Verifying Linear Temporal Logic Properties in UML/OCL Class Diagrams using Filmstripping

Abstract—**Testing system behavior in real world applications often requires analyzing properties over multiple system states to ensure that operations do not interfere with each other in ways that are not desired. In UML class diagrams, behavior is specified using operations with pre- and postconditions, which alone are not sufficient to formulate temporal properties spanning multiple system states and, thus, require additional description means. For this purpose, multiple extensions of OCL with linear temporal logic (LTL) exist, which provide a formalism to describe temporal properties. Using so-called filmstrip models, this paper provides formal semantics for OCL enhanced by LTL through a translation into standard OCL on the basis of class diagrams and enables verifying these temporal properties using existing model checking tools.**

I. Introduction

Nowadays, Model-Driven Engineering (MDE) is considered a promising approach for developing systems. Models, as abstract higher level representations, are used to formally describe hard- and software system structure and behavior. Having such models of the system, which can be simulated and checked for defects before any implementation is done, is invaluable. For example in the hardware/softare co-design in embedded systems, systems and their interactions are first modelled on a high abstraction level before they are devided into hard- and software components [1], [2]. Processes represented by active circuits can be checked for safeness and liveness properties already in the model without requiring an implementation.

Many development methods and tools employ languages as the Unified Modeling Language (UML), the UML profile SysML and the Object Constraint Language (OCL). UML is a general purpose language allowing to model hard- and software systems alike in an abstract fashion. The level of abstraction is chosen by the modeler. However, UML and OCL provide currently only limited support for the descriptive formulation of temporal properties such as safety or liveness properties. Therefore, temporal logic extensions of OCL have been proposed and tool support is beginning to become available. However, these tools often work by enhancing an OCL evaluator with temporal operators that handles the temporal logic formulas.

This contribution studies a new way of dealing with temporal OCL: Temporal OCL is translated into standard OCL together with an accompanying transformation of the underlying class model. The resulting model is a plain UML and OCL model maintaining the same temporal properties and that, in principle, can be checked with any existing UML and OCL tool compatible to plain UML and OCL [3], [4], [5]. Our work is done within the context of the tool UML Specification Environment (USE) [6] that supports central UML diagrams, OCL and model validation and verification in form of a model validator employing relational logic [7]. The approach is based on a transformation of a UML and OCL application model into a so-called filmstrip model [8] that captures system behaviour through explicit description of system state sequences and transitions. On this basis, temporal properties can be expressed and evaluated directly in standard OCL without temporal extensions and is therefore compatible with a wide range of tools.

The rest of this contribution is organized as follows. Section II starts by introducing a running example for a filmstrip model and the basic ideas in temporal logic. Section III explains the applications of linear temporal logic (LTL) in behavioral models. The central Sect. IV puts forward the translation from LTL to standard OCL. Section V shows how temporal properties can be verified employing bounded model checking. After discussing related approaches in Sect. VI, the paper ends with concluding remarks and future work.

II. Background

A. Running Example in UML and OCL

The running example for our approach is a simple scheduler model [9]. It is pictured in Fig. 1 highlighted in the gray box, consisting of two classes `Scheduler` and `Process` and three associations `Waiting`, `Ready` and `Active`. It models the scheduling cycle of processes using two sets *Waiting* and *Ready* and one currently scheduled process *Active*. The scheduler registers and switches processes as defined in its operation's behavior. This source model is called the *application model* in contrast to the *filmstrip model* that results from the transformation, pictured by the entirety of Fig. 1.

Additionally to the structure, OCL invariants are employed to express further constraints that cannot be expressed by UML alone. In particular the processes must have unique identifiers and cannot be connected more than once. This prevents it from being connected to multiple schedulers at once as well as being in multiple queues of one scheduler. The OCL is a constraint language that can be used to query arbitrary information in the system. This information can either be evaluated directly on a given system state, i.e. some schedulers and their linked processes, or be employed as OCL invariants, i.e. contracts

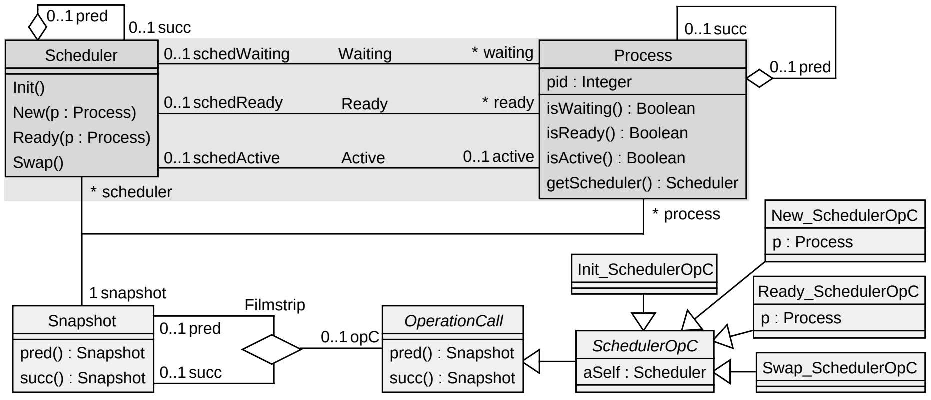

Figure 1. Scheduler filmstrip model.

that every system state have to comply with. Additionally, OCL is used to specify the pre- and postcondition operation contracts that are evaluated before and after the operation call, respectively. However, a limitation is that OCL does not provide description means to formulate arbitrary temporal properties over multiple system states. While postconditions have access to the state before and after the operation call and can enforce properties in the latter, they cannot make sure that these properties remain for a while. It is the task of the modeler to specify the model in a way that no unwanted interactions occur, which is a complex task. Allowing the modeler to use temporal logic in OCL allows them to increase the reach of his expressions.

The `Scheduler` class in the application model has four operations defined by declarative contracts in the form of pre- and postconditions formulated in OCL. These operations: (1) setup the object after its instantiation (`Init()`); (2) allow registering processes to a scheduler, which are then initially added to the waiting queue, represented by the association `Waiting` (`New(p:Process)`); (3) switch a registered process from the waiting queue into the ready queue, marking it ready to be scheduled (`Ready(p:Process)`); and (4) change the active process, giving the resources to a random, ready process and putting the prior active process back in the waiting queue (`Swap()`). All operations on the `Process` class are query operations allowing easy access to certain information, i.e. whether the process is in a particular queue or active and the scheduler instance, if the process is linked to one.

B. Filmstripping

The filmstrip transformation [8] takes the application model and adds components to explicitly model a trace of system states (snapshots) by grouping instances of the application model classes from one system state to the same `Snapshot` object. In addition, the temporary information during an operation call is also made explicit in the type hierarchy starting with `OperationCall`, saving the context of the operation (attribute `aSelf` in `SchedulerOpC`) and potential operation parameters, respectively. The pre- and postconditions of the application model are transformed into invariants in the filmstrip model and attached to the respective operation call class controlling its preceding and succeeding snapshots.

The actual filmstrip is the idea of having several snapshots connected by operation calls in a line representing an execution trace of the application model in a single object diagram, where all information is accessible with OCL. To further increase the accessibility, associations are generated for each application model class to navigate to its predecessor or successor state, respectively, also providing an identification which objects represent the same instances in different snapshots of the execution. The application model remains unchanged in the transformation process, except for the removal of pre- and postconditions and minor changes to the invariants to adapt the new structure. The behavior semantics of both models are identical and single system states of the application model can be extracted from the filmstrip model at any time by focusing the view on a single snapshot.

C. Linear Temporal Logic

The *linear temporal logic* (LTL) [10] allows to create logic expressions involving statements about discrete time. In the case of filmstrip models, states are defined by snapshots and transitions between them are represented by operation calls. Thus, enhancing OCL with LTL allows to formulate expressions about future system states, e.g. *finally ϕ holds* with ϕ being an arbitrary OCL expression. LTL introduces various boolean operators that take arbitrary boolean expressions and evaluates them in the respective time, which is given by the chosen operator.

LTL is usually used in system specifications to formulate safety properties of the form: *something 'bad' never happens* usually derived from the system requirements.[1] Negating such properties leads to LTL expressions stating that eventually something bad will happen, which can be tested against finite traces of system executions to find counterexamples for the original safety property and therefore defects in the model.

Figure 2 gives a quick overview of the semantics of four LTL operators. It shows example execution traces, e.g. system states and operation executions of a UML/OCL model, and the evaluation of different LTL operators on them. Each bullet represents a system state and the symbols ϕ and ψ below them represent an arbitrary OCL expression that is either satisfied in the system state (symbol present) or not (symbol absent). Above the states are LTL expressions that are evaluated on that state consisting of exactly one LTL operator each in combination

[1]Please note, this work concentrates on the translation of temporal logic into OCL rather than how to identify appropriate safety properties in models.

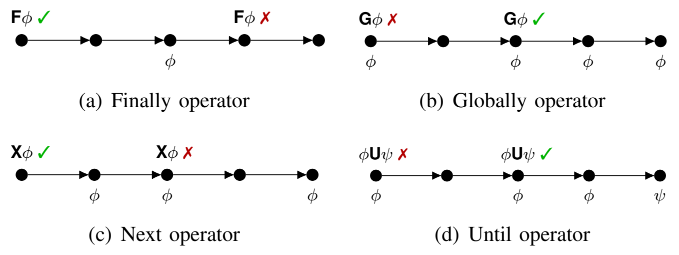

Figure 2. Example traces and evaluations for various LTL operators.

with the expressions ϕ and ψ. A checkmark (✓) next to the LTL expression denotes that the expression is satisfied, i.e. evaluates `true`, and a cross (✗) is displayed if it is not satisfied.

The first operator in Fig. 2(a) is the *Finally* operator (symbol $\mathbf{F}$ or $\Diamond$). It is satisfied if the expression following it is satisfied in the state the finally operator is checked against or any of its successor states. The second operator is the *Globally* operator (symbol $\mathbf{G}$ or $\Box$) shown in Fig. 2(b). It is satisfied if the expression following it is fulfilled in all successor states starting at the one the operator is checked against. The third operator in Fig. 2(c) is the *Next* operator (symbol $\mathbf{X}$ or $\bigcirc$). It is satisfied if the immediately following system state satisfies the expression. The last operator in Fig. 2(d) is the *Until* operator (symbol $\mathbf{U}$). This operator takes two expressions ϕ and ψ and requires that the first expression ϕ holds in every system state starting from the state the expression is checked against until the second expression ψ is satisfied at least once. The operator can only be satisfied when the second expression ψ holds at least once, which may be in the original system state, in which case the first expression ϕ does not need to be satisfied at all.

The last two operators will only be described by their equivalent representations using to the operators above. The operator *Weak-Until* (symbol $\mathbf{W}$) has the equivalent representation $\phi\mathbf{W}\psi \equiv \mathbf{G}\phi \vee \phi\mathbf{U}\psi$, which weakens the property of the Until operator insofar that ψ does not ever have to be satisfied to fulfill the Weak-Until operator. Additionally, the *Release* operator (symbol $\mathbf{R}$) is defined as $\phi\mathbf{R}\psi \equiv \psi\mathbf{W}(\phi \wedge \psi)$. It can be read as "$\phi$ releases ψ", i.e. ψ must hold until ϕ holds. In contrast to Until, this operator requires that ϕ and ψ are satisfied in one system state to evaluate the operator to `true`. Last but not least, the operators Finally and Globally can be equivalently represented as $\mathbf{F}\phi \equiv \mathtt{true}\mathbf{U}\phi$ and $\mathbf{G}\phi \equiv \neg\mathbf{F}\neg\phi$, respectively.

III. Applications of LTL in Behavioral Models

In standard UML class diagrams enhanced with OCL, the features to specify behavior are limited. Invariants are globally checked in every system state, but their context is always limited to the currently checked state. Therefore, without the model saving values for several system states, invariants cannot access information from other system states. Operation preconditions have similar properties and can only check properties in the system state immediately preceding the operation execution. Operation postconditions are the only expressions that can natively access two system states using the keyword `@pre`. However, they are still bound to their operation and therefore cannot check if different operations interfere with each other.

A solution to the problem is the introduction of LTL, which allows all invariants, pre- and postconditions to express conditions that range over multiple system states. In addition, temporal properties can be formulated as a query that are evaluated on a particular system state directly. An example of the possibilities of LTL in postconditions can be shown with the `Ready` operation of the scheduler. Adding a postcondition, a non-starvation property can be added, which requires the now ready process to eventually be the active process.

```
context Scheduler::Ready(p:Process) post:
  F p.isActive()
```

Other use case examples are ensuring a property *globally* or have preconditions check properties multiple system states back.

The evaluation of such temporal properties can pose a problem. The postcondition above cannot be fulfilled in the system state after the operation call and thus leaves the system in a possibly invalid state. Thus, interactively checking the LTL expressions during the system execution to utilize them at runtime needs a thorough definition, when OCL expressions containing temporal expressions have to be satisfied. This is provided by the temporal extension approaches that support validation during runtime. On the other side, evaluating the temporal expressions against a given finite trace does not have this problem. Having the complete trace, it is clear whether it satisfies a temporal expression or not.

When verifying temporal properties using bounded model checking, the goal is to find finite subtraces that violate a given safety property. In case of the filmstrip model, the goal is to generate a sequence of snapshots that satisfies all model invariants, pre- and postconditions and one or more additional temporal expressions. This allows to search for counterexamples in the model, i.e. a system trace that violates the property. For the scheduler running example, there are some temporal properties to test the model for:

1) **noStarvation** A desirable property of a scheduler is that processes that are ready to be scheduled eventually get the resources, i.e. become active. In a LTL formula it can be required that whenever a process is ready, it must eventually be active. As an invariant, this property can be expressed with the following expression:

```
context Process inv noStarvation:
G(self.isReady() implies
    F self.isActive())
```

2) **readySkipping** During the swap operation, a random process is selected from the ready queue. Thus, in order to allow a fair scheduling, all processes must be in the ready queue to eventually become active. An important property is to make sure that no process can overcome this order. Therefore, the next LTL expression checks, if a process can jump from the waiting status to being the active process, skipping the ready state. In contrast to the previous invariant, this one describes an erroneous system behavior.

```
context Process inv readySkipping:
F(self.getScheduler() <> null and
    (self.isWaiting() U self.isActive()))
```

To give an example of the evaluation of such LTL expression in the filmstrip model consider the example system execution

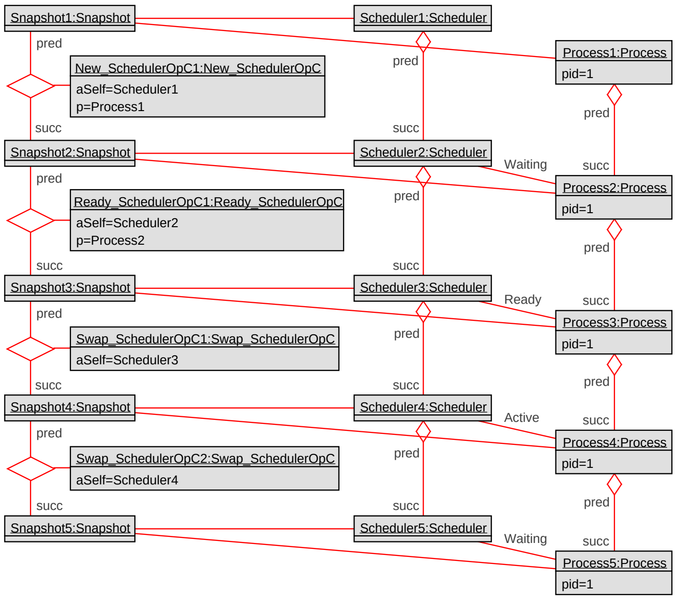

Figure 3. Example system execution trace.

shown in Fig. 3 represented as an instantiation of the filmstrip model. The trace starts at the top and each snapshot represents a system state in between which operation calls are executed. The roles `pred` and `succ` allow a free navigation in between the snapshots from any point. The first example property is a constraint with the context `Process`. The Globally operator requires the evaluation of the following expression on all system states with the variable `self` being the current process instance in each snapshot (`Process1` to `Process5`). The premise of the constraint (`self.isReady()`) is satisfied in `Snapshot3`, therefore the evaluation of the Finally operator starts in this snapshot. From here, any of the successor snapshots have to satisfy the expression after the Finally operator (`self.isActive()`). This is the case in `Snapshot4`, thus the Finally operator is satisfied for this process instance. Since there are no further snapshots where the premise holds, the constraint evaluates to `true`.

IV. LTL to Standard OCL Translation

In this section, the translation of LTL operators into standard OCL on the basis of the filmstrip model is presented. To keep the expressions small and clear, first some definitions are made that help define the translations. The approach allows the specification of LTL formulas on the application model which are then automatically translated into standard OCL based on the filmstrip model during the filmstrip transformation.

All LTL formulas operate on the trace of snapshots, starting from a specific snapshot, which can be seen as the starting point for the evaluation. Thus, in order to evaluate temporal expressions, access to the current snapshot is required.

Definition 1 (Accessing the current snapshot): We define the query operation `snapshot()` to simplify the access to the current snapshot for the evaluation of the LTL formulas. There are two cases where this access is required. First, at the start of an evaluation, the snapshot is determined by the current context, e.g. defined by the variable `self` or other contextual important variables. From there on, the snapshot is easily obtained in the filmstrip model using the associations `SnapshotItem` and `Filmstrip`. Instances of the class `OperationCall` always refer to the preceding snapshot. The definitions for the query operations on an application model class `T` and `Snapshot` are as follows:

```
class T
operations
 snapshot() : Snapshot= self.snapshot
```

```
class Snapshot
operations
 snapshot() : Snapshot= self
```

Second, for nested expressions, the snapshot is given by the context of the translation process which is always available. In the following, `snapshot()` is used in places, where the snapshot of the current context is required.

Once the current snapshot is determined, a recurring task is the navigation between them to check for the temporal properties. The query operations `succ()` and `pred()` can be used to get to the immediate successor or predecessor state, respectively. Using these operations in combination with the OCL operation `closure`, the whole trace of snapshots can be navigated. In addition to this general navigation, an operation is required to access subtraces, which can be achieved with the following definition.

Definition 2 (Extracting subtraces of snapshots): We define the query operations `succTo(Snapshot)` and `predTo(Snapshot)` on the `Snapshot` class as navigation operations with a defined end point given as a snapshot. The definitions in OCL are as follows:

```
class Snapshot
operations
 predTo( s : Snapshot ) : Snapshot=
  if self = s then self else self.pred() endif
 succTo ( s : Snapshot ) : Snapshot=
  if self = s then self else self.succ() endif
```

In order to extract the subtrace between two snapshots `S` and `S'`, the expression `S→closure( succTo(S'))` can be used, if snapshot `S'` comes after snapshot `S` in the trace.

Note that in order to extract a subtrace, the order of the two snapshots at the ends must be known, which is the case in all further translations and thus reduces the complexity of the overall expressions, by only collecting snapshots in one direction. Alternatively, it is possible to either determine whether a snapshot comes before or after another one when needed or a query operation can be defined that works both ways.

Note also that the operations can be realized without using recursion, which greatly helps with the compatibility of the translation to model checking tools.

Finally, after the translation of the LTL operators into the navigation to the corresponding snapshots, a simple representation whether these snapshots satisfy an arbitrary formula has proven useful. The problem here is that the expressions following LTL operators reference objects that might not be present in the snapshot they are tested against. In order to access the correct object that is within the snapshot, the operation `inState(Snapshot)` is used. It uses the roles `pred` and `succ`, which links object instances in different

Table I
TRANSLATION FOR VARIOUS LTL OPERATORS INTO STANDARD OCL.

Operator	Input	OCL Translation
Finally	**F**ϕ	snapshot()→**closure**(succ())→**excluding**(**null**)→**exists**(s | s.sat(ϕ))
Globally	**G**ϕ	snapshot()→**closure**(succ())→**excluding**(**null**)→**forAll**(s | s.sat(ϕ))
Next	**X**ϕ	snapshot().succ() <> **null and** snapshot().succ().sat(ϕ)
Until	ϕ**U**ψ	**let** state = snapshot() **in** state→**closure**(succ())→**excluding**(**null**)→**exists**(s | s.sat(ψ) **and** s→**closure**(predTo(state))→**excluding**(s)→**forAll**(s2 | s2.sat(ϕ)))

snapshots with each other to specify their predecessor and successor, respectively. The definition of the operation on an application model class `T` is as follows:

```
class T
operations
 inState( s : Snapshot ) : T=
  self→closure( succ )→union(
      self→closure( pred ) )→any( me |
      me.snapshot = s )
```

The operation is defined for all application model classes and is applied on every access of a variable either referencing an object or collection or tuple containing objects. In order to simplify the expressions in the translation, this procedure is hidden in an operation `sat()` (short for "satisfies") which hides the bulky OCL navigation details.

Definition 3 (Property satisfaction on snapshots): We define the operation `sat(Expression):Boolean` on the class `Snapshot` as a test, whether the snapshot satisfies a given expression, or not. The expressions can be arbitrary OCL expressions also containing further LTL operators. In order to test if a snapshot `S` satisfies a certain expression ϕ, we write `S.sat(` ϕ `)`, which implicitly applies the operation `inState` where required in the expression ϕ (within the translation) and evaluates to `true` if the snapshot satisfies the expression and `false` otherwise.

With this definition of the `sat` operation which automatically changes all evaluations into the contextually current snapshot, it seems impossible to compare values of different snapshots. However, this issue can be solved by using `let`-expressions to pre-evaluate subexpressions in the snapshot they shall be evaluated in. When such a variable defined by a `let`-expression is then referenced, the same rules from the `sat` operation apply, because in the filmstrip model, the same object in different snapshots has different identities, i.e. they are different objects in the filmstrip model system state and therefore cannot be compared directly. This only affects the result of the expression, not the evaluation of the expression itself, which makes it possible to compare values from different snapshots.

For example, to check if some process `p` is now registered to one scheduler and later was unregistered and then re-registered to another scheduler, the following OCL expression (*scheduler-Switching*) can be used:

```
F(let firstScheduler = p.getScheduler() in
firstScheduler <> null and
  F(p.getScheduler() <> null and
      p.getScheduler() <> firstScheduler))
```

First, the current scheduler is saved in the `let`-expression `firstScheduler`. Then it is checked whether this value is defined and therefore the process is connected to a scheduler. Also, there must eventually be a system state in which this process is registered again and the scheduler it is registered to must differ from the first scheduler. To allow this expression to start at any point when the process is actually registered to a scheduler, the whole expression is wrapped with a finally operator. If all conditions are fulfilled, the OCL expression is satisfied.

A final consideration regarding the translation, special to the filmstrip model, is required. So far, only temporal expressions on certain objects in a certain snapshot have been taken into account. This works for independent expressions that are checked against certain snapshots and pre- and postconditions, since for those expressions the context is known, either by the snapshot the expression is evaluated on or the pre- and post-states of the operation execution. When translating invariants containing LTL operators, the filmstrip model offers multiple translation possibilities, because every class instance has several representations in the form of one object in each snapshot. Regular class invariants have to be checked on every one of those objects to match the semantics for class invariants in the application model. However, this is impractical for temporal expressions. On the last snapshot, for example, simple expressions like **X**ϕ are not satisfied, unless properly protected, e.g. using implications. For an invariant expression ϕ on the class `Process`, we propose the following translation:

context `Process` **inv: self**`.pred =` **null implies** ϕ

The expression ϕ is evaluated for every class instance in the snapshot it is created in, i.e. the earliest possible snapshot the instance is available. This creates semantics that is consistent with the other use cases, more familiar to LTL experts and thus, in general less confusing. For properties that shall be satisfied in every single system state, the Globally operator can be used.

A. Translations

Using the definitions above, this section shows the translation of the LTL operators into standard OCL on the basis of the filmstrip model structure. The filmstrip model is used to represent the traces of system executions on which the temporal operators navigate between system states. The expressions, which are created for the application model, are then evaluated on the actual application model objects representing that system state.

In LTL, only the two operators *Next* and *Until* are required to express all other operators, however, direct translations for other operators result in smaller and thus more efficient OCL expressions. In the following, the four LTL operators from Sect. II are translated into OCL. For the following expressions, keep in mind that the OCL collection operation `closure` describes the *reflexive* transitive closure, i.e. the starting object is included in the result. Table I shows the general translation of the LTL operators in isolation using the definitions above.

Finally (Fϕ) For the Finally operator, any following snapshot has to satisfy the expression ϕ. In the filmstrip model this is evaluated by collecting all successor snapshots and try if any fulfills ϕ using an `exists` expression.

Globally (Gϕ) The Globally operator is translated similar to the Finally operator except that a `forAll` expression is used to evaluate that every following snapshot fulfills the expression ϕ.

Next (Xϕ) The expression for the Next operator first makes sure that a successor snapshot exists and then checks whether the successor snapshot actually satisfies ϕ. Without the check for a successor state, the evaluation of ϕ might result in `null` which causes undefined behavior.

Until (ϕUψ) For the Until operator, first the requirement that eventually a snapshot is reached that satisfies the expression ψ is checked in the lines 2–3 and continuing from that snapshot, all preceding ones back to the starting snapshot `state` have to satisfy the expression ϕ.
It is easy to detect the similarities to the Finally operator, which is expected due to the equivalence $\mathbf{F}\phi \equiv \texttt{true}\mathbf{U}\phi$. This example, resulting in a shorter OCL expression, shows the advantage of using direct translations for all operators.

Similar to the translations in Table I, the operators *Weak-Until* and *Release* are translated. In the following, the translations for two examples from earlier in this paper are presented. They are then verified in the next section.

Recall the example *schedulerSwitching* from earlier. As an invariant, the example is defined as follows:

```
context p : Process inv schedulerSwitching:
F(let firstScheduler = p.getScheduler() in
    firstScheduler <> null
  and F(p.getScheduler() <> null and
      p.getScheduler() <> firstScheduler))
```

After the translation into the filmstrip model the invariant becomes:

```
context p : Process inv schedulerSwitching:
p.pred = null implies
p.snapshot()→closure( succ() )→excluding(
    null )→exists( s |
  let firstScheduler =
      p.inState(s).getScheduler() in
  firstScheduler <> null and s→closure(
      succ() )→excluding( null )→exists( s2
      | p.inState(s2).getScheduler() <> null
      and p.inState(s2).getScheduler() <>
      firstScheduler.inState(s2) ) )
```

First, the actual expression is only evaluated on freshly created objects, i.e. objects that do not have a predecessor. Then the Finally operator is translated using the snapshot of the process `p` as the starting point. Within the operator, all variable accesses are mapped to the snapshot `s` determined by the Finally operator. Also the second Finally operator is based on the snapshot `s` and the value of the let expression is mapped as well.

Note the difference between the two subexpressions in the inner closure: `p.inState(s2).getScheduler()` and `firstScheduler.inState(s2)`. The latter expression expands to `p.inState(s).getScheduler().inState(s2)` evaluating the `getScheduler()` operation in the system state `s` (inside the first Finally operator) whereas in the former expression it is evaluated in the snapshot `s2`. In the end, the comparison (<>) of the two values happens in a single snapshot `s2`, otherwise the unequal operator would always evaluate to `true`.

The transformation process works on the abstract syntax tree (AST) of OCL enhanced with temporal operators. These are iteratively translated following the translations in this section, resulting in plain standard OCL expressions formulating the same semantics as the temporal operators. The transformation of the application model into the filmstrip model is automatically achieved by the transformation process as well [8].

V. Verifying Properties using the USE Model Validator

With the global view on the execution trace given by the filmstrip model, existing verification tools for class diagrams [3], [4], [5], [11] can be used to generate execution traces for arbitrary UML class diagrams enhanced with OCL. Together with the translation of LTL into OCL, these tools can find system executions where certain LTL expressions are fulfilled, or if there is none, the tool verifies that within the given bounds, no trace exists that can satisfy the expression. This is useful to find defects by searching for counterexamples for the safety properties of the system. In this section, the USE model validator [11] is used to show these use cases.

The USE model validator takes a UML/OCL class diagram together with problem bounds and generates a valid system state, if existent. The bounds specify the range of minimum to maximum number of instances for every class and association. In addition, they define the values of the basic types of the OCL, i.e. Integer and String. These bounds define a search space for a SAT solver to find an assignment that satisfies all explicit (OCL constraints) and implicit (multiplicities, composition, etc.) constraints of the model. Further constraints, for example an LTL property, can be added as OCL expressions to the solving process, as well.

Example 1 (Finding a Counterexample): In the scheduler running example, it is not meant for a scheduler to unregister its processes once it is running and there is no operation to do so. To search for a counterexample of this property, the invariant `schedulerSwitching` is used, which requires processes to eventually be registered with a scheduler and later re-registered with another scheduler.

```
context p : Process inv schedulerSwitching:
F(let firstScheduler = p.getScheduler() in
firstScheduler <> null and
  F(p.getScheduler() <> null and
      p.getScheduler() <> firstScheduler))
```

We expect the verification engine to not find a system execution, because this behavior is not intended in the model. However, when giving the model together with the LTL formula and problem bounds to the USE model validator, the outcome is 'satisfiable' and the trace shown in Fig. 4 is generated.

The system's initial state is at the top. It starts in a state with two schedulers and one process that is registered with one of the schedulers and is currently in the waiting queue. From here it is possible to execute the operation `Init` on

```
Searching solution with
SatSolver 'MiniSat' and
bitwidth 8...

Outcome: SATISFIABLE;
Solving time: 387 ms
```

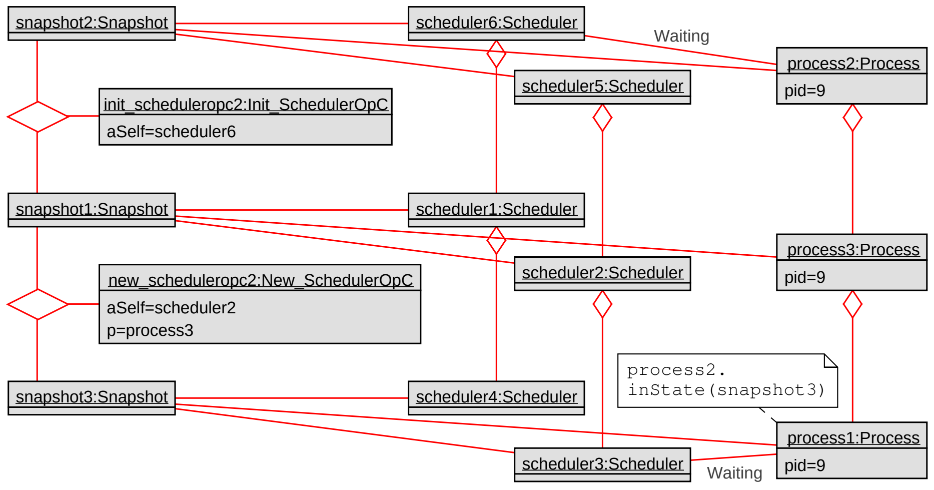

Figure 4. Counterexample showing a process switching schedulers and tool output.

the scheduler that already has processes registered to it. The post condition of the initialization operation ensures that the scheduler has no connected processes, thus unregistering the process. The next operation then re-registers the process to a different scheduler and the LTL expression is satisfied, violating the original property.

The pictured behavior, revealed here by the model validator, is a defect in the model. Certainly, the initialization operation is meant as a one-time initialization of a scheduler object after its creation. However, there is no mechanism in place to prevent the operation being called later. In fact, the operation has no precondition at all. Correcting the behavior is easily achievable by adding a boolean attribute to the scheduler class, or even better utilizing a protocol state machine to enforce a certain execution order.

Example 2 (Verification of Property within Bounds): After the defect from the first example is corrected, a second property is tested. Recall the property `readySkipping`:

```
context Process inv readySkipping:
F(self.getScheduler() <> null and
    (self.isWaiting() U self.isActive()))
```

When a process is registered to a scheduler it is in the waiting queue first. It is now tested, whether the process can become the active process from the waiting queue directly, thereby skipping the ready queue, possibly ignoring the scheduling policies of the scheduler. The problem bounds allow one scheduler with three processes and five operation executions. Figure 3 shows that it takes three operation executions for a process to complete a whole cycle from the waiting queue to becoming the active scheduler and getting back in the waiting queue. Therefore, five operation executions are initially chosen to allow at least one whole cycle plus potential initializations and actions from other processes. Starting the model validator with this LTL-expression and the new bounds, the outcome is 'unsatisfiable', as shown in Fig. 5. The outcome 'unsatisfiable' states that within the complete search space (defined by the problem bounds) there is no valid assignment to satisfy the LTL formula in the given model. Now this does not verify that a process can never skip the ready state. It just excludes it within the stated bounds. This is a common situation when verifying LTL using bounded

```
Searching solution with SatSolver 'MiniSat'
    and bitwidth 8...
Outcome: UNSATISFIABLE; Solving time: 92097 ms
```

Figure 5. Result for the `correctOrder` example.

model checking and there are multiple ways to go on:

- Iteratively increasing the problem bounds might eventually make the search space big enough to find a counterexample that was not instantiable before. This is a good procedure when a defect is expected to be unveiled by a property.
- Reason that the investigated search space covers all unique system executions with regard to the property and model and thus, enlarging the search space would only lead to bigger and/or longer traces repeating the same actions over and over.
- Search for execution loops in the system in which the property is not satisfied by generating a trace containing any snapshot twice. For instance, having a property $\mathbf{F}\phi$, such loops represent an infinitely long execution in which the property will never be satisfied and therefore, are a counterexample to the property.

The runtime of the SAT solver for the second example is just over 92 seconds. Additionally to the runtime of the SAT solver (solving time), the times for the model transformation into the filmstrip model including the LTL operators, the translations from UML/OCL into relational logic and the further translation into SAT need to be taken into account. However, in comparison to the solving time, these times are often negligible. In this example the combined total of the non-solving times was less than a second.

VI. Related Work

A number of extensions of OCL allow for temporal operators. A nice detailed comparison can be found in [12]. One of the earliest works [13] concentrates on temporal business rules without giving a full semantic definition for temporal features. [14] defines the classical LTL operators without going into a possible implementation. [15] focuses on the integration of time bounds in connection with temporal constructs. [16] revisits the work from [14] and defines temporal OCL operators that are intended to be used for more general metamodels than UML-like ones. [12] sketches an implementation of temporal OCL on the basis of Eclipse MDT/OCL.

More recently and in contrast to the more frequently applied LTL approach, CTL (Computation Tree Logic) is the basis of the logic in [17]. Concerning the execution part that work is grounded on graph transformation rules. OCLR [18] does not take LTL or CTL as a basis, but the approach aims at practitioners and tries to hide mathematical details, creates natural language like expressions and is based on frequently used property patterns that need to be expressed in applications. Similar to our approach, [19] is rested upon snapshots, however in order to trace object identities, identifying attributes (in the spirit of relational key attributes) are employed in contrast to direct associations as in our approach. Furthermore, only formula patterns are presented instead of a general translation of temporal logic into OCL. Finally, the details how the temporal logic can be employed in the model for simulation purposes is not discussed. Providing a general integration of temporal logic for the filmstripping approach, enhances existing work based on filmstripping (e.g. [20]), opening more possibilities. In [21] and [22], temporal logic is checked directly against models formulated in relational logic. Our approach also relies on relational logic due to the use of the model validator, but the translation of the temporal logic is lifted to the OCL layer to give modelers a more familiar environment.

In contrast to the mentioned approaches, our work for temporal OCL relies on an explicit filmstrip model that provides the necessary semantic temporal evaluation structure. Another difference to the mentioned works is that our temporal expressions are translated into plain standard OCL expressions and are therefore, compatible with any (decently capable) OCL evaluator.

VII. Conclusion and Future Work

We have presented a translation for LTL enhanced OCL that replaces LTL operators with equivalent OCL expressions on the basis of the filmstrip model, enabling the use of regular UML and OCL verification engines to find model instances that satisfy certain LTL expressions. On this basis, we have presented examples for LTL properties in a scheduler running example and illustrated how to find defects in the model and verify the LTL properties by searching for counterexamples.

Future work can focus on extending the filmstripping approach to allow not only one but multiple successor states after an operation call in order to represent a full computation tree instead of just a single trace. This would enable the support of computation tree logic (CTL) allowing for more temporal properties to be checked. Additionally, allowing the filmstrip model to reference an earlier state after an operation, thereby allowing loops, similar to a Kripke structure, helps with the verification of certain temporal operators. While in [23] the overhead of using the filmstrip transformation compared to other approaches is evaluated, further performance evaluations have to be performed with temporal logic included.

References

[1] G. Martin and W. Müller, *UML for SOC Design*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[2] F. Mischkalla, D. He, and W. Müller, “Closing the gap between uml-based modeling, simulation and synthesis of combined HW/SW systems,” in *Design, Automation and Test in Europe, DATE*, G. D. Micheli, B. M. Al-Hashimi, W. Müller, and E. Macii, Eds. IEEE, 2010, pp. 1201–1206.

[3] J. Cabot, R. Clarisó, and D. Riera, “On the verification of UML/OCL class diagrams using constraint programming,” *Journal of Systems and Software*, vol. 93, pp. 1–23, 2014.

[4] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, “On Challenges of Model Transformation from UML to Alloy,” *Software and System Modeling*, vol. 9, no. 1, pp. 69–86, 2010.

[5] A. D. Brucker and B. Wolff, “HOL-OCL: A Formal Proof Environment for UML/OCL,” in *FASE 2008*, ser. LNCS 4961, J. L. Fiadeiro and P. Inverardi, Eds. Springer, 2008, pp. 97–100.

[6] M. Gogolla, F. Büttner, and M. Richters, “USE: A UML-Based Specification Environment for Validating UML and OCL,” *Science of Computer Programming*, vol. 69, pp. 27–34, 2007.

[7] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. Cambridge, Massachusetts: The MIT Press, 2006.

[8] F. Hilken, L. Hamann, and M. Gogolla, “Transformation of UML and OCL Models into Filmstrip Models,” in *Proc. 7th Int. Conf. Model Transformation (ICMT 2014)*, D. D. Ruscio and D. Varró, Eds. Springer, LNCS 8568, 2014, pp. 170–185.

[9] P. A. P. Salas and B. K. Aichernig, “Automatic Test Case Generation for OCL: a Mutation Approach,” The United Nations University – International Institute for Software Technology, Tech. Rep. 321, 2005.

[10] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

[11] M. Gogolla and F. Hilken, “Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool,” in *Proc. Modellierung (MODELLIERUNG’2016)*, A. Oberweis and R. Reussner, Eds. GI, LNI, 2016, pp. 203–218.

[12] B. Kanso and S. Taha, “Temporal Constraint Support for OCL,” in *SLE 2012*, ser. LNCS 7745, K. Czarnecki and G. Hedin, Eds. Springer, 2012, pp. 83–103.

[13] S. Conrad and K. Turowski, “Temporal OCL Meeting Specification Demands for Business Components,” in *Unified Modeling Language: Systems Analysis, Design and Development Issues*. IGI Publishing, 2001, pp. 151–165.

[14] P. Ziemann and M. Gogolla, “OCL Extended with Temporal Logic,” in *5th Int. Conf. Perspectives of System Informatics (PSI’2003)*, M. Broy and A. Zamulin, Eds. Springer, Berlin, LNCS 2890, 2003, pp. 351–357.

[15] S. Flake and W. Müller, “Past- and Future-Oriented Time-Bounded Temporal Properties with OCL,” in *SEFM 2004*. IEEE Computer Society, 2004, pp. 154–163.

[16] M. Soden and H. Eichler, “Temporal extensions of OCL revisited,” in *Model Driven Architecture - Foundations and Applications, ECMDA-FA*, ser. LNCS, R. F. Paige, A. Hartman, and A. Rensink, Eds., vol. 5562. Springer, 2009, pp. 190–205.

[17] R. Bill, S. Gabmeyer, P. Kaufmann, and M. Seidl, “Model checking of ctl-extended OCL specifications,” in *Software Language Engineering, SLE*, ser. LNCS, B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju, Eds., vol. 8706. Springer, 2014, pp. 221–240.

[18] W. Dou, D. Bianculli, and L. C. Briand, “OCLR: A more expressive, pattern-based temporal extension of OCL,” in *Modelling Foundations and Applications, ECMFA*, ser. LNCS, J. Cabot and J. Rubin, Eds., vol. 8569. Springer, 2014, pp. 51–66.

[19] M. Al-Lail, W. Sun, and R. B. France, “Analyzing behavioral aspects of UML design class models against temporal properties,” in *International Conference on Quality Software*. IEEE, 2014, pp. 196–201.

[20] F. Hilken, P. Niemann, M. Gogolla, and R. Wille, “From UML/OCL to Base Models: Transformation Concepts for Generic Validation and Verification,” in *Proc. Int. Conf. Model Transformation (ICMT)*, ser. LNCS, D. Kolovos and M. Wimmer, Eds. Springer, 2015, pp. 149–165.

[21] A. Vakili and N. A. Day, “Temporal logic model checking in alloy,” in *ABZ*, ser. LNCS, J. Derrick, J. S. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, Eds., vol. 7316. Springer, 2012, pp. 150–163.

[22] A. Cunha, “Bounded model checking of temporal formulas with alloy,” in *ABZ*, ser. LNCS, Y. A. Ameur and K. Schewe, Eds., vol. 8477. Springer, 2014, pp. 303–308.

[23] F. Hilken, P. Niemann, M. Gogolla, and R. Wille, “Filmstripping and Unrolling: A Comparison of Verification Approaches for UML and OCL Behavioral Models,” in *Proc. 8th Int. Conf. Tests and Proofs (TAP 2014)*, M. Seidl and N. Tillmann, Eds. Springer, LNCS 8570, 2014, pp. 99–116.