Coverage of OCL Operation Specifications and Invariants*

Mathias Soeken[1,2], Julia Seiter[1], and Rolf Drechsler[1,2]

[1] Faculty of Mathematics and Computer Science, University of Bremen, Germany
[2] Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
{msoeken,jseiter,drechsle}@cs.uni-bremen.de

Abstract. We consider operation coverage of OCL operation specifications and invariants in class diagrams with respect to sequence diagrams. The coverage criteria are based on the operations that are executed from the sequence diagrams and their asserted OCL subexpressions. We propose an algorithm that automatically generates a set of sequence diagrams in order to maximise these coverage criteria. A model finder is leveraged for this purpose. As a result, also operations and constraints can be determined that can never be executed and asserted, respectively. Our algorithm has been implemented in the UML specification tool USE.

1 Introduction

Given a class diagram with OCL operation specifications, invariants and a set of sequence diagrams, we define two coverage metrics that measure (1) how many operations of the class diagram have been called and (2) how many OCL subexpressions evaluated to true for this purpose. We define the coverage semantics on top of the precise modelling approach that has been presented by Mark Richters in [11]. As a result, the coverage metrics can readily be integrated in the context of formal analysis tools.

We demonstrate this by utilising model finders for behavioural modelling tasks to automatically generate sequence diagrams to increase coverage with respect to the defined metrics. Since model finders traverse the complete search space (and often efficiently), also "dead" operations or "dead" subexpressions can be found with our algorithm. Analogously to "dead code" in software development, these refer to operations that can never be called or subexpressions that never evaluate to true.

We have integrated the coverage metrics in the UML specification tool USE [6]. When starting the program with a class diagram and some initial sequence diagrams, the initial coverage is reported. The user of the tool can then start the

* This work was supported by the German Federal Ministry of Education and Research (BMBF) (01IW13001) within the project SPECifIC, by the German Research Foundation (DFG) (DR 287/23-1), and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

model finder to generate new sequence diagrams which successively increase the coverage or pinpoint the user to "dead" operations or subexpressions.

Coverage metrics for modelling languages have been considered in the past, but rarely have methods been provided to automatically increase the proposed coverage criteria. In [12] coverage criteria based on the execution traces of sequence diagrams have been defined, but no algorithm has been provided that generates input data to increase the coverage. An approach very similar to ours has been proposed in [17] where model finders are exercised in order to find sequence diagrams that adhere to a given specification. However, coverage has not been considered in this context.

The remainder of the paper is structured as follows. The next section reviews class diagrams, system states, and model finding and introduces the formal notation that is used as a basis in the paper. Section 3 proposes two coverage criteria and in Sect. 4 it is described how model finding can be utilised in order to automatically find sequence diagrams to increase the coverage of a class diagram. The implementation in USE is illustrated in Sect. 5 before related work is discussed in Sect. 6. Section 7 concludes the paper.

2 Preliminaries

This section introduces a notation that is used to describe class diagrams and system states in the remainder of the paper. Also, model finding is reviewed.

2.1 Class Diagrams and System States

We are making use of the notation that has been introduced in [11]. Associations have no immediate influence on our proposed coverage metric and therefore we use simpler definitions that omit details on associations.

Definition 1 (Class diagram). *A* class diagram *is denoted as*

$$\mathcal{M} = (\textsc{Class}, \textsc{Att}_c, \textsc{Op}_c, \prec) \ , \tag{1}$$

where

- Class *is a finite set of class names,*
- $\textsc{Att}_c$ *are sets of attributes for each class* $c \in$ Class *defined as signatures* $a : t_c \to t$*, where* a *is the attribute name,* t_c *is the type of class* c*, and* t *is the attribute type,*
- $\textsc{Op}_c$ *are sets of operations for each class* $c \in$ Class *defined as signatures* $\omega : t_c \times t_1 \times \cdots \times t_n \to t$*, where* ω *is the operation name,* t_c *is the type of class* c*,* $t_1, \ldots, t_n$ *are the types of the operation's* n *parameters, and* t *is the operation's return type,*
- *and* $\prec$ *is a partial order on* Class *to reflect the generalisation hierarchy.*

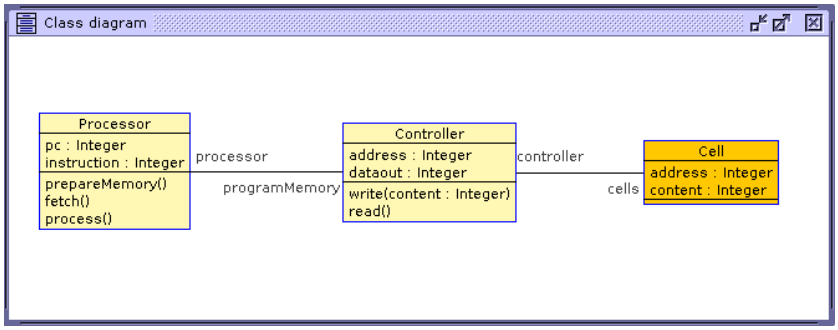

Fig. 1. Class diagram

We define

$$\mathrm{ops}(\mathcal{M}) \stackrel{\mathrm{def}}{=} \bigcup_{c \in \textsc{Class}} \textsc{Op}_c \ . \tag{2}$$

For $\textsc{Att}_c$ and $\textsc{Op}_c$ their reflexive closures

$$\begin{aligned} \textsc{Att}_c^* &\stackrel{\mathrm{def}}{=} \textsc{Att}_c \cup \bigcup_{c' \in \mathrm{parents}(c)} \textsc{Att}_{c'} \quad \text{and} \\ \textsc{Op}_c^* &\stackrel{\mathrm{def}}{=} \textsc{Op}_c \cup \bigcup_{c' \in \mathrm{parents}(c)} \textsc{Op}_{c'} \end{aligned} \tag{3}$$

are defined with $\mathrm{parents}(c) \stackrel{\mathrm{def}}{=} \{c' \mid c' \in \textsc{Class} \wedge c \prec c'\}$.

Example 1. Figure 1 shows a class diagram that will serve as running example throughout the paper. It models the memory access part of a processor architecture. The processor which has a program counter and a current instruction is connected to a memory controller which offers operations to read and write to memory which is represented in terms of cells.[3] The precise formal notation of the class diagram is

$$\textsc{Class} = \{\mathit{Processor}, \mathit{Controller}, \mathit{Cell}\ \}$$

$$\begin{aligned} \textsc{Att}_{\mathit{Processor}} = \{ \quad & \mathrm{pc} : \mathit{Processor} \to \mathit{Integer}, \\ & \mathrm{instruction} : \mathit{Processor} \to \mathit{Integer}\ \} \end{aligned}$$

$$\begin{aligned} \textsc{Att}_{\mathit{Controller}} = \{\ & \mathrm{address} : \mathit{Controller} \to \mathit{Integer}, \\ & \mathrm{dataout} : \mathit{Controller} \to \mathit{Integer}\ \} \end{aligned}$$

$$\begin{aligned} \textsc{Att}_{\mathit{Cell}} = \{\ & \mathrm{address} : \mathit{Cell} \to \mathit{Integer}, \\ & \mathrm{content} : \mathit{Cell} \to \mathit{Integer}\ \} \end{aligned}$$

[3] The class diagram can be downloaded as a model for USE at `www.informatik.uni-bremen.de/agra/files/memory.use`

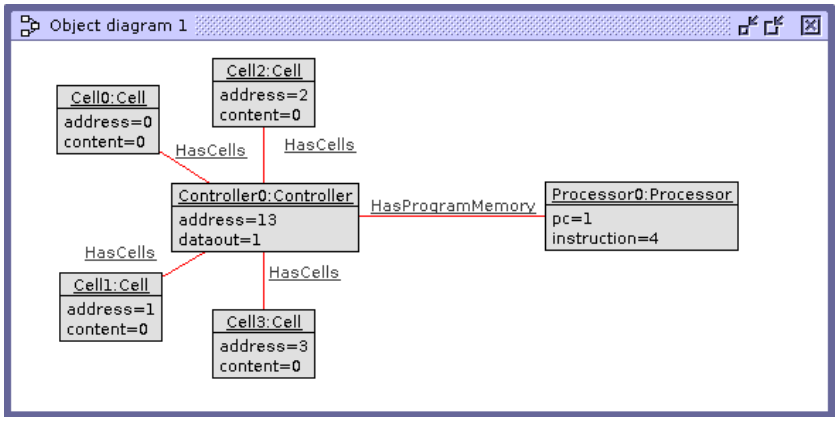

Fig. 2. System state

$$\begin{aligned}\text{OP}_{Processor} = \{\ &\text{prepareMemory} : Processor \to VoidType,\\ &\text{fetch} : Processor \to VoidType,\\ &\text{process} : Processor \to VoidType\ \}\end{aligned}$$

$$\begin{aligned}\text{OP}_{Controller} = \{\ &\text{write} : Controller \times Integer \to VoidType,\\ &\text{read} : Controller \to VoidType\ \}\ .\end{aligned}$$

Definition 2 (Class domain). *The set of* object identifiers *of a class* $c \in \text{CLASS}$ *is given by an infinite set* $\text{oid}(c) = \{\underline{c}_1, \underline{c}_2, \dots\}$*. Then, the domain of* c *is defined as*

$$I_{\text{CLASS}}(c) \stackrel{\text{def}}{=} \bigcup_{\substack{c' \in \text{CLASS} \\ c' \preceq c}} \{\text{oid}(c')\}\ . \tag{4}$$

In general, we will use the letter I to denote an interpretation mapping [11] that defines the semantics of OCL expressions.

Definition 3 (System state). *A system state for a class diagram* $\mathcal{M}$ *is a structure*

$$\sigma(\mathcal{M}) = (\sigma_{\text{CLASS}}, \sigma_{\text{ATT}})\ , \tag{5}$$

with

- *finite sets* $\sigma_{\text{CLASS}}(c) \subset \text{oid}(c)$ *containing all objects of class* $c \in \text{CLASS}$ *in the system state and*
- *functions* $\sigma_{\text{ATT}}(a) : \sigma_{\text{CLASS}}(c) \to I(t)$ *for each* $a : t_c \to t \in \text{ATT}^*_c$.

$I(t)$ *is an interpretation function for variables of types* t.

If the context is clear, $\mathcal{M}$ can be omitted and a system state is simply written as σ.

Example 2. A valid system state for the class diagram in Fig. 1 is shown in Fig. 2.

Class diagrams can be accompanied by expressions in the object constraint language (OCL, [15]) that is part of the UML standard. OCL allows the specification of formal constraints in the context of a model. Since constraints are conditions on all system states and transitions between states, a set of constraints therefore restricts the set of valid system states. In the extreme case, the set of possible system states is empty; in this case the model is called *inconsistent*. Constraints are primarily used to express invariants which are global constraints that hold in every system state and operation specifications in terms of pre- and postconditions that are evaluated locally in the context of an operation call. An operation can only be called if the preconditions hold and must ensure that after execution its postconditions evaluate to true. In general an OCL expression that evaluates to a value of type t is an element of the set Expr_t. The following definitions provide notation for invariants and operation specifications.

Definition 4 (Invariants). *All* invariants *of a class c are contained in the set $\mathcal{I}(c) \subset \mathrm{Expr}_{\mathrm{Boolean}}$. All these Boolean OCL expressions contain a variable* `self` *that is of type c. All invariants of a model are denoted*

$$\mathcal{I}(\mathcal{M}) \stackrel{\mathrm{def}}{=} \bigcup_{c \in \mathrm{CLASS}} \mathcal{I}(c) \ , \tag{6}$$

and as in the notation for system states the $\mathcal{M}$ can be omitted, i.e., we write $\mathcal{I}$, if the use is clear from the context.

Example 3. One invariant for the memory controller model from Fig. 1 is:

```
context Controller
inv uniqueCells: cells->forAll(c1, c2 |
   c1 <> c2 implies c1.address <> c2.address)
```

This invariant states that the cells that are associated to a memory controller must have a unique address.

Definition 5 (Pre- and postconditions). *Given an operation $\omega \in \mathrm{ops}(\mathcal{M})$, the sets $\lhd(\omega) \subseteq \mathrm{Expr}_{\mathrm{Boolean}}$ and $\rhd(\omega) \subseteq \mathrm{Expr}_{\mathrm{Boolean}}$ are the* pre- *and* postconditions *of ω. The notation is borrowed from* [13].

Example 4. The operation specification for the operation *write* of class *Controller* is:

```
context Controller::write(content: Integer)
   pre: address < 10
   pre: content < 4
   post: cells->one(c | c.address = address and c.content = content)
   post: cells->forAll(c | c.address = c.address@pre)
   post: cells->forAll(c | c.address <> address implies
                           c.content = c.content@pre)
```

The two preconditions ensure that a valid address and content have been assigned to the attributes of the *Controller* object. The first postcondition ensures that the cell at the given address has the new content after the operation has been called. The second postcondition ensures that the cell's addresses were not changed by the operation call and the last postcondition ensures that the content of the non-addressed cells is not changed. The model finder cannot *guess* the developer's intention. To ensure that non-related attributes are not changed frame constraints need to be added. Either automatic tools are used that assist the developer in finding them [10] or they are provided manually:

```
post: processor.pc = processor.pc@pre
post: dataout = dataout@pre
post: address = address@pre
```

When evaluating OCL expressions in the context of a system state σ one needs to consider assignments to variables that appear in the OCL expressions. For this purpose, let Var_t be the set of variables of type t, then $\beta : \mathrm{Var}_t \to I(t)$ is a variable assignment. A context for the evaluation of an OCL expression is given by an environment $\tau = (\sigma, \beta)$. Let 'Env' be the set of environments $\tau = (\sigma, \beta)$, then the semantics of an expression $e \in \mathrm{Expr}_t$ is provided by a mapping $I[\![e]\!] : \mathrm{Env} \to I(t)$.

In this paper we make heavy use of the notation that has been introduced in [11] to formalise UML class diagrams and OCL expressions. The meaning of the notation should be comprehensible from the context, however, for more precise definitions the reader is referred to [11].

2.2 Model Finding

Model finding describes the problem of finding a system state σ to a given model $\mathcal{M}$ with invariants $\mathcal{I}$ such that

$$I[\![\mathcal{I}]\!](\sigma) \stackrel{\text{def}}{=} \bigwedge_{c \in \text{CLASS}} \bigwedge_{e \in \mathcal{I}(c)} \bigwedge_{\underline{c} \in \sigma_{\text{CLASS}}(c)} I[\![e]\!]((\sigma, \texttt{self} \mapsto \underline{c})) = \text{true} \ , \tag{7}$$

i.e., all invariants hold for all objects in the system state. Such a system state σ is called *valid* and witnesses the consistency of $\mathcal{M}$. Usually the size of σ_{CLASS} is predefined such that the model finding problem becomes decidable [2]. This assumption is reasonable, since often the size bounds of the model are known in advance. Similar restrictions are assumed for associations, but not discussed in this paper. We refer to this problem also as *structural* model finding since only one system state is considered. Different implementations for the model finding problem have been proposed in the past [7, 14, 3, 9]. The problem can be extended to consider the class diagram's operations which is described in the following. For this purpose, some additional notation to formalise operation calls is required.

An operation call $\underline{\omega} = (\underline{c}, \omega)$ is a tuple consisting of an operation $\omega \in \text{OP}_c$ and an object $\underline{c} \in \sigma_{\text{CLASS}}(c)$. How parameters are bound to operation calls is described later in Sect. 3. For interpreting an operation call, both a pre-state σ_{pre} and a post-state and σ_{post} need to be considered. Consequently, also

two environments $\tau_{\text{pre}} = (\sigma_{\text{pre}}, \beta_{\text{pre}})$ and $\tau_{\text{post}} = (\sigma_{\text{post}}, \beta_{\text{post}})$ are required to formalise the semantics of a postcondition. Then, the interpretation of an operation call $\underline{\omega} = (\underline{c}, \omega)$ is

$$I[\![\underline{\omega}]\!](\sigma, \sigma') \stackrel{\text{def}}{=} \bigwedge_{e \in \triangleleft(\omega)} I[\![e]\!](\tau_{\text{pre}}) \bigwedge_{e \in \triangleright(\omega)} I[\![e]\!](\tau_{\text{pre}}, \tau_{\text{post}}) \ , \tag{8}$$

with $\{\texttt{self} \mapsto \underline{c}\} \in \beta_{\text{pre}}$ and $\{\texttt{self} \mapsto \underline{c}'\} \in \beta_{\text{post}}$ where $\underline{c}'$ refers to $\underline{c}$ in the post-state.

Given a Boolean expression e_{task}, the *behavioural* model finding problem [13] asks to find a set of system states $\sigma_1, \ldots, \sigma_T$ and a set of operation calls $\underline{\omega}_1, \ldots \underline{\omega}_{T-1}$ such that

$$\bigwedge_{t=1}^{T} I[\![\mathcal{I}]\!](\sigma_t) \wedge \bigwedge_{t=1}^{T-1} I[\![\underline{\omega}_t]\!](\sigma_t, \sigma_{t+1}) \wedge e_{\text{task}} = \text{true} \ . \tag{9}$$

Also for this problem, T is usually predefined, since otherwise the problem is undecidable. This assumption is reasonable, since one is interested in a short sequence diagram. The common practice is to increase T in an iterative manner until a solution is found. This is similar to reachability analysis with bounded model checking [4]. Furthermore, the initial state σ_1 is typically preassigned by the developer to exclude false positives, i.e., system states that cannot be reached. Verification tasks in behavioural model checking can, e.g., be the check for a deadlock, i.e., a final system state in which no operation can be called since no preconditions can be asserted, or it can be checked whether an operation is executable by preassigning one of the operation call variables. Besides these, a variety of other reachability tasks can be formulated. Implementations for behavioural model finders have been realised, e.g., in [5, 13].

3 Operation Coverage

In this section, two coverage criteria are defined. Since coverage is witnessed and increased in terms of sequence diagrams, we first need to formalise them analogously to the definitions of the previous section. In this paper, we only consider very simple sequence diagrams with no nested operation calls and no control structures. Consequently, these can also be represented by a sequence of operations.

Definition 6 (Operation sequence). *Given a class diagram $\mathcal{M}$ and a sequence of system states $\sigma_i(\mathcal{M}) = (\sigma_{\text{Class},1}, \sigma_{\text{Att},1})$ for $1 \leq i \leq n$, an* operation sequence *is an ordered set*

$$S = \langle r_1 \leftarrow \underline{\omega}_1(\boldsymbol{p}_1),\ r_2 \leftarrow \underline{\omega}_2(\boldsymbol{p}_2), \ldots,\ r_n \leftarrow \underline{\omega}_n(\boldsymbol{p}_n) \rangle \ . \tag{10}$$

The i^{th} operation call in the sequence S is of the form $r \leftarrow \underline{\omega}(\boldsymbol{p})$ with $\underline{\omega} = (\underline{c}, \omega)$ stating that an operation $\omega : t_c \times t_1 \times \cdots \times t_k \to t \in \text{Op}_c$ is called on an

object $\underline{c} \in \sigma_{\text{CLASS},i}(c)$ *of class* c *with parameters* $\boldsymbol{p} = (p_1, \ldots, p_k)$ *with* $p_i \in t_i$ *and returns a value* $r \in t$*. We define*

$$\operatorname{ops}(S) \stackrel{\text{def}}{=} \{\omega_1, \ldots, \omega_n\} \ . \tag{11}$$

An operation sequence may not necessarily be executable since invariants and operation specifications could prevent the OCL expressions from being asserted.

Definition 7 (Validity of operation sequences). *Operation sequences are executed with respect to some given valid initial system state* σ_1*. An operation sequence is called* valid *if, and only if there exist valid system states* $\sigma_2, \ldots, \sigma_{n+1}$ *such that*

$$\bigwedge_{i=1}^{n} I[\![\underline{\omega}_i]\!](\sigma_i, \sigma_{i+1}) = \text{true} \ . \tag{12}$$

Example 5. Figure 3 shows a valid system state for the running example written in the USE syntax. First, the initial state is prepared which includes object creation and linking. The command `!openter` initiates the operation call and also the preconditions are evaluated. Afterwards, the post-state is prepared before the operation call is finished using `!opexit`, which also evaluates the operation's postcondition.

3.1 Operation call coverage

Now that operation sequences are formally defined, we can formalise the coverage metrics. The first coverage metric, called *operation call coverage*, checks how many of the model's operations have been called in a given set of operation sequences.

Definition 8 (Operation call coverage). *Given a class diagram* $\mathcal{M}$ *and valid operation sequences* $S_1, \ldots, S_n$*, the* operation call coverage *is defined as*

$$\sum_{i=1}^{n} \frac{|\operatorname{ops}(S_i)|}{|\operatorname{ops}(\mathcal{M})|} \ . \tag{13}$$

3.2 Subexpression coverage

The second coverage metric is based on an expression's subexpressions and should first be illustrated by means of an example.

Example 6. The operation *process* from the class *Processor* has among other the following two postconditions:

```
post: pc@pre < 9 implies pc = pc@pre + 1
post: pc@pre = 9 implies pc = 0
```

They ensure that the program counter *pc* overflows to 0 if its value is 9 before the operation is called.

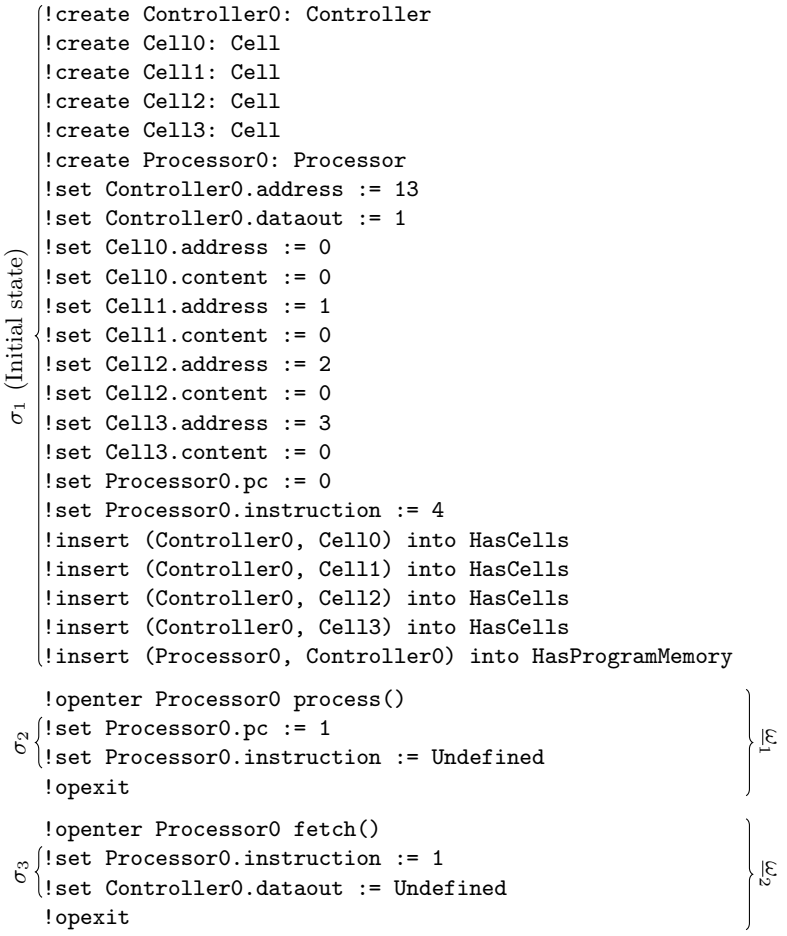

Fig. 3. Valid operation sequence

An implication expression always evaluates to true if the antecedent is false. Hence, one is interested in finding operation sequences that assert many subexpressions. The same argument holds for expressions of the form e_1 `or` e_2. Even expressions of the form e_1 `and` e_2 need to be considered although in this case, both e_1 and e_2 must have been true in order to assert the overall conjunction. However, the expression can be more complicated if nested expressions are used as, e.g., in (e_1 `or` e_2) `and` (e_3 `or` e_4). Also here, it is desired that all subexpressions have been asserted once in some sequence diagram.

For this purpose, we define a function 'sub' that returns all Boolean subexpressions of a given expression $e \in \text{Expr}$:

$$\text{sub}(e) \stackrel{\text{def}}{=} \begin{cases} \{e\} \cup \bigcup\{\text{sub}(e') \mid e' \text{ is subexpression of } e\} & \text{if } e \in \text{Expr}_{\text{Boolean}}, \\ \bigcup\{\text{sub}(e') \mid e' \text{ is subexpression of } e\} & \text{otherwise .} \end{cases} \tag{14}$$

Before we define the second coverage metric, we define the evaluation of a Boolean expression $e \in \text{Expr}_{\text{Boolean}}$ in the context of a valid operation sequence $S = \langle \underline{\omega}_1 = (\underline{c}_1, \omega_1), \ldots, \underline{\omega}_n = (\underline{c}_n, \omega_n) \rangle$ as defined in (10) (For brevity we omitted parameters and return values from the operation sequence). The operation sequence implies $n+1$ valid system states $\sigma_1, \ldots, \sigma_{n+1}$ as defined in (12). The evaluation of e depends on whether it is part of an invariant, a precondition, or a postcondition. If e is part of an invariant, it is checked whether e evaluates to true in some of these system states for some object, i.e.,

$$I_{\text{inv}}[\![e]\!](S) \stackrel{\text{def}}{=} \bigvee_{i=1}^{n} \bigvee_{\underline{c} \in \sigma_{\text{CLASS}}(c)} I[\![e]\!]((\sigma_i, \texttt{self} \mapsto \underline{c})) \ . \tag{15}$$

If e is part of a precondition of some operation ω, it is checked whether it evaluates to true if the operation has been called, i.e.,

$$I_{\text{pre}}[\![e]\!](S) \stackrel{\text{def}}{=} \bigvee_{i=1}^{n} (\omega_i \equiv \omega) \wedge I[\![e]\!]((\sigma_i, \texttt{self} \mapsto \underline{c}_i)) \ . \tag{16}$$

Finally, if e is part of a postcondition of some operation ω, it is checked whether it evaluates to true in the state after the operation has been called, i.e.,

$$I_{\text{post}}[\![e]\!](S) \stackrel{\text{def}}{=} \bigvee_{i=1}^{n} (\omega_i \equiv \omega) \wedge I[\![e]\!]((\sigma_i, \texttt{self} \mapsto \underline{c}_i), (\sigma_{i+1}, \texttt{self} \mapsto \underline{c}'_i)) \ , \tag{17}$$

where $\underline{c}'_i$ refers to $\underline{c}_i$ in state σ_{i+1}.

We also define a partition over all subexpressions in a model based on their context, i.e., whether the expression is part of an invariant or of an operation specification:

$$\text{sub}_{\text{inv}} \stackrel{\text{def}}{=} \bigcup_{e \in \mathcal{I}} \text{sub}(e),$$

$$\text{sub}_{\text{pre}} \stackrel{\text{def}}{=} \bigcup_{\omega \in \text{ops}(\mathcal{M})} \bigcup_{e \in \lhd(\omega)} \text{sub}(e), \text{ and}$$

$$\text{sub}_{\text{post}} \stackrel{\text{def}}{=} \bigcup_{\omega \in \text{ops}(\mathcal{M})} \bigcup_{e \in \rhd(\omega)} \text{sub}(e)$$

Definition 9 (Subexpression coverage). *Given a class diagram $\mathcal{M}$ and valid operation sequences $S_1, \ldots, S_n$, the subexpression coverage is defined as*

$$\frac{\left| \bigcup_{x \in \{\text{inv,pre,post}\}} \{e \in \text{sub}_x \mid \exists S_i \,:\, I_x[\![e]\!](S_i)\} \right|}{|\,\text{sub}_{\text{inv}} \cup \text{sub}_{\text{pre}} \cup \text{sub}_{\text{post}}\,|} \ . \tag{18}$$

The formal definition for subexpression coverage is a bit tedious, however, the intuitive idea can readily be stated in an informal manner. We take all subexpressions from the model's invariants and operation specifications. Then for each of them we check whether they evaluate to true in some intermediate state of the given operation sequences. The number of such subexpressions is divided by the total amount of subexpressions. It is important to ensure that a subexpression of a pre- or postcondition is only checked for a positive evaluation if the respective operation has been called.

Example 7. The operation sequence in Fig. 3 leads to an operation call coverage of 0.4 since 2 out of 5 operations have been called. Furthermore, the subexpression coverage is approximately 0.3 as 14 out of 46 subexpressions are asserted. For instance, the second postcondition of Example 6 is not covered by the operation sequence in Fig. 3 since the program counter *pc* is initially set to 0. Consequently, the subexpression `pc@pre = 9` evaluates to false.

4 Algorithm to Automatically Enhance Coverage

Now that the coverage criteria have been defined, in this section we are proposing methods that aim at automatically increasing the coverage by generating new sequence diagrams. Since model finders are used for this purpose, always the whole solution space is considered. Consequently, the algorithm either yields a sequence diagram that indeed increases the coverage or finds that some operation is not executable or some subexpression cannot be asserted by any sequence diagram. Hence, the algorithm is able to determine "dead" model code (analogously to dead code in programs that can never be executed).

The model finder is utilised by choosing an appropriate expression for e_{task} as described in (9). To increase operation call coverage the model finder is used to find an operation sequence that contains an operation ω that has not been executed thus far. In order to constrain the model finder to call an operation in an operation sequence one needs to assign

$$e_{\text{task}} = \bigvee_{t=1}^{T-1} (\omega_t \equiv \omega) \quad . \tag{19}$$

When increasing subexpression coverage a task needs to be formalised for each subexpression e that is not covered yet. Again, the description of the task expression depends on the origin of the subexpression. If e is part of an invariant of class c, we assign

$$e_{\text{task}} = \bigvee_{t=1}^{T} \bigvee_{\underline{c} \in \sigma_{\text{CLASS}}(c)} I[\![e]\!]((\sigma_t, \texttt{self} \mapsto \underline{c})) \quad . \tag{20}$$

For the case if e is part of an operation specification, we could also find a suitable task expression, however, instead we are making use of a small trick. We

are using the same task expression as in (19) but additionally add e as pre- or postcondition to the considered model. Since the task forces the operation to be called also e must evaluate to true.

It can easily be seen that many sequence diagrams may need to be generated in order to obtain full coverage if initially many operations and subexpressions are uncovered. The first measure to avoid this is to recompute the coverage metrics after each generated sequence diagram, since other uncovered elements may be covered by the newly generated sequence diagram. Furthermore, one can also try to cover multiple uncovered elements at once by combing the task expressions that have been introduced above. As an example one can try to find a sequence diagram in which three uncovered operations ω_1, ω_2, and ω_3 are called by assigning

$$e_{\text{task}} = \bigvee_{t=1}^{T-1} (\omega_t \equiv \omega_1) \wedge \bigvee_{t=1}^{T-1} (\omega_t \equiv \omega_2) \wedge \bigvee_{t=1}^{T-1} (\omega_t \equiv \omega_3) \quad .$$

However, this approach needs to be applied with care since it may lead to false negatives, since operations may be executable independently but not in combination. A strategy can be to first try to generate sequence diagrams that cover a lot of operations and then decrease the number if no more sequence diagrams can be found.

Alternatively, one can make use of Boolean *select variables* $s_1, \ldots, s_\ell$ for each uncovered operation $\omega_1, \ldots, \omega_\ell$. Then these can be considered at once in a single task expression and let the model finder decide which ones to use in the sequence diagram and which ones not:

$$e_{\text{task}} = \bigwedge_{i=1}^{\ell} \left(s_i \Rightarrow \bigvee_{t=1}^{T-1} (\omega_t \equiv \omega_i) \right) \wedge \sum_{i=1}^{\ell} s_i \geq k \quad . \tag{21}$$

If a select variable s_i is assigned 1, then the operation ω_i must be called in the sequence diagram. Since e_{task} can easily be satisfied by assigning all select variables to 0, a cardinality constraint ensures that at least k select variables must be assigned 1 and hence at least k operations must be called in the operation sequence. The value for k can be initially set high and then decreased successively if e_{task} cannot be satisfied.

5 Tool Support

We implemented the proposed algorithm as a plugin of the UML specification tool USE [6]. Consequently, models in the form of class diagrams as well as sequence diagrams are to be provided in the USE format (`.use` and `.cmd` to specify class diagrams and operation sequences, respectively). We have used the SMT-based behavioural model finder that has been proposed in [13].

The features realised in the plugin are the following:[4]

[4] A USE plugin for computing and displaying coverage information can be downloaded from `www.informatik.uni-bremen.de/agra/files/coverage-plugin.zip`

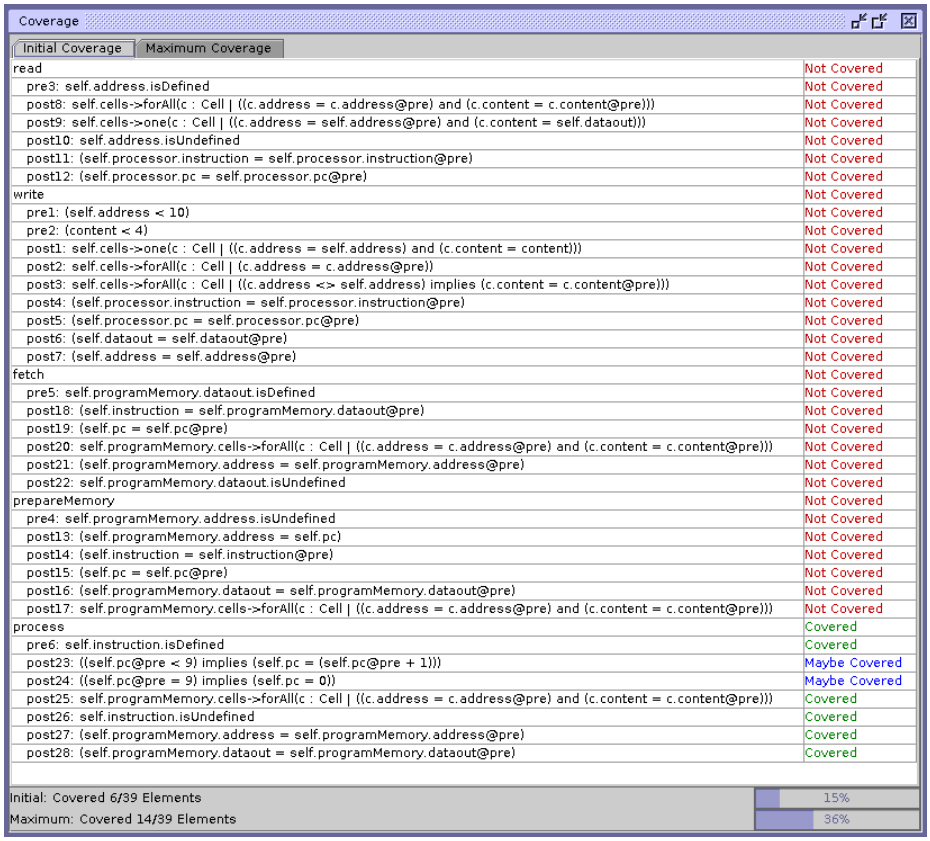

Fig. 4. Initial coverage

1. computation of the initial coverage of operations and constraints based on the provided sequence diagram(s) (Fig. 4)
2. computation of the maximal possible coverage of operations and constraints using a model finder in the background (Fig. 5)
3. display of the results

Both types of coverage are computed upon start of the plugin in the background if a model is provided. Without given sequence diagrams, the initial status of each operation and each constraint is displayed as *not covered*. If coverage of the respective operation or constraint has been reached, the mark is changed to *covered*. The remaining constraints are marked as *maybe covered* in the initial computation and as *partially covered* in the final state.

A constraint is *maybe covered* if the respective operation has been executed but the constraint contains subexpressions which might not hold, e.g., a part of a disjunction where the whole constraint can evaluate to true while a single subexpression evaluates to false. A constraint is eventually *partially covered* when

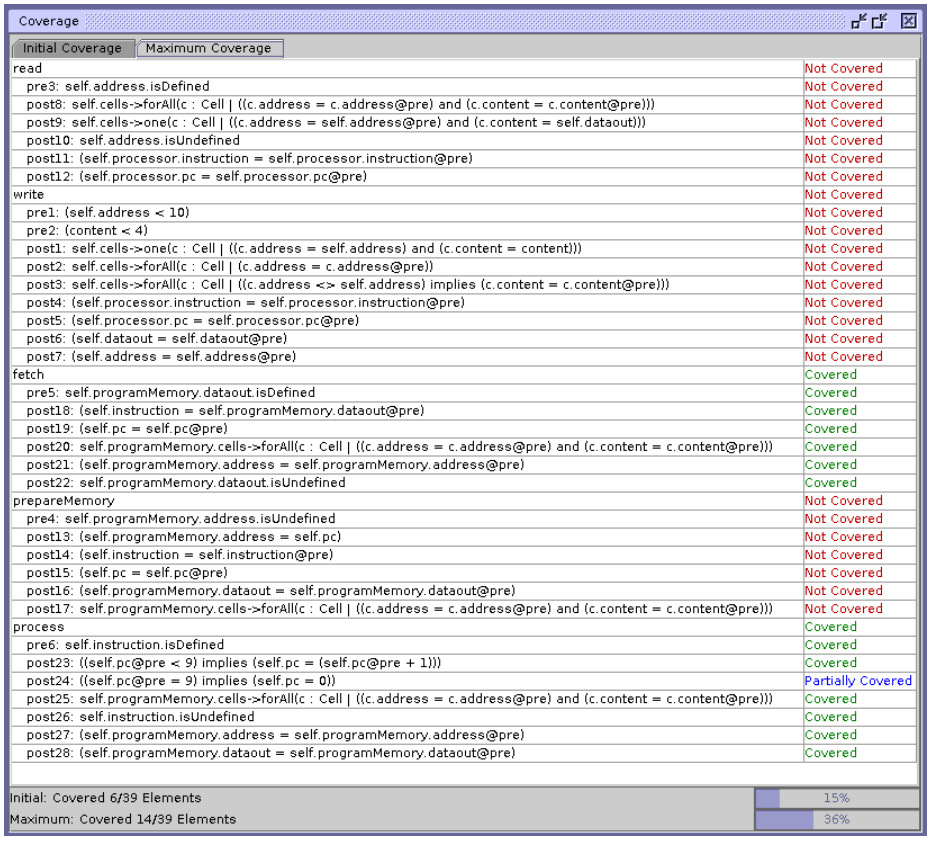

Fig. 5. Maximised coverage

it was not possible to find an operation sequence such that all subexpressions eventually hold, i.e., one part of the constraint is covered and one part of it is not covered.

During computation of the maximum coverage, sequence diagrams for the not yet covered operations and constraints are produced and printed to the USE shell. The overall results of the coverage enhancement are always displayed at the bottom of the window. Here, the amount of initially and finally covered elements are provided as well as a progress bar depicting the coverage percentage.

5.1 Experimental Evaluation

For an experimental evaluation, the approach has been applied to several models which have been provided with the tool USE or have been written by the authors. Table 1 shows the results of said evaluation. The first column gives the names of the models. In the second and third column, their initial and maximal coverage is

stated. Then, the amount of generated sequences is provided and the last column contains the required run-times.

In all cases except for one, an increase in coverage up to 94–100% could be achieved. Only for the test case *CPU*, the initial coverage of 0% remained unchanged. This means that no operation sequences could be generated and, consequently, none of the constraints were triggered. This scenario may occur due to two reasons: (1) The initial state may be chosen poorly so that no operation's pre-condition evaluates to true. (2) The post-conditions of the operations may be contradictory, preventing the operation's execution even if a pre-condition is satisfied. In both cases, the model as well as the initial state have to be revised.

With regard to the run-time, it can be stated that only the test case *Memory* required a slightly longer execution time than the remaining examples. Since this example is by far the largest one in terms of OCL constraints and has a relatively low initial coverage of 15%, a slight increase in run-time was to be expected.

Overall, it was possible to reach high coverage percentages in negligible run-time by generating a maximum of 6 operation call sequences. In two cases, constraints and/or operations could be detected thanks to our approach which could not be covered by sequences starting in the initial state. These constraints/operations would result in dead code, so by uncovering them, the quality of the resulting implementation can be improved.

6 Related Work

In [12] coverage criteria are defined based on sequence diagrams which are extracted by reverse engineering of existing Java source code. Branches in the source code are mapped to guarded messages in the sequence diagram and therefore each sequence diagram defines a set of possible execution paths. Coverage criteria are based on these paths. The authors have not provided methods to automatically increase the coverage criteria.

The authors of [1] propose three test coverage criteria for class diagrams. These criteria specify a certain structure of an object diagram which has to be created by a test case in order to reach full coverage. In two cases, this structure is determined by constructing representative values for association-end multiplicities and attributes. The third criterion considers generalisation relationships. All three criteria do not consider behavioural aspects and only one criterion takes OCL constraints into consideration.

Table 1. Experimental results

Model	Initial	Maximal	#Sequences	Run-time
CPU	0%	0%	0	<0.01s
Traffic	35%	94%	4	<0.01s
Memory	15%	97%	6	0.22s
Car	0%	100%	4	<0.01s
Life	0%	100%	3	<0.01s

Scenario-based design analysis (SUDA) is defined in [16]. The authors introduce snapshot models to transform a class model with operations into a static model of behaviour which can be verified against a sequence of operation calls. Based on this technique an automatic approach is presented in [17]. Given a UML class diagram with OCL operation specifications and invariants and a specification for a desired scenario, a scenario is automatically generated using model finding techniques.

In [8] a technique is presented that allows animation of UML class diagrams with OCL operation specifications and invariants. Given a current state, a post-state is computed that satisfies the class diagram's invariants and underspecified postconditions.

7 Conclusions

We concerned ourselves with the question of how to formalise coverage criteria of class diagrams with respect to a set of sequence diagrams in terms of operation sequences. We have defined two coverage criteria, one which checks how many operations are called and another one which considers whether all subexpressions in the involved OCL constraints evaluate to true. Based on the formalisation of UML class diagrams and OCL expressions introduced in [11] we formalised the coverage criteria to enable their application in formal analysis.

We demonstrated how model finders can be utilised in order to automatically generate new sequence diagrams that increase overall coverage of the class diagram. By using the model finder the developer is also pinpointed to operations and OCL subexpressions that can never be executed or asserted, respectively. Our algorithm has been implemented in the UML specification tool USE.

It is a well-known fact that model finders cannot be applied to arbitrarily large models and face scalability problems as the number of classes and constraints increases. Consequently, the efficiency of our approach for automatically generating sequence diagrams heavily depends on the efficiency of the model finder. For now, we have evaluated the generation approach to class diagrams similar to the running example of the paper. For these, sequence diagrams can be generated within a few seconds. In future work we want to evaluate the scalability of the approach in more detail by comparing different model finders. Alternatively, model finders can be tuned to perform well for these kind of problem. Furthermore, more advanced sequence diagrams, e.g., with nested operations, will enhance the usability of the approach.

References

1. Andrews, A., France, R.B., Ghosh, S., Craig, G.: Test adequacy criteria for UML design models. Software Testing, Verification and Reliability 13, 95–127 (2003)
2. Berardi, D., Calvanese, D., De Giacomo, G.: Reasoning on UML class diagrams. Artificial Intelligence 168(1–2), 70–118 (2005)

3. Cabot, J., Clarisó, R., Riera, D.: Verification of UML/OCL class diagrams using constraint programming. In: Int'l. Conference on Software Testing Verification and Validation Workshop. pp. 73–80. IEEE (2008)
4. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
5. Frias, M.F., Galeotti, J.P., Pombo, C.L., Aguirre, N.: DynAlloy: upgrading Alloy with actions. In: Int'l Conf. on Software Engineering. pp. 442–451. ACM (2005)
6. Gogolla, M., Büttner, F., Richters, M.: USE: a UML-based specification environment for validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)
7. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
8. Krieger, M.P., Knapp, A.: Executing underspecified OCL operation contracts with a SAT solver. Electronic Communication of the European Association of Software Science and Technology 15 (2008)
9. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive validation of OCL models by integrating SAT solving into USE. In: Int'l Conf. on Objects, Models, Components, Patterns. Lecture Notes in Computer Science, vol. 6705, pp. 290–306. Springer (2011)
10. Niemann, P., Hilken, F., Gogolla, M., Wille, R.: Assisted generation of frame conditions for formal models. IEEE (2015)
11. Richters, M.: A Precise Approach to Validating UML Models and OCL Constraints. Ph.D. thesis, University of Bremen, Logos Verlag, Berlin, BISS Monographs, No. 1 (2002)
12. Rountev, A., Kagan, S., Sawin, J.: Coverage criteria for testing of object interactions in sequence diagrams. In: Int'l Conf. on Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science, vol. 3442, pp. 289–304. Springer (2005)
13. Soeken, M., Wille, R., Drechsler, R.: Verifying dynamic aspects of UML models. In: Design, Automation and Test in Europe. pp. 1077–1082. IEEE (2011)
14. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using Boolean satisfiability. In: Design, Automation and Test in Europe. pp. 1341–1344. IEEE (2010)
15. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley Longman (1999)
16. Yu, L., France, R.B., Ray, I.: Scenario-based static analysis of UML class models. In: Int'l Conf. on Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science, vol. 5301, pp. 234–248. Springer (2008)
17. Yu, L., France, R.B., Ray, I., Sun, W.: Systematic scenario-based analysis of UML design class models. In: Int'l Conf. on Engineering of Complex Computer Systems. pp. 86–95. IEEE Computer Society (2012)