Towards a Developer-Oriented Process for Verifying Behavioral Properties in UML and OCL Models

Khanh-Hoang Doan, Martin Gogolla, and Frank Hilken

University of Bremen, Computer Science Department, 28359 Bremen, Germany
{doankh,gogolla,fhilken}@informatik.uni-bremen.de

Abstract. Validation and verification of models in the software development design phase have a great potential for general quality improvement within software engineering. A system modeled with UML and OCL can be checked thoroughly before performing further development steps. Verifying not only static but also dynamic aspects of the model will reduce the cost of software development. In this paper, we introduce an approach for automatic behavioral property verification. An initial UML and OCL model will be enriched by frame conditions and then transformed into a (so-called) filmstrip model in which behavioral characteristics can be checked. The final step is to verify a property, which can be added to the filmstrip model in form of an OCL invariant. In order to make the process developer-friendly, UML diagrams can be employed for various purposes, in particular for formulating the verification task and the verification result.

Keywords: Validation and verification, Model testing, UML and OCL model, Behavior property verification.

1 Introduction

In software development, Model-Driven Engineering (MDE) is playing now a more and more important role. In recent years, the model-based approach has been becoming accepted, in particular by combining the UML (Unified Modeling Language) [15], the OCL (Object Constraint Language) [3] and some efficient tools. Available techniques in tools can be employed for the verification and validation of both static and dynamic properties of a software system.

As model validation and verification have been studied for a long time, a variety of approaches have been introduced. Typical approaches following this line have been discussed, e.g., the Dresden OCL tool [5], a toolset based on Abstract State Machines [17], and the tool USE for UML and OCL model property validation [7,9,19]. In [4] an approach for model consistency checking is introduced, and several correctness properties are automatically checked in [2]. UML model properties such as consistency, independence and consequence are validated in [9]. [16,13] present approaches for OCL constraint validation. However, many of these proposals concentrate on static aspects of the model, e.g., on consistency, independence, and consequences of and between OCL invariants. In order to also validate dynamic aspects, the approach in [8] introduces a transformation from UML and OCL models into so-called filmstrip models that represent sequences of system snapshots in a single object diagram. This filmstripping approach allows to check dynamic properties and will be applied as a central step within the complete verification process as described here.

In technical terms, the context of our work is the tool USE (UML-based Specification Environment) [7]. USE supports the description of a model in terms of a UML class diagram (with e.g. classes, associations) and UML state machines enriched by OCL constraints including class and state invariants as well as pre- and postconditions for operations and transitions. USE can visually represent class, object, sequence, statechart, and communication diagrams of a UML model and animate the model behavior based on command sequences. Offering a precise subset of UML and OCL can support the developer in employing a visual and thus user-oriented language for formulating development artifacts, in particular models and model properties. One central USE component is the so-called model validator that supports the validation and verification of properties based on the Kodkod relational logic [18]. A further USE component that we will employ is the so-called filmstrip transformation. It transforms an application model (with invariants and pre- and postconditions) into an equivalent so-called filmstrip model (with invariants only). This filmstrip model will be checked and tested with the model validator.

The remainder of the paper is organized as follows. Section 2 illustrates the general idea of our approach for a process consisting of several steps, and a simple example is introduced. Details of each verification step are introduced from Sect. 3 to Sect. 6. Particularly, Sect. 3 explains frame conditions and how to formulate and add them to the original model. Sect. 4 describes filmstrip models and how to transform models with frame conditions into filmstrip models. The step for verifying a behavior property is presented in Sect. 5, and in Sect. 6 we introduce the final step, transforming the verification result back into a sequence diagram of the original model. In Sect. 7 we discuss run-times of the verification tasks given in this paper before ending this contribution with some concluding remarks.

2 General Idea and Running Example

2.1 General Idea

Our approach for behavioral property verification can be divided into four steps as illustrated in Fig. 1. The input is a UML model enriched by OCL constraints and the output is a sequence diagram corresponding to a found scenario. The general idea for the verification process can be described as follows:

Step 1: Starting from an application model that describes structure and behavior of a system, add frame conditions. An application model is a UML model describing system structure and behavior completely in terms of OCL

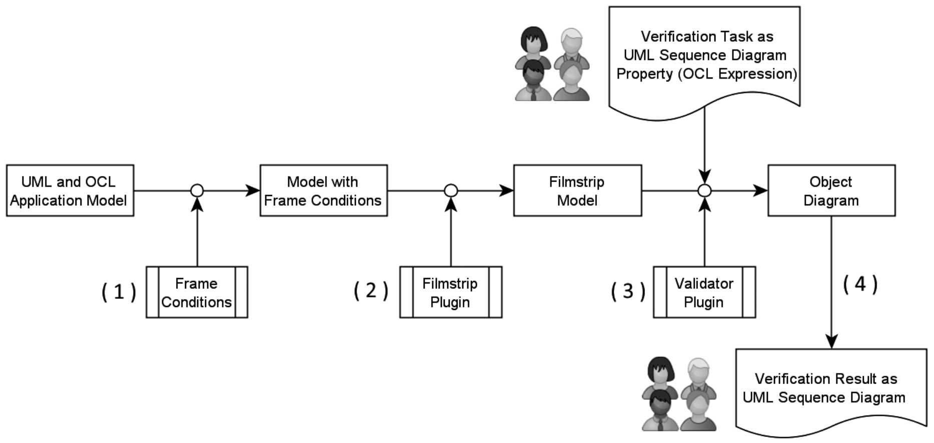


Fig. 1. General idea for the proposed verification process

and consisting of any number of classes, attributes, associations, and operations. The class diagram is enriched by class invariants and operation pre- and postconditions in form of OCL constraints. A frame condition makes the frame of an operation explicit. Particularly, it is a postcondition of an operation to assure that everything that is not in the scope of the operation (the frame) remains unchanged after the operation has been executed. Frame conditions will help the model validator in Step 3 to construct a scenario in a correct way.

Step 2: Transform the frame-conditioned model into a filmstrip model. In a filmstrip model, a single object diagram will describe a sequence of system states and operation calls between them. Roughly speaking, we can use an object diagram of a filmstrip model to describe interactions between objects in sequential order and the state transitions between the objects. Consequently, in the next step, a dynamic property, which is related to state transitions, can be verified in the context of the filmstrip model. The transformation step into the filmstrip model is performed automatically using the filmstrip plugin of USE.

Step 3: Verify a behavioral property of the filmstrip model. A behavioral property can be presented as an application model sequence diagram and can be analyzed by automatically constructing a scenario (an object diagram of the filmstrip model), in which a specified property is satisfied, or by showing that a valid scenario cannot be constructed within a finite search space. This step is performed automatically using the validator plugin of USE employing a configuration file describing the finite search space.

Step 4: Transform the generated object diagram from Step 3 (if the behavior property was satisfied) into a corresponding sequence diagram in the context of the application model. Presenting the verification result as a sequence diagram of the application model will increase readability and understandability of the verification process. The functionality of automatically transforming a filmstrip

object diagram to an application model sequence diagram will become part of the USE tool.
These four steps will be discussed in detail in the later parts of this paper.

2.2 Running Example

In this section a small example application model is given in order to demonstrate our approach. Its class diagram is presented in the left part of Fig. 2. The model describes synchronized traffic lights with (a) three attributes: r for red, y for

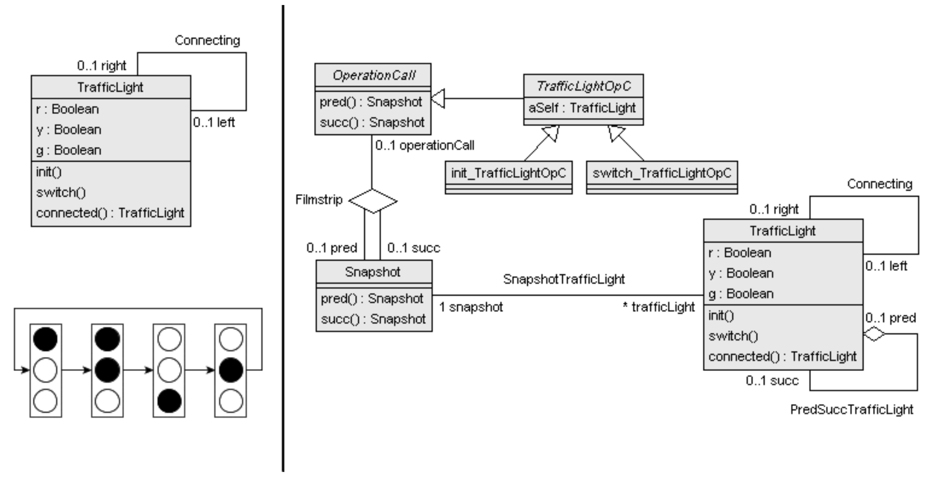


Fig. 2. Class diagram of example application model and switching phases.

yellow, g for green; (b) three operations: init() that initializes the values for a new traffic light, switch() that switches the light to the next state; and connected() that retrieves the connected light that is to be synchronized. Each traffic light is connected with at most one other traffic light to achieve the synchronization. The full model declaration including all invariants and pre- and postconditions is presented in [6]. The listing below shows the OCL invariants and postconditions of the switch() operation:

```
context TrafficLight
inv Ryg_RYg_ryG_rYg:
  (r=null and y=null and g=null) or
  (r and not y and not g) or (r and y and not g) or
  (not r and not y and g) or (not r and y and not g)
inv oneLight_onePair:
  (left→size()=1 and right→isEmpty()) or (left→isEmpty() and
  right→size()=1)
```

```
inv synchronize:
  (r<>null and y<>null) and
  (r and not y implies not connected().r and not connected().y) and
  (r and y implies not connected().r and connected().y) and
  (not r and not y implies connected().r and not connected().y) and
  (not r and y implies connected().r and connected().y)
context TrafficLight::switch()
post Ryg_2_RYg_2_ryG_2_rYg_2_Ryg:
  (r<>null and y<>null) and
  (r@pre and not y@pre implies r and y) and
  (r@pre and y@pre implies not r and not y) and
  (not r@pre and not y@pre implies not r and y) and
  (not r@pre and y@pre implies r and not y)
```

The first invariant `Ryg_RYg_ryG_rYg` identifies the states (values of attributes r, y and g) of a traffic light. In particular, a traffic light can only be in one of four states: red (r, not y, not g), red-yellow (r, y, not g), green (not r, not y, g), and yellow (not r, y, not g). The invariant `synchronize` determines how a pair of traffic lights synchronizes. When the left traffic light changes its state, the right light automatically changes its state respectively, e.g., a traffic light changes from the green state to yellow state when the connected light changes from red to red-yellow. As can be seen, the postcondition `Ryg_2_RYg_2_ryG_2_rYg_2_Ryg` only defines the changes of the attributes r and y, but does not include the attribute g for specifying the effect after the switch() operation has been executed. In a postcondition, the tag `@pre` refers to the state given at precondition time. On the other hand, we can see that the relationship between the r, y, and g attributes is fully fixed by the invariant `Ryg_RYg_ryG_rYg`. Consequently, the value of attribute g is fully determined by the value of r and y. The change of attribute g would be ambiguous if one would consider only the switch() postconditions. The question that comes up now is: what happens to attribute g? How will it be changed by the switch() operation? These questions will be answered by using the verification technique introduced in Sect. 5.

3 Adding Frame Conditions to Application Models

Postconditions typically specify in a declarative way effects of an operation, by expressing what they change. They often implicitly assume that everything else (the frame) remains unchanged. For a verification engine the question comes up how they can infer from postconditions which model elements are *not* allowed to change during an operation execution. This problem is called "*frame problem*" [1] and can be addressed by adding so-called frame conditions in form of OCL expressions. They indicate attributes and association ends that should not be changed after an operation has been executed. To add a frame condition to a model, we formulate it as an OCL expression in form of a postcondition. Various approaches for determining and formulating frame conditions have been introduced [11,12]. In this paper we apply the solution discussed in [14] to specify which properties are not allowed to change during the execution of the operations init() and switch(). Here we only explain how to formulate the frame condition for operation switch(). As can be seen from the postcondition `Ryg_2_RYg_2_ryG_2_rYg_2_Ryg` presented in Fig. 2, the properties r and y at post-state (i.e., not marked with `@pre`) are referenced in this postcondition. Consequently, properties r and y are *variable* to the switch() operation, which means that these properties are allowed to change when switch() is executed. Property g is not referenced in any postcondition, however it is referenced in the invariant `Ryg_RYg_ryG_rYg` with the connection to the variable properties r and y. Therefore, property g is also classified as variable. On the other hand, the state of the other traffic light objects, except the connected one, should not be changed. As the result, the frame conditions of the switch() operation is formulated as follows:

```
context TrafficLight::switch()
post trafficLightUnchangedExcept: let x=self in
  TrafficLight.allInstances@pre=TrafficLight.allInstances and
  TrafficLight.allInstances→forAll(t|
    (t.left@pre = t.left) and (t.right@pre = t.right) and
    (t<>x and t<>x.connected() implies t.r@pre=t.r) and
    (t<>x and t<>x.connected() implies t.y@pre=t.y) and
    (t<>x and t<>x.connected() implies t.g@pre=t.g))
```

In summary, this postcondition says: the switch() operation called on the traffic light object 'self' is only allowed to change the attributes r, y and g of self and its connected traffic light; everything else remains unchanged.

4 Transformation to Filmstrip Model

The application model enriched by frame conditions will be transformed into a so-called filmstrip model. A filmstrip model can describe dynamic aspects of an original application model, i.e., operations and state transitions, by static elements, i.e., UML classes and OCL invariants [10]. Particularly, each operation of classes from the application model is transformed into an `OperationCall` class, and a `Snapshot` object is created in the filmstrip model to represent the application model state at a point of time. With a filmstrip model we can describe information on the changes between the application model states and operation calls in one object diagram. It offers many possibilities for validation and verification of dynamic aspects, e.g., behavioral properties. Some elements of the application model are left unchanged, while others are converted with modification compared to the application model [8]. More detail about fimstrip model is introduced in [10]. The right part of Fig. 2 shows the class diagram of the filmstrip model after transforming the frame-conditioned model.

Most importantly, pre- and postconditions from the application model are transformed into invariants of the filmstrip model and realize behavioral properties, which are related to state transitions. These invariants can be checked in a single filmstrip model object diagram. One example of a transformed postcondition is presented as follows:

```
context switch_TrafficLightOpC
inv post_Ryg_2_RYg_2_ryG_2_rYg_2_Ryg:
  (aSelf.succ.r<>null and aSelf.succ.y<>null) and
  ((aSelf.r and not aSelf.y) implies
      (aSelf.succ.r and aSelf.succ.y)) and
  ((aSelf.r and aSelf.y) implies
      (not aSelf.succ.r and not aSelf.succ.y)) and
  ((not aSelf.r and not aSelf.y) implies
      (not aSelf.succ.r and aSelf.succ.y)) and
  ((not aSelf.r and aSelf.y) implies
      (aSelf.succ.r and not aSelf.succ.y))
```

The postcondition `Ryg_2_RYg_2_ryG_2_rYg_2_Ryg` of the operation switch() is renamed and altered to the invariant `post_Ryg_2_RYg_2_ryG_2_rYg_2_Ryg` of the new class `switch_TrafficLightOpC` in the filmstrip model. aSelf is an attribute of the `TraffictLightOpC` class, from which `switch_TrafficLightOpC` inherits. This attribute refers to the traffic light object on which the switch() operation is called. And `aSelf.succ` is the successor of the `aSelf` object after the switch() operation has been executed (i.e., the self object in the next snapshot).

Some new filmstrip invariants are generated by the filmstrip component. These invariants prevent faulty executions that would have been possible and thus bring the filmstripping model in line with execution of the operations in UML and OCL. In other words, they ensure correct behavior of the filmstrip model, e.g., by forbidding the snapshot object sequence to become a cycle (invariant `cycleFree`). More details of the complete filmstrip model description can be seen in [6].

5 Verifying Behavioral Properties

A behavioral property is a property related to a behavioral aspect of a design model, typically in connection with the model operations. In other words, checking a behavioral property is a type of verification task, that tries to prove whether a specific property can be reached or not reached under specific conditions, for example, with an operation call sequence. In our approach, first of all, an OCL expression for a behavioral property is added to the filmstrip model. This can be realized by the USE command `constraints -load constraintFile`, in which `constraintFile` is name of the file that contains the added OCL expression. Next, we execute the model validator from the USE GUI or on the CLI through the command `mv -validate propertyFile`. The `propertyFile` specifies the bounds for the search space of the model validator. For example, in the properties file the number of `OperationCall` objects, the number of `Snapshot` objects, or the number of links are stated. The model validator tries to construct a valid system state (object diagram) within the specified bounds. If successful, a system state will be established, that means the property is proved. And if not, the model validator reports that an object diagram cannot be found. This means that the logically negated property has been proved within the given bounds. Specifying proper bound numbers in the `propertyFile` for the model validator is important. Bounds must be big enough so that the property can be proved, but not too big so that the model validator can find answers within a small time frame.

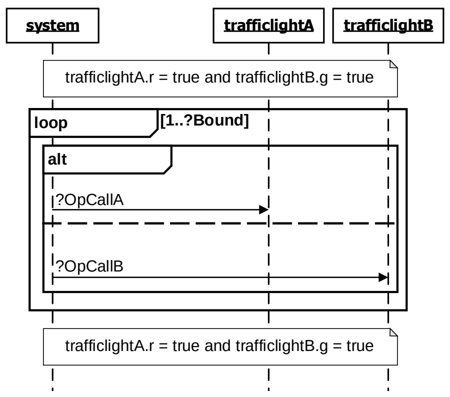

system
trafficlightA
trafficlightB
trafficlightA.r = true and trafficlightB.g = true
loop
[1..?Bound]
alt
?OpCallA
?OpCallB
trafficlightA.r = true and trafficlightB.g = true

Fig. 3. Verification task for Example 1 as sequence diagram

Example 1. In this example, the behavior property to be proved is: "Is it possible to construct a scenario (starting in a valid state and having transitions to future states only by operation calls) in which a pair of synchronized traffic light exists that shows initially red and green and after a number of operation calls red and green again?". This property can be expressed with a UML sequence diagram as in Fig. 3. The property can be made precise also with the following OCL expression:

```
context TrafficLight inv rg_And_rg_Again:
  TrafficLight.allInstances→exists(t|t.r and t.connected().g and
    Set{t.succ}→closure(succ)→exists(t1|t1.r and t1.connected().g))
```

After loading this invariant to the filmstrip model by running the command `constraints -load`, we execute the model validator with parameters specified in a property file. Fig. 4 presents the found object diagram [6]. The configuration specifies that the generated object diagram has exactly 10 `TrafficLight` objects, 5 `Snapshot` objects and 4 `switch_TrafficLightOpC` objects. As can be seen from the generated object diagram, a pair of synchronized traffic lights, trafficlight1 and trafficlight10 (upper dashed oval), shows red-green and the later incarnation, i.e., trafficlight3 and trafficlight8 (lower dashed oval), shows red-green again. From this we can confirm the claim, that the property can be satisfied for the running example. The protocol in Fig. 5 shows the detailed commands and the result. The run-times for verifying the property are specified within the outputs as well.

There are three run-times that the model validator shows in the result message. The 1st 'Translation time' (1200 ms) is the time needed to translate the

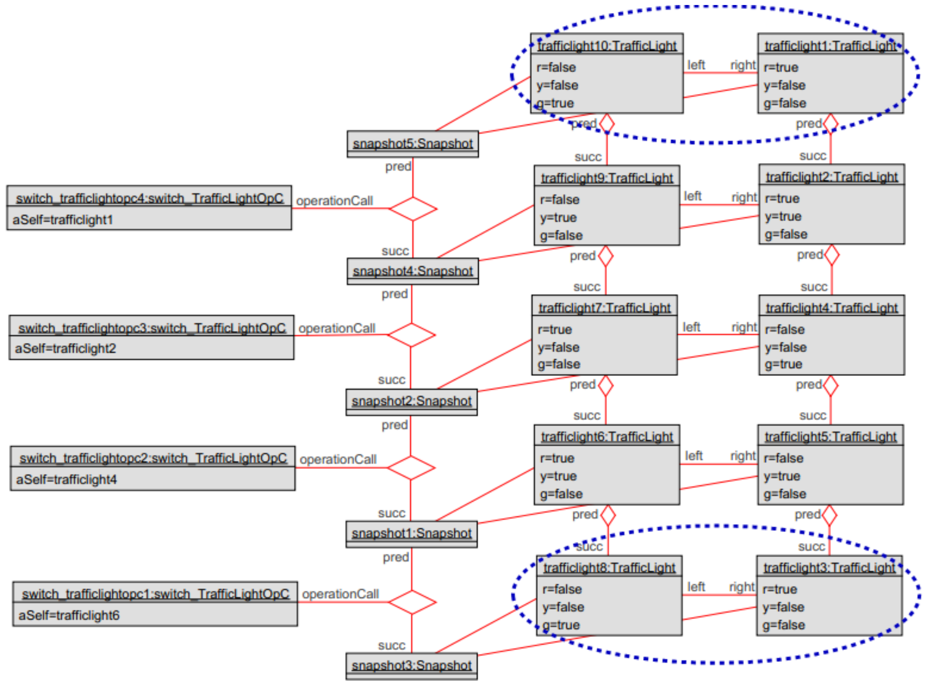

trafficlight10:TrafficLight
r=false
y=false
g=true
left
right
trafficlight1:TrafficLight
r=true
y=false
g=false
pred
snapshot5:Snapshot
pred
succ
switch_trafficlightopc4:switch_TrafficLightOpC
aSelf=trafficlight1
operationCall
trafficlight9:TrafficLight
r=false
y=true
g=false
trafficlight2:TrafficLight
r=true
y=true
g=false
snapshot4:Snapshot
switch_trafficlightopc3:switch_TrafficLightOpC
aSelf=trafficlight2
trafficlight7:TrafficLight
r=true
y=false
g=false
trafficlight4:TrafficLight
r=false
y=false
g=true
snapshot2:Snapshot
switch_trafficlightopc2:switch_TrafficLightOpC
aSelf=trafficlight4
trafficlight6:TrafficLight
r=true
y=true
g=false
trafficlight5:TrafficLight
r=false
y=true
g=false
snapshot1:Snapshot
switch_trafficlightopc1:switch_TrafficLightOpC
aSelf=trafficlight6
trafficlight8:TrafficLight
r=false
y=false
g=true
trafficlight3:TrafficLight
r=true
y=false
g=false
snapshot3:Snapshot

Fig. 4. Generated object diagram for Example 1

class diagram into the relational logic of Kodkod. This translation is only performed one time when executing the model validator the first time in a working session. The validator needs 1212 ms (2nd 'Translation time') to translate the relational formula and the configuration into SAT (this step is performed by Kodkod), and 180 ms (validator 'Solving time') to solve the translated relational formula by the underlying SAT solver. The total time for all verification tasks in this example with the specified bounds parameters is 5941 ms.

Example 2. The behavior property to be checked in the second example is: "Is it possible to construct a scenario in which one traffic light exists so that between two green states we have less than four transitions?". The following listing is the OCL expression for this property:

```
context TrafficLight inv lessthan_4transitions_between_2G:
  TrafficLight.allInstances→ exists (t | t.g and not t.succ.g and
    Set{t.succ}→closure(succ)→exists(t1|t1.g and
      Set{t}→closure(succ)→size() - Set{t1}→closure(succ)→size()<4))
```

To emphasize the important role of frame conditions for a verification task, first we apply our process without Step 2 that adds the frame conditions to the original application model. As the result, the validator finds a satisfying scenario as shown in Fig. 6. The configuration requires exactly 16 `TrafficLight` objects,

```
use> constraints -load rg_And_rg_Again.invs
     Added invariants:
     TrafficLight:: rg_And_rg_Again
use> mv -validate trafficLight1.properties
     ModelTransformator: Translation time: 1200 ms
     ModelValidator: SATISFIABLE
     ModelValidator: Translation time: 1212 ms Solving time: 180 ms
     ModelValidator: Create object Diagram
```

Fig. 5. Detailed commands and result for verifying property 1

4 `Snapshot` objects and 3 `switch_TrafficLightOpC` objects. It can be seen from Fig. 6, trafficlight6 shows green (at the point of time corresponding to snapshot1) and its latest reincarnation, trafficlight7, shows green again after three system state transitions (in the later point of time corresponding to snapshot2). These trafficlight objects are marked with the left and right dash ovals respectively in Fig. 6. In this case, the validator can find a satisfying scenario, because, without frame conditions, the attributes of one light can be changed when the switch() operation is executed on another light, which is not connected to the considered light. Here we have that trafficlight6 changes from the green to red when the switch() operation is executed on trafficlight5. On the other hand, when we apply our full process, with the same configuration, the validator answers 'Unsatisfiable'. That means that such scenario cannot be constructed within the bounds. Fig. 7 shows the detailed commands and the result 'Unsatisfiable'.

This example shows the importance of adding frame condition to the original application model in the entire verification process. Frame conditions support the validator to go not into the wrong direction when finding the answer.

6 Transforming Verification Results to Application Model Sequence Diagrams

The result of the model validator, if the verification property is satisfied, is a scenario in form of a filmstrip model object diagram. The ordinary developer, who must not know all details of the filmstripping approach, may find it difficult to understand and use the result in terms of the application model. Therefore, the transformation of the filmstrip model object diagram to an equivalent application model sequence diagram, which is more readable and practical, is helpful. The test case generated by the validator may be used in the later phases of software development. Fig. 8 is the application model sequence diagram corresponding to the generated filmstrip model object diagram for Example 1 in Sect. 5.

To built the sequence diagram from the filmstrip model object diagram, firstly, each application object (i.e., an object from a class of the original model) connected to the first snapshot object is considered as an initial object involved in the interaction (here, trafficlight1, trafficlight10). Each OperationCall object in the filmstrip model object diagram is turned into an operation call from the

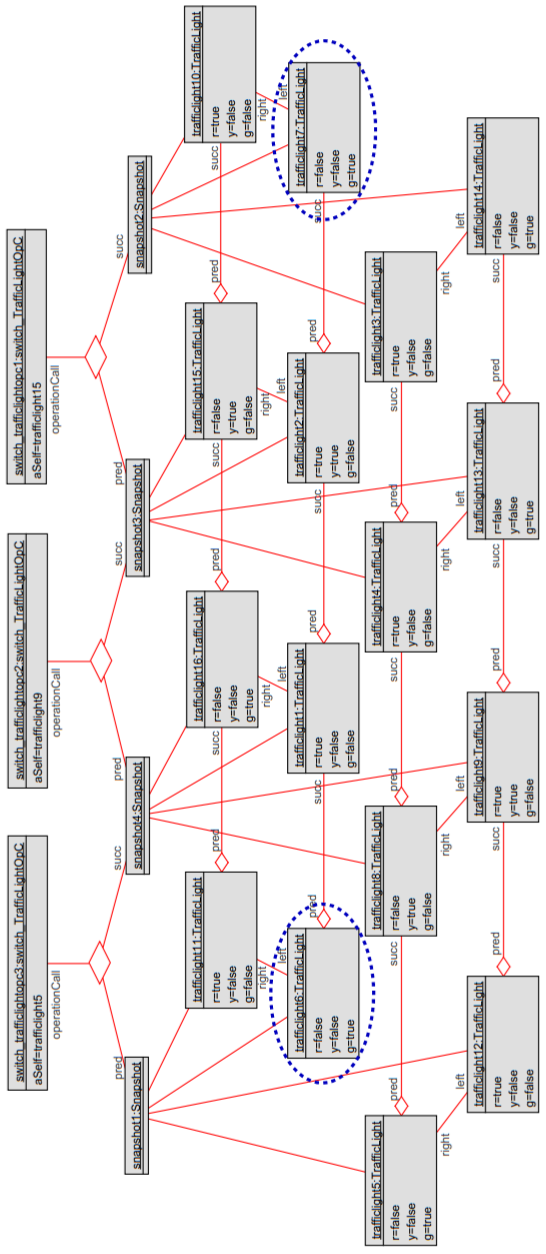
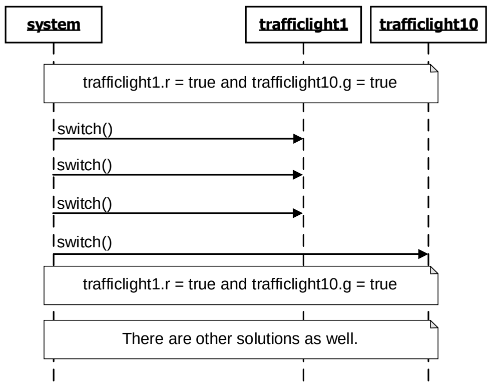

switch_trafficlightopc1:switch_TrafficLightOpC
aSelf=trafficlight15
operationCall
switch_trafficlightopc2:switch_TrafficLightOpC
aSelf=trafficlight9
operationCall
switch_trafficlightopc3:switch_TrafficLightOpC
aSelf=trafficlight5
operationCall
snapshot2:Snapshot
snapshot3:Snapshot
snapshot4:Snapshot
snapshot1:Snapshot
succ
pred
trafficlight10:TrafficLight
r=true
y=false
g=false
trafficlight7:TrafficLight
r=false
y=false
g=true
trafficlight14:TrafficLight
r=false
y=false
g=true
trafficlight15:TrafficLight
r=false
y=true
g=false
trafficlight2:TrafficLight
r=true
y=true
g=false
trafficlight3:TrafficLight
r=true
y=false
g=false
trafficlight13:TrafficLight
r=false
y=false
g=true
trafficlight16:TrafficLight
r=false
y=false
g=true
trafficlight1:TrafficLight
r=true
y=false
g=false
trafficlight4:TrafficLight
r=true
y=false
g=false
trafficlight9:TrafficLight
r=true
y=true
g=false
trafficlight11:TrafficLight
r=true
y=false
g=false
trafficlight6:TrafficLight
r=false
y=false
g=true
trafficlight8:TrafficLight
r=false
y=true
g=false
trafficlight12:TrafficLight
r=true
y=false
g=false
trafficlight5:TrafficLight
r=false
y=false
g=true
left
right
succ
pred

Fig. 6. Generated object diagram for Example 2 without frame conditions

```
use> constraints -load lessthan_4transitions_between_2g.invs
     Added invariants:
     TrafficLight:: lessthan_4transitions_between_2g
use> mv -validate trafficLight2.properties
     ModelTransformator: Translation time: 1248 ms
     KodkodModelValidator: UNSATISFIABLE
     KodkodModelValidator: Translation time: 1689 ms Solving time:
         1964544 ms
```

Fig. 7. Detailed commands and result for verifying property 2



system
trafficlight1
trafficlight10
trafficlight1.r = true and trafficlight10.g = true
switch()
switch()
switch()
switch()
trafficlight1.r = true and trafficlight10.g = true
There are other solutions as well.

Fig. 8. Application model sequence diagram for verification result of Example 1

system actor to one of the corresponding initial objects. The sequence diagram is complete when the last operation call has been handled.

7 Evaluation of Run-times for the Verification Tasks

One of the biggest problem that any verification tool has to deal with is the state-space explosion, i.e., the number of system states (the search space) may be huge even for relatively simple systems, or easily exceed the available computer memory. In our approach, the number of `OperationCall` objects (determining system transitions) is a key element that affects the search space, and that number corresponds to the run-time of a verification task.

We evaluate the run-time of our verification tasks by executing the model validator for the filmstrip model using Example 1 with gradually increasing the number of `switch_TrafficLightOpC` objects. Table 1 shows the resulting times. As can be seen from the table, the run-time of the verification tasks increases rapidly when the number of `OperationCall` objects increases gradually. On the other hand, from Fig. 7 we can see that the solving time for Example 2 is much higher than those in Tab. 1 although there were only 3 `OperationCall` objects configured for Example 2. In case of Example 2, the answer was “Unsatisfiable”,

Table 1. Run-times of verification tasks

Number of operation objects	Translation time [ms]	Solving time [ms]	Total time [m]
4	1 212	180	0.04
8	3 610	2 640	0.12
12	8 402	29 697	0.65
16	20 234	84 963	1.77
20	48 232	462 997	8.54
24	86 946	600 703	11.48

and therefore the model validator had to test all possibilities within the given bounds. Consequently, the solving time is high compared with the solving time for Example 1, for which the answer was "Satisfiable".

The results show that scenarios, i.e., sequence diagrams, with about 20 operation calls can be constructed in less than 10 minutes.

8 Conclusion and Future Work

This contribution has proposed a process for the verification of a behavioral property of a UML model enriched by OCL constraints. The inputs are an application model in form of a USE file and a property that needs to be verified in form of an OCL invariant or in form of a UML sequence diagram; the output is typically a test scenario in form of a sequence diagram. The idea of combining frame conditions, the filmstrip model, and the model validator in a complete process for behavior property verification together with sequence diagrams for verification tasks and verification results has not been discussed before. The last step of our process, the transformation of filmstrip object diagrams to a sequence diagram, will increase the readability and understandability of the verification approach. Most of the steps in our process are automatically performed by the USE tool and its plugins. The process will be adjusted and optimized in later works.

Future work can be done in various directions. First of all, a functionality that allows to automatically transform the generated filmstrip model object diagram to an application model sequence diagram will be worked out. Secondly, the idea for automatically formulating and adding frame conditions to a UML and OCL model should be studied further and supported by tool options. Future work has also to consolidate the approach with larger case studies and has to improve the efficiency of the validator searching process in the presence of filmstrip models.

References

1. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. Software Engineering, IEEE Transactions on 21(10), 785–798 (Oct 1995)

2. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: A tool for the formal verification of UML/OCL models using constraint programming. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering. pp. 547–548. ASE '07, ACM, New York, NY, USA (2007)
3. Cabot, J., Gogolla, M.: Object constraint language (OCL): A definitive guide. In: Bernardo, M., Cortellessa, V., Pierantonio, A. (eds.) Formal Methods for Model-Driven Engineering, Lecture Notes in Computer Science, vol. 7320, pp. 58–90. Springer Berlin Heidelberg (2012)
4. Dan, C., Mihai, P., Adrian, C., Cristian, B., Sorin, M.: Ensuring UML models consistency using the OCL environment. Electronic Notes in Theoretical Computer Science 102, 99 – 110 (2004), proceedings of the Workshop, OCL 2.0 - Industry Standard or Scientific Playground?
5. Demuth, B., Wilke, C.: Model and object verification by using Dresden OCL. In: Russian-German WS Innovation Information Technologies: Theory and Practice (2009)
6. Doan, K.H., Gogolla, M., Hilken, F.: Addendum to A Complete Process for Behavioral Properties Verification. Tech. rep., University of Bremen (2016), `http://www.db.informatik.uni-bremen.de/publications/intern/HOFM2016ADD.pdf`
7. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. 69(1-3), 27–34 (Dec 2007)
8. Gogolla, M., Hamann, L., Hilken, F., Kuhlmann, M., France, R.: From application models to filmstrip models: An approach to automatic validation of model dynamics. In: Modellierung (MODELLIERUNG'2014) (2014)
9. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, independence and consequences in UML and OCL models. In: Dubois, C. (ed.) Tests and Proofs, Lecture Notes in Computer Science, vol. 5668, pp. 90–104. Springer Berlin Heidelberg (2009)
10. Hilken, F., Hamann, L., Gogolla, M.: Transformation of UML and OCL models into filmstrip models. In: Di Ruscio, D., Varró, D. (eds.) Theory and Practice of Model Transformations, Lecture Notes in Computer Science, vol. 8568, pp. 170–185. Springer International Publishing (2014)
11. Kosiuczenko, P.: Specification of invariability in OCL. Software & Systems Modeling 12(2), 415–434 (2011)
12. Krieger, M.P., Knapp, A., Wolff, B.: Automatic and efficient simulation of operation contracts. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering. pp. 53–62. GPCE '10, ACM, New York, NY, USA (2010)
13. Kuhlmann, M., Gogolla, M.: Modeling and validating mondex scenarios described in UML and OCL with USE. Formal Aspects of Computing 20(1), 79–100 (2007)
14. Niemann, P., Hilken, F., Gogolla, M., Wille, R.: Extracting frame conditions from operation contracts. In: ACM/IEEE 18th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2015) (2015)
15. Object Management Group – OMG: Unified Modeling Language Specification, version 2.5 (2013), `http://www.omg.org/spec/UML/`
16. Richters, M., Gogolla, M.: Validating UML models and OCL constraints. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000 — The Unified Modeling Language, Lecture Notes in Computer Science, vol. 1939, pp. 265–277. Springer Berlin Heidelberg (2000)

17. Shen, W., Compton, K., Huggins, J.: A toolset for supporting UML static and dynamic model checking. In: Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International. pp. 147–152 (2002)
18. Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 4424, pp. 632–647. Springer Berlin Heidelberg (2007)
19. Ziemann, P., Gogolla, M.: Validating OCL specifications with the USE tool: An example based on the BART case study. Electronic Notes in Theoretical Computer Science 80, 157 – 169 (2003), eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03)