pre- and postconditions) is used, again because the original expressions do not cross snapshots. As an example in the transformation process the OCL invariant expression `Process.allInstances()` becomes `self.snapshot.process`.

Other elements that need alternative representations in the filmstrip model are the expression `oclIsNew()` and the keyword `@pre`. `oclIsNew()` is a special expression only available in postconditions. Because the expressions of postconditions become invariants, the expression is unusable. It checks whether an object is created during an operation call, which equals to the statement: It was not existent in the predecessor state. In the filmstrip model this property is replicated by checking for the predecessor of the object using the association `PredSucc`.

```
context Scheduler::New( p:Process )
post: waiting = waiting@pre→including(p) and
      ready = ready@pre and active = active@pre
```

Fig. 7. Postcondition of the scheduler `New()` operation in the application model

```
context New_SchedulerOpC
inv: aSelf.succ.waiting = aSelf.waiting→collectNested( p1 |
         p1.succ )→asSet()→including( p.succ )
     and aSelf.succ.ready = aSelf.ready→collectNested( p1 |
         p1.succ )→asSet()
     and aSelf.succ.active = aSelf.active.succ
```

Fig. 8. Transformed postcondition of the scheduler `New()` operation

Thus an OCL expression `p.oclIsNew()`, where `p` is a process object, becomes `p.pred.oclIsUndefined()` in the filmstrip model.

The keyword `@pre` is also only available in postconditions. If an expression is postfixed by this keyword, the expression is evaluated in the pre state of the operation call. The keyword only affects one expression and can be used multiple times in an OCL query. In the filmstrip model the pre state is explicitly available for every operation call. To evaluate the expression in the filmstrip model, the association `PredSucc` is used. The expression switches to the predecessor snapshot of the current object, executes the postfixed expression and switches back to the original snapshot.

However there are a few pitfalls depending on the actual type of the current objects. Basic types, i.e. `Boolean`, `Real`, `Integer` and `String`, are stateless and do not need to be switched. For collection and tuple types the contents have to be switched to the predecessor state. Particularly collection types require caution, as the OCL operation `collectNested`, which is used to switch to the predecessor state, changes the type of collections from `Set` to `Bag` or from `OrderedSet` to `Sequence`. To counteract these changes, the OCL expressions `asSet()` and `asOrderedSet()` have to be added, to keep the original behavior. Also the types when switching to and from the predecessor state may differ, depending on the evaluated expression. Let `sched1` be a scheduler object of the filmstrip model, the expression `sched1.waiting@pre` in a postcondition to access the waiting processes in the pre state, is transformed into `sched1.pred.waiting→collectNested( succ )→asSet()`. Note the type of the evaluated value: it goes from `Scheduler` to `Set{Process}` to `Bag{Process}` (during the `collectNested()` evaluation) and back to `Set{Process}` again.

To give an example for the OCL transformations, Fig. 7 shows the postcondition of the operation `New` in the scheduler application model. After the transformation the expression has become an invariant of the operation call class, the variable access changes and the keyword `@pre` is transformed. The result is shown in Fig. 8.

```
context Snapshot inv cycleFree:                                           (a)
  Set{ self }→closure( s | s.succ() )→excludes( self )
context Snapshot inv oneFilmstrip:                                        (b)
  Snapshot.allInstances()→select( s |
    s.pred().oclIsUndefined() )→size() = 1
  and Snapshot.allInstances()→select( s |
    s.succ().oclIsUndefined() )→size() = 1
context OpC inv assocClassBehavior:                                       (c)
  self.pred()→size() = 1 and self.succ()→size() = 1
  and OpC.allInstances()→forAll( op |
    (self.pred() = op.pred() and self.succ() = op.succ()) implies
       self = op )
context SchedulerOpC inv aSelfDefined:                                    (d)
  not self.aSelf.oclIsUndefined()
context SchedulerOpC inv aSelfInPred:                                     (e)
  self.aSelf.snapshot = self.pred()
context New_SchedulerOpC inv paramPInPred:                                (f)
  not self.p.oclIsUndefined() implies
    self.p.snapshot = self.pred()
context Scheduler inv validSnapshotLinking:                               (g)
  not self.succ.oclIsUndefined() implies
    self.succ.snapshot = self.snapshot.succ()
context Scheduler inv validLinkingActive:                                 (h)
  not self.active.oclIsUndefined() implies
    self.snapshot = self.active.snapshot
context Scheduler inv validLinkingReady:                                  (i)
  self.ready→forAll( c | c.snapshot = self.snapshot )
```

Fig. 9. Various invariants of the filmstrip model to ensure correct usage and behavior

4.3 Filmstrip Model Constraints

To complete the filmstrip transformation, additional invariants are added to the resulting model, in order to force correct interaction of the filmstrip model elements and being able to reproduce the application model behavior correctly. In a first step three invariants are added to the filmstrip core elements. The definitions are shown in Fig. 9(a)–(c). The first invariant is called `cycleFree` and ensures, that the filmstrip line is free of cycles. The second invariant is called `oneFilmstrip` which prohibits the existence of more than one filmstrip per object diagram. And the last definition is called `assocClassBehavior` and makes sure that two snapshots are linked with at most one operation call.

The next invariants are applied to the operation call classes generated during the filmstrip transformation. These ensure correct values for the attributes. At first the attribute `aSelf` must be defined, since it is the object, the operation is called on. Furthermore it must point to an object in the pre state of the operation call, i.e. it must be assigned to the snapshot accessible by the query operation `pred()`. The definitions are shown in Fig. 9(d)–(e).

Additionally the attributes covering the operation parameters need similar constraints depending on the type of the attribute. For those attributes, the value must be in the pre state of the operation call, the same as the value of aSelf. Unlike the attribute aSelf the parameters may have undefined values. An example definition of such invariant for the parameter p of the operation New of the class Scheduler is shown in Fig. 9(f). Collection and tuple type attributes must be covered accordingly. Types other than classes of the application model, like String and Integer, are stateless as they cannot be assigned to a snapshot and therefore do not need restrictions. This includes enumerations.

Lastly, a few invariants are added to the classes from the application model. The first of these is called validSnapshotLinking and is added to all classes transformed from the application model. It assures, that links of the association PredSucc are only established between objects from consecutive snapshots in the right order. An example for the class Scheduler is shown in Fig 9(g). This invariant only checks objects, that have a successor instead of every object of a snapshot, because objects may be deleted during an operation call and therefore do not necessarily have a successor. Furthermore objects without a predecessor are created during the operation call leading to its snapshot.

The next invariants affect associations from the application model. To represent a single state from the application model, objects of one snapshot may only be linked with objects from the same snapshot. The invariant validLinking does this by comparing the snapshot objects of the association ends. Figures 9(h)–(i) show examples for 0..1 and * multiplicity association ends. N-ary associations and association classes must be covered accordingly.

5 Examples of Use

This section uses the object diagram from Fig. 3 to detail some of the benefits of the filmstrip model. The state in the object diagram demonstrates a sequence of operation calls of the transformed scheduler as modelled by the filmstrip model. The object diagram contains a total of four snapshots connected with three operation calls. Thick lines indicate elements from the application model (compare Fig. 2). The other objects and links are elements of the filmstrip model. The order of the operation invocation is from top to bottom.

The object diagram shows a whole process cycle (as introduced in Sect. 2) of the process with pid 1 starting from the Active state. Since all information is available in the object diagram and therefore accessible with OCL, it is possible to create an OCL query, which checks the order in which the process passes the states and whether it hits every state or leaves any of them out. Other queries can e.g. list Ready processes of certain snapshots:

```
Sequence{ s1, s2, s3, s4 }→collect( s | s.scheduler.ready )
→ Sequence{ Set{p2}, Set{}, Set{p5}, Set{} }
    : Sequence(Set(Process))
```

The query lists all Ready processes for every scheduler of the given input snapshots. Compared to this example, starting from the snapshot objects, it is also possible to find snapshots, in which a given process is ready next:

```
Set{ p1, p2 }→collect( p | Tuple{ idProcess=p,
    snapshots=Set{p}→closure( succ )→select( pp |
      pp.schedulerReady <> null } )

→ Set{ Tuple{ idProcess=p1, snapshots=Set{s3} },
        Tuple{ idProcess=p2, snapshots=Set{} } }
    : Set(Tuple( idProcess:Process, snapshots:Set(Snapshot) ))
```

This query uses the order of the objects given by the association PredSucc to find all future incarnations of the processes and selects those snapshots, where the process is in the Ready state. A better overview of the resulting map is given in the following table:

idProcess	p1	p2
snapshots	Set{s3}	Set{}

The result shows that in this particular operation sequence the process p1 is next Ready only in snapshot s3, whereas the process p2 is never Ready again after the first snapshot.

Further test objectives include, whether an operation sequence exists, so that a certain process is never scheduled or if deadlocks exist in the system. These objectives can be expressed with OCL invariants. Some constraints can be expressed easier using OCL extended with linear temporal logic (LTL) [18]. For example, the temporal expression

$$\text{Unassigned} \wedge (\text{Unassigned } \textit{until} \text{ Waiting})$$

on processes asserts that every process begins in the unassigned state and remains in that state until it finally gets into the waiting state after being assigned to a scheduler. In the filmstrip model this property is expressible with plain OCL. For the resulting expression it does not matter how many processes are part of the system state and in what order they are processed. Another test scenario is the reachability of a certain state from a given start state, which is done by constraining the last snapshot to the desired final state.

The example system state of the filmstrip model is built up without actually invoking any operation call. Those are only modelled using the new elements of the filmstrip model. All invariants of the transformed model are fulfilled by the system state. Extracting object diagrams, as they appear in the application model, can be done by removing all but the elements with a thick border in the state. Four distinct states remain that equal to four object diagrams from the application model. Sequence diagrams can be extracted by looking at the operation call objects and comparing the two snapshots linked with it. The delta between the objects linked to the snapshots are the actions taken place during the operation call.

Since no operation needs to be invoked to create these system states, verification and validation tools for UML and OCL models without support for model dynamics can be utilized to generate system states like the one in Fig 3. When enriching the model with further constraints, e.g. test objectives using temporal logic, those tools can be used to verify such dynamic properties in a bounded environment.

6 Implementation

The whole transformation process is implemented in Java as a plugin for the USE tool [11]. This particular transformation is intended to be an integral part of our validation and verification framework and therefore we have decided to implement the transformation in Java. Alternatively, we could provide a programming language-independent formulation in transformation languages like QVT-R or ATL.

The implementation follows the definitions of the transformation in this paper closely and has a high compatibility to different models. The plugin uses the setup described in this paper and therefore does not require a configuration. It transforms all UML and OCL features supported by USE and is compatible to all models, loadable in USE. The transformation is a linear process, which results in fast transformation times[1]. The plugin is available for download on the USE website[2].

7 Conclusion and Future Work

We have provided a widely applicable and automated way of transforming UML and OCL application models into filmstrip models and presented a fully functional implementation on the basis of the USE tool. Transformed models can, however, also be processed and validated with other tools. The approach forms a baseline for further verification and validation processes on the filmstrip model by being compatible to a maximum number of application models with only one generic transformation. Basic ideas of test objectives have been provided.

Future work should study automated test generation on the basis of the filmstrip model. Furthermore, the verification times of the approach needs to be analysed in a detailed case study. Improvements to the filmstrip model include compatibility to nested operation calls to allows for an even wider range of application models to be transformed and more detailed tests on them. Another field of study is the comparability of snapshot objects and the possibility to allow multiple operation calls from one snapshot. This introduces the reusability of snapshots to create snapshot graphs instead of linear filmstrips, similar to Kripke structures. Thus getting closer to the specification of CTL formulas instead of LTL formulas.

Acknowledgements

Thanks to the reviewers and Dániel Varró for their constructive comments.

References

1. Al-Lail, M., Abdunabi, R., France, R.B., Ray, I.: Rigorous Analysis of Temporal Access Control Properties in Mobile Systems. In: ICECCS. IEEE (2013)

[1] The scheduler example is transformed in less than a second.
[2] `http://sourceforge.net/projects/useocl/`

2. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On Challenges of Model Transformation from UML to Alloy. Software and System Modeling 9(1), 69–86 (2010)
3. Bill, R., Gabmeyer, S., Kaufmann, P., Seidl, M.: OCL meets CTL: Towards CTL-Extended OCL Model Checking. In: Proceedings of the MODELS 2013 OCL Workshop. vol. 1092, pp. 13–22 (2013)
4. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models using Constraint Programming. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) ASE 2007. pp. 547–548. ACM (2007)
5. Clark, T.: Model Based Functional Testing using Pattern Directed Filmstrips. In: Proceedings of the 4th International Workshop on the Automation of Software Test. pp. 53–61. IEEE (2009)
6. Conrad, S., Turowski, K.: Temporal OCL Meeting Specification Demands for Business Components. In: Unified Modeling Language: Systems Analysis, Design and Development Issues, pp. 151–165. IGI Publishing (2001)
7. Dick, J., Faivre, A.: Automating the generation and sequencing of test cases from model-based specifications. In: Woodcock, J.C., Larsen, P.G. (eds.) FME '93: Industrial-Strength Formal Methods, Lecture Notes in Computer Science, vol. 670, pp. 268–284. Springer Berlin Heidelberg (1993)
8. D'Souza, D., Wills, A.: Catalysis. Practical Rigor and Refinement: Extending OMT, Fusion, and Objectory. Tech. rep., http://catalysis.org (1995)
9. Flake, S., Müller, W.: Past- and Future-Oriented Time-Bounded Temporal Properties with OCL. In: SEFM 2004. pp. 154–163. IEEE Computer Society (2004)
10. Gil, J., Kent, S.: Three Dimensional Software Modeling. In: Torii, K., Futatsugi, K., Kemmerer, R.A. (eds.) ICSE 1998. pp. 105–114. IEEE Computer Society (1998)
11. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69 (2007)
12. Gogolla, M., Hamann, L., Hilken, F., Kuhlmann, M., France, R.B.: From Application Models to Filmstrip Models: An Approach to Automatic Validation of Model Dynamics. In: Fill, H.G., Karagiannis, D., Reimer, U. (eds.) Modellierung (2014)
13. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge, Massachusetts (2006)
14. Kanso, B., Taha, S.: Temporal Constraint Support for OCL. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. pp. 83–103. LNCS 7745, Springer (2012)
15. OMG (ed.): Object Constraint Language, Version 2.3.1. OMG (2012), OMG Document, `www.omg.org`
16. Salas, P.A.P., Aichernig, B.K.: Automatic Test Case Generation for OCL: a Mutation Approach. Tech. Rep. 321, The United Nations University – International Institute for Software Technology (2005)
17. Snook, C.F., Butler, M.J.: UML-B: A Plug-in for the Event-B Tool Set. In: Börger, E., Butler, M.J., Bowen, J.P., Boca, P. (eds.) ABZ. Lecture Notes in Computer Science, vol. 5238. Springer (2008)
18. Soden, M., Eichler, H.: Temporal Extensions of OCL Revisited. In: Paige, R., Hartman, A., Rensink, A. (eds.) Model Driven Architecture - Foundations and Applications, Lecture Notes in Computer Science, vol. 5562, pp. 190–205. Springer Berlin Heidelberg (2009), `http://dx.doi.org/10.1007/978-3-642-02674-4_14`
19. Yu, L., France, R.B., Ray, I.: Scenario-Based Static Analysis of UML Class Models. In: Model Driven Engineering Languages and Systems. Springer (2008)
20. Ziemann, P., Gogolla, M.: OCL Extended with Temporal Logic. In: Broy, M., Zamulin, A. (eds.) 5th Int. Conf. Perspectives of System Informatics (PSI'2003). pp. 351–357. Springer, Berlin, LNCS 2890 (2003)