Filmstripping and Unrolling: A Comparison of Verification Approaches for UML and OCL Behavioral Models*

Frank Hilken, Philipp Niemann, Martin Gogolla, and Robert Wille

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

Abstract. Guaranteeing the essential properties of a system early in the design process is an important as well as challenging task. Modeling languages such as the UML allow for a formal description of structure and behavior by employing OCL class invariants and operation pre- and postconditions. This enables the verification of a system description prior to implementation. For this purpose, first approaches have recently been put forward. In particular, solutions relying on the deductive power of constraint solvers are promising. Here, complementary approaches of how to formulate and transform respective UML and OCL verification tasks into corresponding solver tasks have been proposed. However, the resulting methods have not yet been compared to each other. In this contribution, we consider two verification approaches for UML and OCL behavioral models and compare their methods and the respective workflows with each other. By this, a better understanding of the advantages and disadvantages of these verification methods is achieved.

1 Introduction

The *Unified Modeling Language* (UML) has been widely accepted as the standard language for modeling and documentation of software systems. UML allows for an initial description of a system at a high level of abstraction, i.e. before precise implementation steps are performed. For this purpose, UML employs appropriate description means which hide implementation details while being expressive enough to formally describe the structure and behavior of a complex system. Additionally, the *Object Constraint Language* (OCL) can be applied to refine a UML model with textual constraints describing further properties e.g. of the respective components or defining pre- and postconditions of their operations.

The resulting models may be composed of numerous different components with various relations, dependencies, or constraints and usually lead to non-trivial descriptions where errors can easily arise. Hence, guaranteeing that the

* This work was partially funded by the German Research Foundation (DFG) under grants GO 454/19-1 and WI 3401/5-1 as well as within the Reinhart Koselleck project DR 287/23-1.

resulting descriptions are plausible and consistent is an important as well as challenging task. This motivated the development of approaches for the validation and verification of UML/OCL models.

In this contribution, we focus on the verification of behavioral models, i.e. descriptions employing operations whose functionality is provided by OCL pre- and postconditions. Due to the formal nature of the corresponding UML/OCL components, automatic reasoning engines can be utilized in order to check whether certain properties do or do not hold. In particular, solutions relying on the deductive power of constraint solvers such as Kodkod or for *SAT Modulo Theory* (SMT) have been shown to be promising [17,23]. Here, two complementary approaches of how to formulate and transform respective UML and OCL verification tasks into corresponding solver tasks have been proposed, namely

- a solution which transforms the given problem into a so called *filmstrip model* [14], i.e. an equivalent UML/OCL description in which all behavioral model elements and the verification task are represented by static descriptions and, afterwards, are checked for interesting properties, and
- a solution which *unrolls* the dynamic behavior resulting in a skeleton for all possible system states while constraints and the verification task are directly formulated by means of an SMT theory to be solved by a corresponding solving engine [23].

Both approaches represent proper solutions which address the respective UML and OCL verification tasks. However, while certain differences between both approaches are evident at a first glance (e.g. the use of relational logic versus the use of an SMT engine), a detailed comparison of them has not been conducted yet.

In this contribution, we conduct such a comparison. More precisely, we contrast the workflows of both verification approaches to each other and provide a step-by-step description of the respective steps for each of them. Using a recently proposed UML/OCL model representing the *dining philosophers* problem (taken from [4]), the application of both approaches is illustrated. By this, an in-depth understanding of respective benefits and drawbacks of these complementary verification approaches is provided. This enables a better comprehension of their potential and possible application scenarios.

The remainder of this contribution is structured as follows: Section 2 provides an overview of the workflows of both verification approaches including their respective workflow steps. Afterwards, each step is described and illustrated in more detail in Section 3 using the model of the *dining philosophers* problem. Based on that, a discussion on the benefits and drawbacks of the approaches is provided in Section 4 before related work is considered and conclusions are drawn in Section 5 and Section 6, respectively.

2 Conceptual Workflows

Before the considered verification approaches are described in detail, this section briefly reviews their conceptual workflows. For this purpose, the major steps are

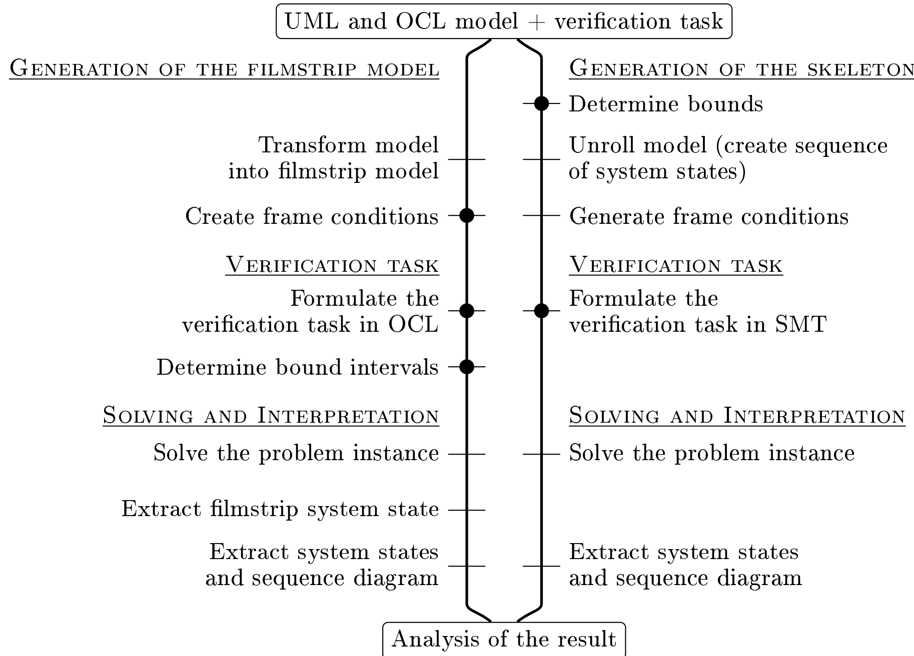


Fig. 1. Conceptual workflows of filmstripping (left) and unrolling (right)

illustrated in Fig. 1. Steps that require manual interaction are indicated by a bullet in order to distinguish them from steps that are performed automatically.

Both approaches take as input a UML model description enriched by OCL constraints together with a verification task which is to be performed on the model. Possible verification tasks comprise e.g. checking for deadlocks, verifying executability of operations, reachability of particular system states, or may address other behavioral aspects of the model.

First, the given UML/OCL model is extended in order to support the consideration of behavioral aspects. For the filmstripping approach, this includes an automatic transformation of the source model into the corresponding *filmstrip model* [14] followed by a manual creation of *frame conditions*, i.e. additional OCL constraints to limit the effects of the operation call to the relevant changes. For the unrolling approach, behavioral aspects are supported by automatically *unrolling* the model, i.e. creating an empty *skeleton* of system states (containing objects, their attributes and associations) for a certain number of observation points as well as operation calls connecting consecutive states. In contrast to the filmstripping approach, a (restricted set of) frame conditions is automatically generated. In order to create a skeleton of appropriate size, problem bounds (e.g. the number of observation points, the number of objects to be instantiated, or the range for primitive data types like integers to be considered) need to be fixed manually at this early stage. Note that bounding is not a special

characteristic of the unrolling approach, but a common procedure for verification purposes and also the filmstripping approach will employ this technique. Beyond that, bounding is necessary due to the complexity of the problem and justified by the fact that actual instances/implementations of the models will have finite dimensions and occupy finite resources anyway.

In the second stage, the addressed verification task is incorporated by adding constraints expressed in terms of OCL or SMT, respectively. The filmstripping approach requires a manual transformation of the verification task into OCL. More precisely, the verification task is first formulated in source model compatible OCL including elements of temporal logic. In a second step, this formulation is transformed into an OCL form that respects the characteristic structure of the filmstrip model. Finally, problem bound intervals are determined for the filmstripping approach as well. In contrast, in the unrolling approach several standard tasks like checking for deadlocks can be handled automatically. Others that involve more model-specific behaviour, have to be formulated manually in the SMT language [3], i.e. by further constraining attributes, associations, or operation calls.

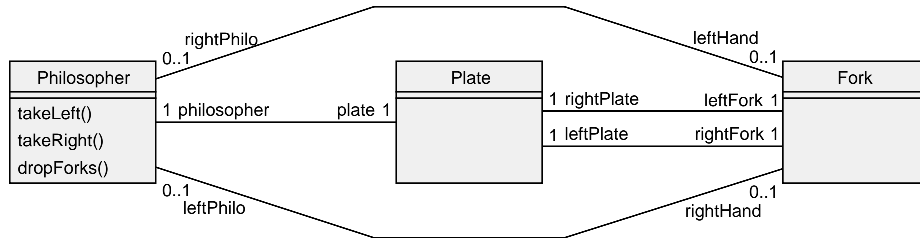
In both cases, the result of the second stage is a problem instance carrying the source model as well as the targeted verification task and problem bounds. This is passed to an appropriate solving engine which is supposed to determine an assignment satisfying all OCL/SMT constraints or has to prove the absence of such an assignment. For the filmstripping approach, relational logic is used on the basis of Kodkod [26] and Alloy [16], while the unrolling approach employs an SMT solver (like Boolector [6] or Z3 [9]). As a last step, both approaches translate the assignment retrieved from the solver back to the model context. The filmstripping approach provides an instance of the filmstrip model which contains all necessary information of the system states. Source model compatible object and sequence diagrams can be extracted if needed. Due to the different layer of abstraction, the unrolling approach extracts a sequence of system states (one for each observation point) and additionally creates a sequence diagram in order to provide the same information as contained in a (single) system state of the filmstrip model. Using this representation, the developer can analyze the result and draw conclusions with respect to the given verification task.

3 Comparison of the Verification Approaches

For a better comparison of the verification processes, they are exemplified using the same UML and OCL model (*dining philosophers*) and verification task (finding a deadlock) as a running example. We begin with the model definition and afterwards explain the approaches in detail separately.

3.1 Running Example Model Definition

The classic *dining philosophers* problem serves as an example to compare the verification approaches. The UML and OCL model is derived from [4] and slightly




```
context Fork inv maxOnePhilo:
  self.leftPhilo.oclIsUndefined()
    or self.rightPhilo.oclIsUndefined()
context Plate inv oneCircle: Set{ self }→closure(
  rightFork.rightPlate ) = Plate.allInstances()

-- operation takeLeft()
  pre emptyHand: self.leftHand.oclIsUndefined()
  post forkInHand: self.leftHand = self.plate.leftFork
-- operation takeRight()
  pre emptyHand: self.rightHand.oclIsUndefined()
  post forkInHand: self.rightHand = self.plate.rightFork
-- operation dropForks()
  pre hasLeftFork: not self.leftHand.oclIsUndefined()
  pre hasRightFork: not self.rightHand.oclIsUndefined()
  post emptyLeftHand: self.leftHand.oclIsUndefined()
  post emptyRightHand: self.rightHand.oclIsUndefined()
```

Fig. 2. The dining philosophers model

simplified. The well known problem of the forks, being shared by two philosophers each, has not changed. The model definition is shown in Fig. 2. A philosopher is connected to exactly one plate, which should not change during an execution period. Each plate has a left fork and a right fork, where two adjacent plates share one fork in between them, i.e. the left fork of one plate is the right fork of another plate. Lastly the philosopher is connected with the forks to model the picking up and dropping of forks.

The model embodies two additional constraints that cannot be expressed in UML. These requirements are specified as OCL invariants and shown at the bottom of Fig. 2. The first invariant `Fork::maxOnePhilo` enforces the important rule that a fork may only be used by a single philosopher at a time. The second invariant `Plate::oneCircle` assures a single circle of plates and forks. Otherwise the philosophers can split up into small groups that each have their own set of plates and forks.

The model dynamics are specified as pre- and postconditions for the three operations of the model, namely `takeLeft()`, `takeRight()` and `dropForks()`. The operations `takeLeft()` and `takeRight()` make the philosopher pick up the left fork – or right fork, respectively – if it is not already picked up. The operation `dropForks()` puts the forks back between the plates, leaving them ready to

be picked up again. The latter operation can only be invoked, if the involved philosopher has both forks in his hands. The model as shown in Fig. 2 has a serious flaw, in fact leads to a deadlock. This can be detected using verification approaches as described next.

3.2 Verification Using the Filmstripping Approach

Generation of the Filmstrip Model To find a deadlock in the *dining philosophers* model using the filmstripping approach, the first step is to transform the source model into the philosopher filmstrip model [14]. For easier reference, we call the source model the *application model* and our transformed model the *filmstrip model*. The result of the transformation is another UML and OCL model, which represents the model dynamics of the application model with classes, associations and invariants instead of operation pre- and postconditions. The expressiveness of the filmstrip model is the same as the application model, but all dynamic model elements have been transformed into static ones.

Example. Consider the class diagram of the filmstrip model in Fig. 3. It is an extension of the application model (Fig. 2), where the behavioral elements of the application model became structural elements in the filmstrip model. A system state of the application model is described by a `Snapshot`, to which every object of a state is linked, and several snapshots can be connected to a filmstrip with operation call objects (`OpC`) between each of them. Aside of this filmstrip connection, each object gets a reflexive association to link different representation of the same object along the snapshots. The three operations of the application model become concrete classes extending the base operation call class. The operation pre- and postconditions are transformed into invariants and corresponding OCL constraints ensure the correct representation of the model dynamics.

The next step in the verification process is the creation of frame conditions. The OCL invariant `noForkChangeExcept`, in Fig. 4, forces links between philoso-

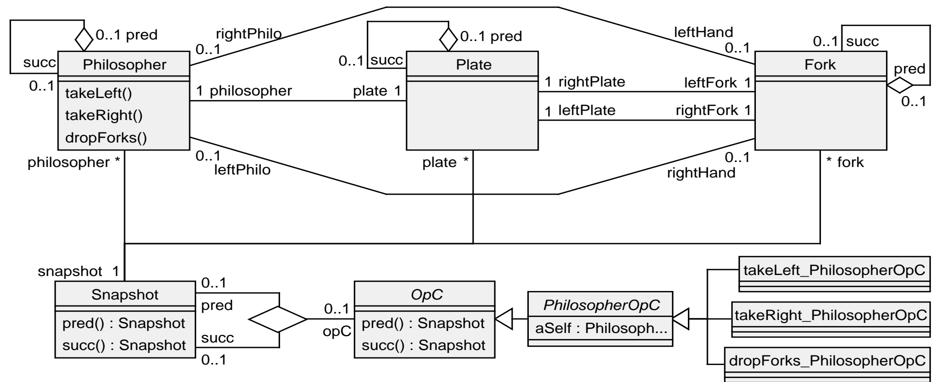


Fig. 3. Philosopher filmstrip model class diagram

```
context dropForks_PhilosopherOpC inv noForkChangeExcept:
  let except = aSelf in
  pred().philosopher→forAll( p | p <> except implies
      ( p.leftHand.succ = p.succ.leftHand
          and p.rightHand.succ = p.succ.rightHand ))
context Snapshot inv sameCountPhilosopher:
  not succ().oclIsUndefined() implies philosopher→size() =
      succ().philosopher→size()
context OpC inv predBecomesSucc:
  self.pred().philosopher→forAll( p |
      self.succ().philosopher→includes( p.succ ) )
```

Fig. 4. Example frame conditions for the dining philosophers filmstrip model

phers and forks to persist during the `dropForks()` operation call, except for the one philosopher, on which the operation is invoked. This philosopher shall drop the forks onto the table, i.e. the links between the philosopher and the forks are removed. Due to the assignment of the invariant to the specific operation call class, it is possible to specify different behavior for different operation calls, e.g. the same invariant for the operation `takeLeft()` allows changes for the specified exceptions' left hand only and all other links persist. Similar invariants are created for the other associations, to keep their links between operation calls.

In the philosophers example there are two more invariant types, also shown in Fig. 4, that are added to the model. One constrains the number of objects to be the same in every snapshot, i.e. no new philosopher joins the system, once it is running. An example for this constraint is represented by the invariant `sameCountPhilosopher`. The second condition enforces a link between all objects of successive snapshots, to prevent object destruction and creation, i.e. no philosopher gets exchanged with a new one. The invariant `predBecomesSucc` illustrates such constraint for the philosopher objects. These two conditions are added for every class of the application model. These extra constraints are not necessarily required for the verification, since the filmstrip model can handle object creation and destruction, but they simplify the formulation of the verification task for the application model, because they remove a lot of side effects. If an application model specifically handles object creation and destruction, these constraints do not need to be added.

Verification Task The next step towards the verification of the system is the transformation of the verification task into OCL. A wide variety of verification tasks can be expressed in OCL due to the capabilities of the filmstrip model. Special properties for a defined state can be expressed as an OCL invariant, e.g. to define an initial state for the system. Temporal requirements for the model can also be expressed as invariants, e.g. using one of the temporal OCL proposals [21,27,10]. Given that an application model state sequence is captured by a single filmstrip model state, it is possible to verify tasks expressed in terms of LTL formulas.

```
context Snapshot inv initialState:
  let firstSnapshot = Snapshot.allInstances()→select( s |
      s.pred().oclIsUndefined() ) in
  firstSnapshot.philosopher→forAll( ph | ph.leftHand = null and
      ph.rightHand = null )
context Snapshot inv deadlock:
  Snapshot.allInstances()→exists( s |
    not s.philosopher→exists( ph |
-- takeLeft preconditions and invariants
      (ph.leftHand.oclIsUndefined() and
          ph.plate.rightFork.leftPhilo.oclIsUndefined())
-- takeRight preconditions and invariants
      or (ph.rightHand.oclIsUndefined() and
          ph.plate.leftFork.rightPhilo.oclIsUndefined())
-- dropForks preconditions and invariants
      or (ph.leftHand.oclIsUndefined() and
          ph.rightHand.oclIsUndefined()) ) )
```

Fig. 5. Formulation of the verification task

Example. The OCL invariants to describe the verification task for the filmstrip model are shown in Fig. 5. The invariant `initialState` asserts that the system starts in a state where no philosopher has picked up a fork yet. The deadlock verification task is expressed in the OCL invariant `deadlock`. It defines a state for a snapshot, where no more operations can be invoked. This is done by looking at all possible preconditions of the model operations and ensure, that none of those are valid in the snapshot. Additionally the invocation of the operation may not interfere with an invariant, once the operation call is finished, thus invariants interfering with the postconditions are added as well. The individual parts for the three operations from the application model are marked in the constraint. Other properties like the number of philosophers are left open and will be chosen by the solver.

The result of this verification task not only describes the condition whether the system contains a deadlock or not, but also the final state, in which the system came to hold and the whole sequence of operation invocations that lead there from the initial state. In the case that there is no deadlock in the system, there exists no valid system state for the prepared model and the solver yields `unsatisfiable`.

The last preparation for the model is to determine the problem bound intervals, i.e. the minimum and maximum quantities of objects for each class and association. Especially by addressing the classes from the filmstrip model, it is possible to define how many and which operation invocations are allowed. For the dining philosopher example, the number of operation invocations is limited to 4 and the number of application class objects is limited to 10. The lower bounds are set to 0. Note, however, that these bounds limit the system state of the filmstrip model instead of a system state in the application model, i.e. they

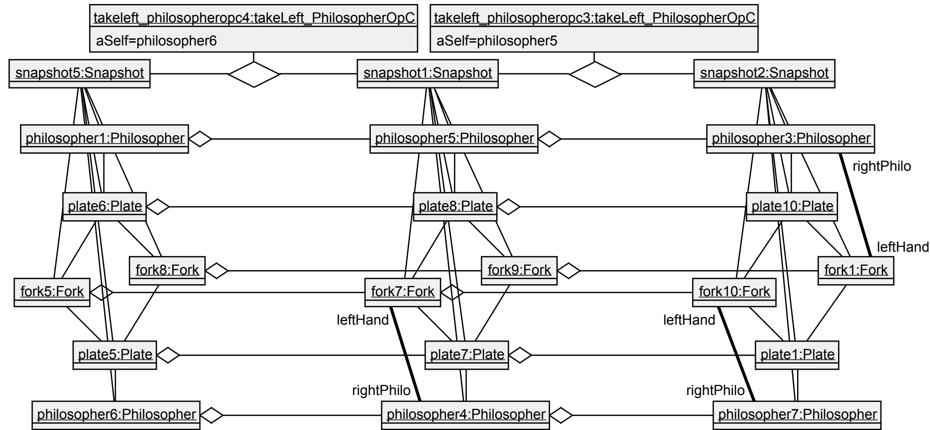


Fig. 6. Object diagram of the found solution for the philosopher verification task

allow for a maximum of 10 objects per class split among the snapshots of the filmstrip model. To affect the distribution per snapshot, OCL constraints can be used. The chosen configuration gives no hints for the verification engine, i.e. it is unknown whether a deadlock exists in the system and the validation tool shall try every possible combination.

Solving and Interpretation Now the problem description is complete and the next step is to solve the problem instance. Since all behavioral aspects of the application model have been transformed and eliminated in the filmstrip model, it can be validated with techniques designed for structural analysis [2,7,20,12]. We solve the problem using relational logic utilizing our model validator [17]. The model validator uses Kodkod [26] to transform the model, which itself uses Alloy [16] to encode the problem. The resulting problem instance is then solved by one of the supported SAT solvers, e.g. Sat4j, MiniSat or Glucose, which yields either the bindings of the found solution, if the problem is `satisfiable`, or marks the problem as `unsatisfiable` otherwise.

Example. In our example the solver finds a solution for the problem. The model validator extracts the bindings and creates an object diagram from it. Figure 6 illustrates the solution, representing a system state of the filmstrip model. The elements on the left show a snapshot, which represents the initial state. No philosopher is linked with any fork in this state, as defined. At the top are the operation calls. The first operation call lets the lower philosopher pick up the left fork. The snapshot in the middle illustrates this property with a link between the philosopher and the fork, labelled (`leftHand`,`rightPhilo`). All other connections remain the same. For the second operation call, the upper philosopher picks up the left fork, shown by the right snapshot. Now both philosophers each have their left fork picked up and the system is stuck. Nobody can pick up a left fork (`takeLeft()`), since everybody already has one, nobody can pick up a right fork (`takeRight()`), since they are in use and nobody can

drop their fork (`dropForks()`), because that is only possible once both forks are acquired. The classic *dining philosophers* problem.

The resulting system state contains all features of the filmstrip model. Therefore it is possible to express OCL queries using temporal conditions to validate the model even further and get hints for more verification tasks. For example, the expected behavior of the *dining philosophers* model can be expressed as a regular expression as

$$\big(\,(\,\texttt{tL tR}\mid\texttt{tR tL}\,)\,\texttt{dF}\,\big)^*$$

where `tL` represents the operation `takeLeft()`, `tR` represents `takeRight()` and `dF` represents `dropForks()`. In the context of the filmstrip model the expression can be transformed into an OCL query to find sequences of operation invocations in the system state, that match the pattern.

Finally, the object diagram of the filmstrip model can be transformed back to object and sequence diagrams of the application model. This is useful to track back errors in the application model revealed by performing the verification task on the filmstrip model.

3.3 Verification Using the Unrolling Approach

Generation of the Skeleton In this section, we describe the unrolling approach in detail using the previously introduced *dining philosophers* model as a running example. The basic idea of the approach is to unroll the source model, thereby generating a skeleton, i.e. an initially empty sequence of system states [23]. Note that the maximum number of objects and states must be determined in advance in order to generate a skeleton of appropriate size.

Example. Figure 7 shows a skeleton generated for the *dining philosophers* model with two instances of each class per state. Objects of the same type are automatically enumerated which allows to immediately identify corresponding

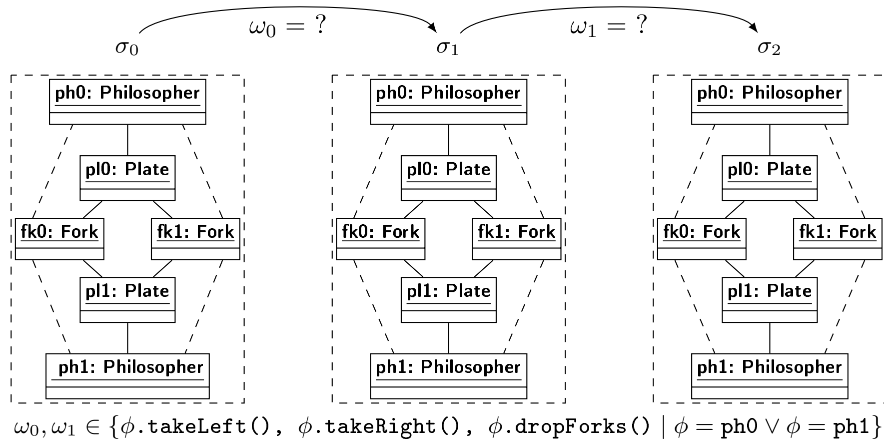


Fig. 7. Skeleton consisting of a sequence of system states and connecting operations

objects at different states (observation points) and to easily observe the lifeline of each object. Transitions between the states are made by operation calls ω_0, ω_1. Note that the number of objects and states must be determined in advance in order to generate a skeleton of appropriate size.

The purpose of the skeleton is to describe the dynamic behavior of the model at a level that can easily be transferred to a formulation suitable for SMT constraint solvers. In contrast to classical SAT solvers which expect the problem instance to be in *Conjunctive Normal Form (CNF)*, SMT solvers support higher-level theories which allows to formulate the problem instance at a higher level of abstraction thereby providing structural information that can accelerate the solving process. In our context, we especially make use of the theory of *Quantifier-Free Bit-Vectors (QF_BV)* logic which features bit-vectors of arbitrary length, comparisons like, e.g., $<$ or $\leq$, and other bit-vector operations [3]. Accordingly, object attributes and associations are translated to bit-vectors of appropriate length as illustrated by the following example.

Example. Consider the association (`leftHand`, `rightPhilo`) of the *philosophers* model (Fig. 2). In order to represent this in the skeleton we introduce bit-vector variables $\lambda_{\mathsf{leftHand}}$, one for each `Philosopher` object. Since the target of this association is of type `Fork`, the bit-width is set to the maximum number of forks with the implicit semantics that the i-th bit of $\lambda_{\mathsf{leftHand}}$ is set to 1 if and only if `Fork` i is part of the `leftHand` relation. Likewise, $\lambda_{\mathsf{rightPhilo}}$ variables (one for each `Fork` object) are used for the other association end. This is illustrated by means of Fig. 8.

Finally, the cardinality constraint 0..1 for the association ends is translated to the constraint that at most one bit of the bit-vector is set to 1. Many SMT solvers natively support such cardinality constraints for bit-vectors. For others, transformation frameworks like metaSMT [15] can be used to automatically translate these constraints to a more explicit, solver compatible form.

To complete the translation of the model's static components, also class invariants have to be addressed. Though the whole translation is performed automatically and hidden from the developer, this process is illustrated by the following example for the sake of completeness.

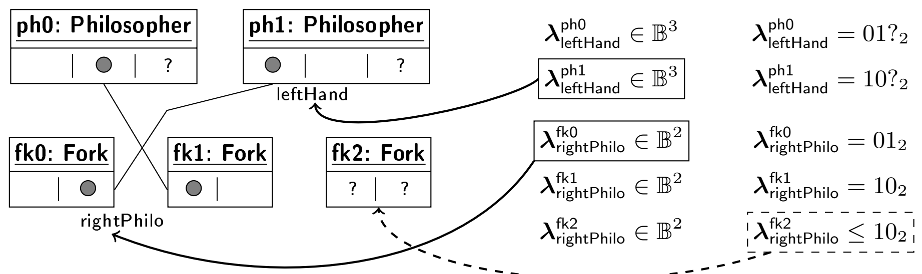


Fig. 8. Translating associations to λ-variables

Example. Consider the two invariants of the example model shown in Fig. 2.

1. The invariant `Fork::maxOnePhilo` is translated to a set of SMT constraints

```
(OR (= State_i::Fork_j::leftPhilo  #b00)
    (= State_i::Fork_j::rightPhilo #b00))
```

   one for each state (i) and fork (j). The `oclIsUndefined()` property of the association ends, i.e. there is no link to any philosopher, is translated to the assertion that the corresponding λ-variables are equal to the bit-vector 00_2.
2. The invariant `Plate::oneCircle` contains the iterator `closure` whose translation to SMT is very complex and not supported in our current implementation. For simplicity, we use the invariant

```
inv noIsolatedPlate:   self.leftFork <> self.rightFork
```

   instead. This invariant can be easily translated to a set of SMT constraints as above and is equivalent to the original invariant for up to three plates, since it then suffices that none of the plates forms a "circle" on its own.

The last important step of generating the skeleton is to add transitions between the states, i.e. to translate operation calls. For this purpose, we introduce ω-variables (one per transition) that are further constrained in order to represent respective operation calls.

Example. Consider the transition from the initial state σ_0 to the following state σ_1. Since there are two philosophers and three operations per philosopher, there are six possible operations in total, resulting in a bit-width of $\lceil \log_2(6) \rceil = 3$ for the corresponding ω_0-variable. For each possible value of ω (corresponding to some operation call), we add an SMT constraint saying that if this particular operation call is chosen for the state transition, we require that the corresponding pre- and postconditions hold in the pre- and post-state, respectively, and enforce frame conditions, e.g. which attributes and associations are allowed to be changed during an operation call and which shall not be altered. Figure 9 exemplarily shows the respective constraint for the `takeLeft()` operation invoked on philosopher `ph0`.

```
(=> (= omega_0 #b000)    ; representing ph0.takeLeft()
         ; pre-conditions hold in current state
    (AND (= State_0::Philosopher_0::leftHand #b00)
         ; post-conditions hold in succeeding state
         (= State_1::Philosopher_0::leftHand
            (ite (= State_1::Philosopher_0::plate #b01)
                 State_1::plate_0::leftFork
                 State_1::plate_1::leftFork))
         ; enforce frame conditions
         (= State_0::Philosopher_0::rightHand
            State_1::Philosopher_0::rightHand)
         ...
```

Fig. 9. SMT constraint representing a call to the `ph0.takeLeft()` operation

Remark. While pre- and postconditions are given by the source model, obtaining frame conditions is a non-trivial problem and can be an elaborate task. However, there is built-in functionality to generate frame conditions automatically under certain premises, e.g. fixing all variables that do not occur in postconditions. For more details on the whole translation process, we refer to [22].

Verification Task The next step is to consider the targeted verification task. First of all, the skeleton can be passed to the solving engine directly in order to check *consistency* of the model, i.e. to answer the question whether or not there exists a sequence of operation calls starting from an arbitrary initial state and satisfying all invariants, pre- and postconditions. In most cases, however, the verification task has to be included by further constraining attributes, associations, or operation calls. The approach offers a wide range of possibilities for this purpose:

- The constraining can be done very fine granular by addressing single variables of the skeleton, e.g. enforce a certain operation to be called at least once or at a certain position by constraining the corresponding ω-variables.
- At a larger scale, partially or completely preassigned states, e.g. the initial or final state, can be loaded and automatically constrain the corresponding variables or additional invariants can be enforced for a selection of states.
- Beyond that, several standard tasks like checking for deadlocks can be handled automatically. For instance, a deadlock finder extends the skeleton by adding a helper state for each possible operation call. These states have the same structure as normal system states, but only have a single invariant that states that the respective operation may not be called since either the preconditions are not fulfilled or the postconditions would raise a conflict with some invariant.

Example. In our running example, the deadlock finding method can be employed in multiple ways. If the number of states is set to one, i.e. no dynamic behaviour, it can prove whether deadlock states exists at all. Increasing the number of states, also sequences of operation calls leading to a deadlock state can be determined. Alternatively, the extracted deadlock state can be fed in as the final state of a reachability problem. Clearly, this is most useful in combination with a preassigned initial state.

Solving and Interpretation In the last stage of the approach, the problem instance is passed to an SMT solver, e.g. Boolector [6] or Z3 [9], which, in turn, either determines a satisfying assignment (SAT) or proves the absence of such an assignment (UNSAT). In the case of UNSAT, it is proven that the desired behaviour can not be achieved in the underlying model with respect to the specified problem bounds. In the case of SAT, a witness for the desired behaviour in form of object and sequence diagrams can be extracted automatically by translating the assignments of λ- and ω-variables (as demonstrated by Fig. 8).

Finally, the whole flow of the approach from the developer's perspective is illustrated by means of example code shown in Fig. 10. Our current implementation is written in Xtend/Closure and is fully integrated into Eclipse. We use

```
// Generate skeleton
input    = loadModel( "DiningPhilosophers.ecore" )
bounds   = new Bounds( AllObjectsSameBounds(2),
                       FixedNumberOfStates(3) )
skeleton = generateSkeleton( input.model, bounds )

// Incorporate verification task
initialState = loadState( "PhilosophersInitial.xmi" )
skeleton.getState(0).assign( initialState )
instance = DeadlockFinder( skeleton )

// Solve the problem instance
solver = new SMT_Solver
sat = solver.solve( instance )
assertEquals( true, sat )

// Extract states and sequence diagram
extractStatesAsXMI( solver.solution )
printTransitions( solver.solution )
```

Fig. 10. Deadlock finding in the *dining philosophers* model (developer's perspective)

Ecore as the input format for source models as well as XMI to in- and output system states. Note again that the whole translation process to SMT is performed automatically and hidden from the developer who does not need to write a single line of SMT code.

First, a skeleton is generated for the *dining philosophers* model with two philosophers/plates/forks each. Then the verification task, i.e. finding a deadlock that can be reached in three steps from a given initial state, is incorporated. Finally, the problem instance is passed to a solver and system states and transitions are extracted from the solution. Most parts of the code serve as template which can be reused for other problems or bounds. So far, problem bounds like the number of states or the number of objects per state have to be specified explicitly. However, we plan to support interval bounds in order to delegate the exact determination of bounds from the developer to the solving engine.

After setting up the instructions shown in Fig. 10, the problem considered here can be solved fully automatically.

4 Discussion of Comparison Criteria

After we have seen both verification approaches illustrated and applied to the same example, we now discuss their respective pros and cons. A comparison of core criteria (that are not necessarily disjoint) is summarized in Table 1.

The level of operation is a crucial difference between the approaches. The filmstripping approach mainly operates at the model level (UML/OCL) while the unrolling approach operates much closer to the solver level (SMT).

The procedure and applicability of the approaches is consequently quite different. Filmstripping essentially relies on manual interaction, particularly for the formulation of frame conditions and the verification task. However, these are to be formulated in UML/OCL which can be expected to be the designer's expertise. This allows for a higher flexibility and more universal applicability. In contrast, the unrolling approach is highly automated (automatic generation of frame conditions, predefined verification tasks) at the expense of a somewhat more restricted applicability. More precisely, some features of OCL are currently not supported.

Frame conditions are formulated manually for the filmstripping approach in due consideration of the structure of the derived filmstrip model. Therefore, they are problem-specific and thus compatible to the input model. In contrast, for the unrolling approach the frame conditions are generated automatically following a given set of rules, which may not be adequate for every given model.

Verification task To formulate the verification task, the engineer might need to understand the basics of the approaches. For the filmstripping approach, this is the structure of the filmstrip model which have to be enriched by additional OCL constraints. In contrast, the unrolling approach requires the verification task specified by means of SMT constraints which require a deeper understanding of SMT. This can not always be expected from the designer. However, for common verification tasks, such as reachability and deadlock detection, predefined automatic checks can be conducted which require no further expertise at all.

Search bounds are provided as intervals for the filmstripping approach. This makes the determination easier, e.g. when the exact problem bounds are unknown, but has a negative impact on solving times. Changing the bounds also does not affect the other steps of the verification task. In contrast, the unrolling approach is currently based on fixed bounds. In the case that the initial bounds are not sufficient, new bounds have to be determined and individual constraints may have to be adapted to these new bounds.

Table 1. Overview of important comparison criteria for the verification approaches

Criterion	**Filmstripping**	**Unrolling**
Level of operation	model level (UML/OCL)	solver level (SMT)
Procedure	essential manual interaction, thus more flexible	Highly automated, but less flexible
Applicability	Universal	Restricted
Frame conditions	Explicit formulation for model	Generated from set of rules
Verification task	Formulation in OCL (templates possible)	Formulation in SMT (some predefined)
Search bounds	Intervals	Fixed
Validation on result	OCL queries on filmstrip state	Not directly possible
Runtime	Good solving time	Optimized solving time
Solving engine	Relational logic (Kodkod)	SMT solver

Validation on result In the filmstripping approach, the extracted filmstrip state contains all behavioral features and can be directly accessed by OCL queries, allowing for further validation on the result. For the unrolling approach, the extracted diagrams are split into several object and sequence diagrams that are not directly accessible by OCL queries.

Runtimes Previous results for static model aspects [24] indicate a structural advantage (in terms of *runtimes*) of solver-driven approaches against model-driven approaches. This is in accordance with what we observed for the verification of behavioral aspects in this study (though a detailed analysis is left for future work), even if the higher manual interaction for the filmstripping is ignored.

Solving engine The filmstripping approach uses relational logic to solve problem instances. In this field currently only one solving engine is available. Using SMT in the unrolling approach allows to choose from a wide variety of solvers.

Performance In general, the performance of both approaches highly depends on the complexity of the input model and the desired verification task and, thus, is hard to compare. Moreover, there is no standard metric for the complexity of OCL or SMT constraints. Consequently, the effort for manual creation of frame conditions (in the filmstripping approach) and manual incorporation of verification tasks (in both approaches) cannot be measured precisely.

Overall, the unrolling approach promises fast results for a set of common, predefined verification tasks and input models that are compatible with the automatic generation of frame conditions (like, but clearly not limited to the considered *dining philosophers* model). For rather complex models, e.g. containing sophisticated side effects in the OCL constraints, or very dedicated verification tasks, the more flexible filmstripping approach is likely to be the better choice, at the price of substantial manual interaction.

5 Related Work

Besides from the approaches already mentioned, the two discussed methods have connections to related papers. In a previous contribution [13] we have identified verification tasks like consistency and independence of invariants in UML and OCL models and established a benchmark. The running example in this paper (*dining philosphers*) would be another candidate for the benchmark. In contrast to testing methods, there are a number of works applying interactive theorem proving techniques for UML and OCL, like for example the works based on PVS [18], the KeY approach [1], and the combination of testing and proving based on Isabelle and HOL/OCL [5]. A classification of model checkers with respect to verification tasks can be found in [11].

(Semi)-automatic proving approaches for UML class properties have been put forward on the basis of description logics [19], on the basis of relational

logic and pure Alloy [2] using a subset of OCL, and in [25] focussing on model inconsistencies by employing Kodkod, the programming interface of Alloy.

Verification of OCL operation contracts have been studied on the basis CSP solvers in [8]. The unrolling approach tackled in this paper was presented in [23] and the filmstripping approach in [14].

6 Conclusion and Future Work

In this contribution, we provided a comparison of the filmstripping approach to the unrolling approach – two recently proposed solutions aiming for the verification of behavioral models given in UML/OCL. Both approaches allow to check the functional correctness of a system description prior to its implementation. However, the fashion in which they formulate and eventually solve the respective verification tasks is significantly different. Our comparison discussed the main differences and, by this, provided a better understanding of the advantages and disadvantages of these verification methods. Future work will focus on the analysis and extension of these verification approaches with respect to scalability, i.e. the support of larger and more complex models, as well as applicability, i.e. the support of further descriptions means and verification tasks.

References

1. Ahrendt, W., Beckert, B., Hähnle, R., Schmitt, P.H.: KeY: A Formal Method for Object-Oriented Systems. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS. Lecture Notes in Computer Science, vol. 4468, pp. 32–43. Springer (2007)
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. Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2010)
4. 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)
5. Brucker, A.D., Wolff, B.: Semantics, calculi, and analysis for object-oriented specifications. Acta Inf. 46(4), 255–284 (2009)
6. Brummayer, R., Biere, A.: Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: Tools and Algorithms for the Construction and Analysis of Systems. pp. 174–177 (2009)
7. 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)
8. Cabot, J., Clarisó, R., Riera, D.: Verifying UML/OCL Operation Contracts. In: Leuschel, M., Wehrheim, H. (eds.) IFM. Lecture Notes in Computer Science, vol. 5423, pp. 40–55. Springer (2009)
9. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)

10. Flake, S., Müller, W.: Past- and Future-Oriented Time-Bounded Temporal Properties with OCL. In: SEFM 2004. pp. 154–163. IEEE Computer Society (2004)
11. Gabmeyer, S., Brosch, P., Seidl, M.: A Classification of Model Checking-Based Verification Approaches for Software Models. Proceedings of VOLT 13 (2013)
12. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69 (2007)
13. Gogolla, M., Büttner, F., Cabot, J.: Initiating a Benchmark for UML and OCL Analysis Tools. In: Veanes, M., Vigano, L. (eds.) Proc. 7th Int. Conf. Tests and Proofs (TAP 2013). pp. 115–132. Springer, Berlin, LNCS 7942 (2013)
14. 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.) Proc. Modellierung (MODELLIERUNG'2014). Gesellschaft für Informatik, LNI (2014)
15. Haedicke, F., Frehse, S., Fey, G., Große, D., Drechsler, R.: metaSMT: Focus on your application not on solver integration. In: In: DIFTS'12 (2012)
16. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge, Massachusetts (2006)
17. Kuhlmann, M., Gogolla, M.: From UML and OCL to Relational Logic and Back. In: France, R., Kazmeier, J., Breu, R., Atkinson, C. (eds.) Proc. 15th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2012). pp. 415–431. Springer, Berlin, LNCS 7590 (2012)
18. Kyas, M., Fecher, H., de Boer, F.S., Jacob, J., Hooman, J., van der Zwaag, M., Arons, T., Kugler, H.: Formalizing UML Models and OCL Constraints in PVS. Electr. Notes Theor. Comput. Sci. 115, 39–47 (2005)
19. Queralt, A., Artale, A., Calvanese, D., Teniente, E.: OCL-Lite: Finite reasoning on UML/OCL conceptual schemas. Data Knowl. Eng. 73, 1–22 (2012)
20. 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)
21. 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)
22. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In: Gogolla, M., Wolff, B. (eds.) TAP. Lecture Notes in Computer Science, vol. 6706, pp. 152–170. Springer (2011)
23. Soeken, M., Wille, R., Drechsler, R.: Verifying dynamic aspects of UML models. In: DATE. pp. 1077–1082. IEEE (2011)
24. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using Boolean satisfiability. In: DATE. pp. 1341–1344. IEEE (2010)
25. Straeten, R.V.D., Puissant, J.P., Mens, T.: Assessing the Kodkod Model Finder for Resolving Model Inconsistencies. In: France, R.B., Küster, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA. Lecture Notes in Computer Science, vol. 6698, pp. 69–84. Springer (2011)
26. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer (2007)
27. 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)