Verifying Action Semantics Specifications in UML Behavioral Models (Extended Version)

Elena Planas[1], Jordi Cabot[1] and Cristina Gómez[2]

[1] Estudis d'Informàtica, Multimèdia i Telecomunicacions, Universitat Oberta de Catalunya
{eplanash | jcabot}@uoc.edu
[2] Dept. de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya
cristina@lsi.upc.edu

Abstract. MDD and MDA approaches require capturing the behavior of UML models in sufficient detail so that the models can be automatically implemented/executed in the production environment. With this purpose, Action Semantics (AS) were added to the UML specification as the fundamental unit of behavior specification. Actions are the basis for defining the fine-grained behavior of operations, activity diagrams, interaction diagrams and state machines. Unfortunately, current proposals devoted to the verification of behavioral schemas tend to skip the analysis of the actions they may include. The main goal of this paper is to cover this gap by presenting several techniques aimed at verifying AS specifications. Our techniques are based on the static analysis of the dependencies between the different actions included in the behavioral schema. For incorrect specifications, our method returns a meaningful feedback that helps repairing the inconsistency.

1 Introduction

One of the most challenging and long-standing goals in software engineering is the complete and automatic implementation of software systems from their initial high-level models [21]. This is also the focus of current MDD (Model-driven development) and MDA (Model-driven architecture) approaches.

Recently, the OMG itself has issued a RFP for the "Foundational Subset for Executable UML Models" [20], with the goal of reducing the expressivity of the UML to a subset that can be directly executable [17]. A key element in all executable UML methods is the use of Action Semantics (AS) to specify the fine-grained behavior of all behavioral elements in the model. Actions are the fundamental unit of behavior specifications. Their resolution and expressive power are comparable to the executable instructions in traditional programming languages. Higher-level behavioral formalisms of UML (as operations, activity diagrams, state machines and interactions diagrams) are defined as an additional layer on top of the predefined set of basic actions (e.g. creation of new objects, removals of existing objects, among others) [19].

Given the important role that actions play in the specification of the behavioral aspects of a software system, it is clear that their correctness has a direct effect on the quality of the final system implementation. As an example, consider the class diagram

of Fig. 1.1 including the operations *changeAddress* and *addPerson*. Both operations are incorrect, since *changeAddress* tries to update an attribute which does not even exist in the class diagram and *addPerson* can never be successfully executed (i.e. every time we try to execute *addPerson* the new system state violates the minimum '1' cardinality constraint of the *department* role in *WorksIn*, since the created person instance *p* is not linked to any department). Besides, this operation set is not complete, i.e. through these operations users cannot modify all elements of the class diagram, e.g. it is not possible to create and destroy departments. These errors must be fixed before attempting to generate the system implementation.

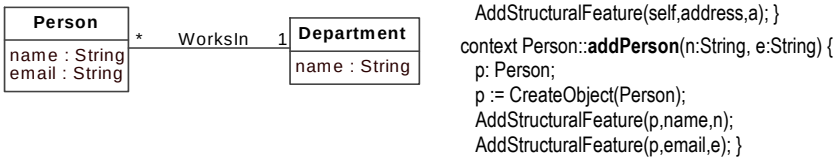

Fig. 1.1. A simple example of a class diagram with two operations.

The main goal of this paper is to provide a set of lightweight techniques for the verification of correctness properties (*syntactic correctness*, *weak executability, completeness* and *redundancy*) of action-based behavior specifications at design time. Due to space limitations, we will focus on the verification of AS specifications used to define the effect of the operations included in the class diagram (as the example above). Nevertheless, the techniques presented herein could be similarly used to verify action sequences appearing in other kinds of UML behavior specifications.

Roughly, given an operation *op*, our method (see Fig. 1.2) proceeds by first, analyzing the syntactic correctness of each action $ac \in op$. Then, the method analyzes *op* to determine all its possible execution paths. Executability of each path *p* is determined by performing a static analysis of the dependencies among the actions in *p* and their relationship with the structural constraints (as cardinality constraints) in the class diagram. Next, our method analyses the completeness of the whole operation set, as well as possible redundancies. For each detected error, possible corrective procedures are suggested to the designer as a complementary feedback. After our initial analysis, model-checking based techniques could also be used to get more information (e.g. incorrect execution traces) on the operations.

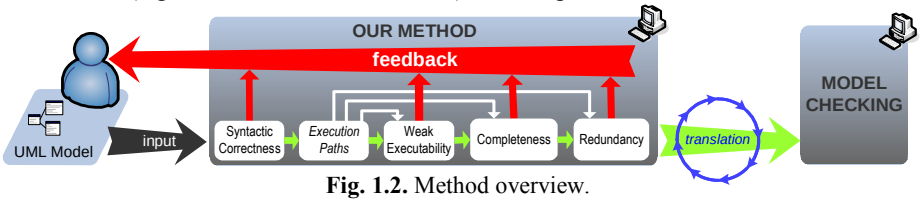

Fig. 1.2. Method overview.

The rest of the paper is structured as follows. The next section describes basic AS concepts. Section 3 focuses on the operations' syntactic correctness. Section 4 explains how to determine the different execution paths in an operation and Section 5 determines their executability. Sections 6 and 7 study the completeness and the redundancies of the operations, respectively. In Section 8, we compare our method with the related work and, in Section 9, we present the conclusions and further work.

2 Action Semantics in the UML

The UML standard [19] defines the actions that can be used in behavioral specifications. In this paper, we will focus on the following write actions[1] (actions that modify the system state) since they are the ones that can compromise the system consistency:

1. *CreateObject(class:Classifier):InstanceSpecification*: Creates a new object that conforms to the specified classifier. The object is returned as an output parameter.
2. *DestroyObject(o:InstanceSpecification)*: Destroys the object *o*. We assume that links in which *o* participates are not automatically destroyed.
3. *AddStructuralFeature(o:InstanceSpecification, at:StructuralFeature, v: ValueSpecification)*: Sets the value *v* as the new value for the attribute *at* of the object *o*. We assume that multi-valued attributes are expressed (and analyzed) as binary associations between the class and the attribute data type.
4. *CreateLink(as:Association, p_1:Property, o_1:InstanceSpecification, p_2:Property, o_2:InstanceSpecification)*: Creates a new link in the binary association *as* between objects o_1 and o_2, playing the roles p_1 and p_2, respectively.
5. *DestroyLink(as:Association, p_1:Property, o_1:InstanceSpecification, p_2:Property, o_2:InstanceSpecification)*: Destroys the link between objects o_1 and o_2 from *as*.
6. *ReclassifyObject(o:InstanceSpecification, newClass:Classifier[0..*], oldClass: Classifier[0..*])*: Adds *o* as a new instance of classes in *newClass* and removes it from classes in *oldClass*. We consider that classes in *newClass* may only be direct superclasses or subclasses of classes in *oldClass*.
7. *CallOperation(op:Operation, o:InstanceSpecification, arguments: List(LiteralSpecification)): List(LiteralSpecification)*: Invokes *op* on *o* with the *arguments* values and returns the results of the invocation.

These actions can be accompanied with several read actions (e.g. to access the values of attributes and links of the objects). Read actions do not require further treatment since they do not affect the correctness properties we define in this paper.

Additionally, UML defines that actions can be structured to coordinate basic actions in action sequences, conditional blocks or loops (*do-while* or *while-do* loops).

As an example, we have defined three operations: *endOfReview*, *submitPaper* and *dismiss* (Fig. 2.2) for the class diagram of Fig. 2.1, aimed at representing part of a conference management system. The first operation reclassifies a paper as rejected or accepted depending on the *evaluation* parameter. The second one creates a new "under review" paper and links the paper with its authors. The last one deletes the *WorksIn* link between a person and his/her department.

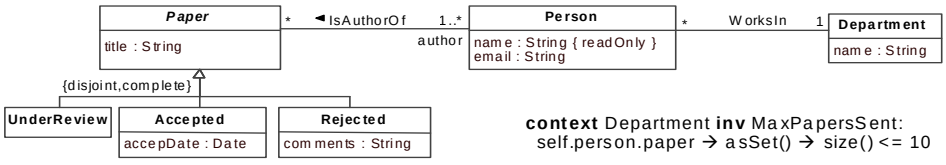

Fig. 2.1. Excerpt of a conference management system class diagram.

[1] UML provides an abstract syntax for these actions [19]. Our concrete syntax is based on the names of the action metaclasses. For structured nodes we will use an ASL-based syntax [17].

```
context Paper::endOfReview(com:String,d:Date,
evaluation:String) {
 if self.oclIsTypeOf(UnderReview) then
  if evaluation = 'reject' then
   ReclassifyObject(self,[Rejected],[ ]);
   AddStructuralFeature(self,comments,com);
  else
   ReclassifyObject(self,[Accepted],[ ]);
   AddStructuralFeature(self,accepDate,d);
  endif
endif }
```

```
context Paper::submitPaper(tit:String, authors:Person[1..*]) {
 i: Integer := 1;
 p: Paper;
 p := CreateObject(UnderReview);
 AddStructuralFeature(p,title,tit);
 while i <= authors->size() do
  CreateLink(IsAuthorOf,author,authors[i],paper,p);
  i := i+1;
endwhile }
```

```
context Person::dismiss() {
 DestroyLink(WorksIn,person,self,department,self.department); }
```

Fig. 2.2. Specification of *endOfReview*, *submitPaper* and *dismiss* operations.

3 Syntactic Correctness

The UML metamodel includes a set of constraints (i.e. well-formedness rules (WFRs)) that restrict the possible set of valid (or well-formed) UML models. Some of these WFRs are aimed at preventing syntactic errors in action specifications. For instance, when specifying a *CreateLink* action *ac* over an association *as*, the WFRs ensure that the type and number of the input objects in *ac* are compatible with the set of association ends defined for *as*.

An operation is syntactically correct when all the actions included in the operation satisfy the WFRs. Unfortunately, our analysis of the WFRs relevant to the Action Packages has revealed several flaws (see the detected errors in Appendix A). Besides, several required WFRs are missing. For instance, in actions of type *WriteStructuralFeature* we should check that the type of the input object (i.e. the object to be modified) is compatible with the classifier owning the feature (in OCL: *context WriteStructuralFeature inv: self.value.type = self.structuralFeature.type*). Also, in *CreateObject*, the input classifier cannot be the supertype of a covering generalization set (in a covering generalization, instances of the supertype cannot be directly created). Similar WFRs must be defined to restrict the possible *newClassifiers* in the *ReclassifyObject*. For instance, we should check that the *newClassifiers* set and the *oldClassifiers* set are disjoint sets. Additional rules are needed to check that values of *readOnly* attributes are not updated after their initial value has been assigned and so forth. These WFRs must be added to the UML metamodel to ensure the syntactic correctness of action specifications.

After this initial syntactic analysis, we proceed next with a more semantic verification process that relates the specified actions with other model elements.

4 Computing the Execution Paths

The correctness properties that will be presented in the next sections are based on an analysis of the possible *execution paths* allowed by the structured group of actions that define the operation effect. An execution path is a sequence of actions that may be

followed during the operation execution. For trivial groups of actions (e.g. with neither conditional nor loop nodes) there is a single execution path but, in general, several ones will exist.

To compute the execution paths, we propose to represent the actions in the operation as a model-based control flow graph (MBCFG), that is, a control flow graph based on the model information instead of on the program code, as traditional control flow graph proposals. MBCFGs have been used to express UML sequence diagrams [9]. Here we adapt this idea to express the control flow of action-based operations.

For the sake of simplicity, we will assume that the group of actions defining the operation behavior is defined as a structured *SequenceNode* (see the metamodel excerpt in Fig. 4.1) containing an ordered set of *ExecutableNodes*, where each executable node can be either one of the basic modification actions described in Section 2 (other types of actions are skipped since they do not affect the result of our analysis), a *ConditionalNode*, a *LoopNode* or an additional nested *SequenceNode*. We also use two "fake" nodes, an initial node (representing the first instruction in the operation) and a final node (representing the last one). These two nodes do not change the operation effect but help in simplifying the presentation of our MBCFG.

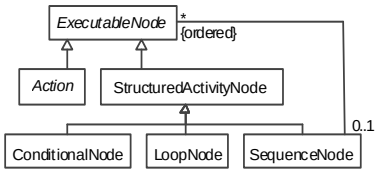

Fig. 4.1. Fragment of UML metamodel.

The digraph $MBCFG_{op} = (V_{op}, A_{op})$ for an operation *op* is obtained as follows:

- Every executable node in *op* is a vertex in V_{op}.
- An arc from an action vertex v_1 to v_2 is created in A_{op} if v_1 immediately precedes v_2 in an ordered sequence of nodes.
- A vertex v representing a conditional node n is linked to the vertices $v_1 \ldots v_n$ representing the first executable node for each *clause* (i.e. the *then* clause, the *else* clause,…) in n. The last vertex in each clause is linked to the vertex v_{next} immediately following n in the sequence of executable nodes. If n does not includes an *else* clause, an arc between v and v_{next} is also added to A_{op}.
- A vertex v representing a loop node n, is linked to the vertex representing the first executable node for *n.bodyPart* (returning the list of actions in the body of the loop) and to the vertex v_{next} immediately following n in the node sequence. The last vertex in *n.bodyPart* is linked back to v (to represent the loop behavior).
- A vertex representing an *OperationCall* action is replaced by the sub-digraph corresponding to the called operation c like follows: (1) the initial node of c is connected with the node that precedes the *OperationCall* node in the main operation (2) the final node of c is connected with the node/s that follow the *OperationCall* node and (3) the parameters of c are replaced by the arguments in the call.

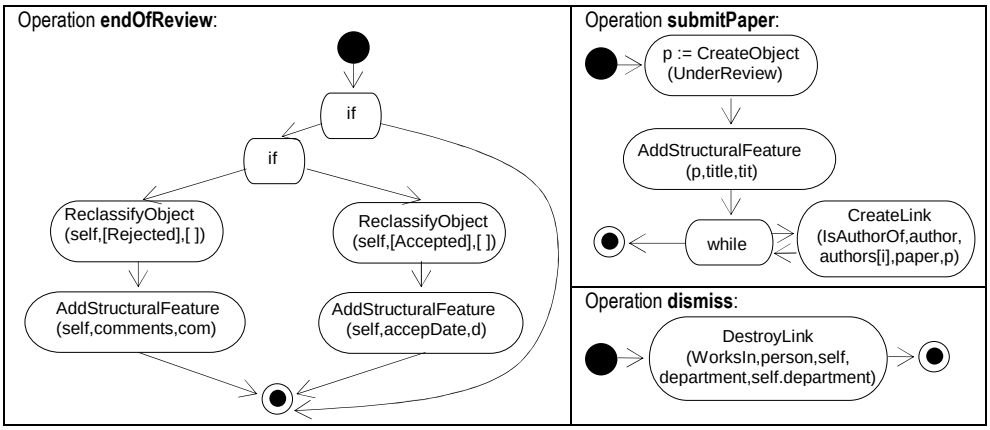

Fig. 4.2. MBCFG of *endOfReview, submitPaper* and *dismiss* operations for the example.

Fig. 4.2 shows the MBCFGs for the operations in Fig. 2.2. Test conditions of conditional and loop nodes are not shown since they are not part of our analysis[2].

Given a $MBCFG_{op}$ graph G, the set of execution paths ex_{op} for *op* is defined as $ex_{op}=allPaths(MBCFG_{op})$ where *allPaths(G)* returns the set of all paths in G that start at the initial vertex (the vertex corresponding to the initial node), end at the final node and does not include repeated arcs (these paths are also known as *trails* [2]).

Each path in ex_{op} is formally represented as a sequence of <*number,action*> node tuples where *number* indicates the number of times that the action *action* is executed in that node. Vertices representing other types of executable nodes are discarded.

The *number* in the tuple is only relevant for actions included in loop nodes. For other actions the number value is always '1'. For an action *ac* within a loop, *number* is computed as follows: (1) each *while-do* loop in the graph is assigned a different variable *N,…,Z* representing the number of times the loop may be executed. *Do-while* loops are assigned the value *1+N,…,1+Z* to express that the body is executed at least once and (2) the *number* of *ac* is defined as the multiplication of the variable values of all loop nodes we find in the path between *ac* and the initial vertex, i.e. *ac* will be executed *N* times if *ac* is in a top-level loop, *N*M* if *ac* is part of a single nested loop, and so forth. Fig. 4.3 shows the execution paths for the graphs in Fig. 4.2.

endOfReview:
p1 = ø
p2 = [<1,ReclassifyObject(self,[Rejected],[ ])>, <1,AddStructuralFeature(self,comments,com)>]
p3 = [<1,ReclassifyObject(self,[Accepted],[ ])>, <1,AddStructuralFeature(self,accepDate,d)>]

submitPaper:
p = [<1,p:=CreateObject(UnderReview)>, <1,AddStructuralFeature(p,title,tit)>,
<N,CreateLink(IsAuthorOf,author,authors[i],paper,p)>]

dismiss:
p = [<1,DestroyLink(WorksIn,person,self,department,self.department)>]

Fig 4.3. Execution paths of *endOfReview, submitPaper* and *dismiss* operations.

[2] Detection of infeasible paths due to unsatisfiable tests conditions is out of scope of this paper. This SAT-problem could be tackled with UML/OCL verification tools [3] adding the test condition as an additional constraint and checking if the extended model is still satisfiable.

5 Weak Executability

An operation is *weakly executable* when there is a chance that a user may successfully execute the operation, that is, when there is at least an initial system state and a set of arguments for the operation parameters for which the execution of the actions included in the operation evolves the initial state to a new system state that satisfies all integrity constraints. Otherwise, the operation is completely useless: every time a user tries to execute the operation (and regardless of the input values provided to the operation) an error will arise because some integrity constraint will become violated. We define our *executability* property as *weak executability* since we do not require all executions of the operation to be successful, which could be defined as *strong executability*. Obviously, weak executability is a prerequisite for strong executability. So, designers could check first our weak executability and then, if they think it is necessary, they could apply other techniques (see the related work) to determine the stronger property.

As an example, consider again the operations of Fig. 2.2. Clearly, *dismiss* is not executable since every time we try to delete a link between a person *p* and a department *d*, we reach an erroneous system state where *p* has no related department, a situation forbidden by the minimum '1' multiplicity in the *WorksIn* association. As we will see later, in order to dismiss *p* from *d* we need to either assign a new department *d'* to *p* or to remove *p* itself within the same operation execution. Instead, *submitPaper* is weakly executable since we are able to find an execution scenario where the new paper can be successfully submitted (e.g. when submitting a paper whose authors belong to a department that has not previously submitted any other paper). Note that, as discussed above, classifying *submitPaper* as weakly executable does not mean that every time this operation is executed the new system state will be consistent with the constraints. For instance, if a person *p* passed as a value for the *authors* parameter belong to a department with already 10 submissions, then, the operation execution will fail because the constraint *MaxPapersSent* will not be satisfied by the system state at the end of the operation execution.

The weak executability of an operation is defined in terms of the weak executability of its execution paths: an operation is weakly executable if at least one of its paths is weakly executable[3]. Executability of a path *p* depends on the set of actions included in the path. The basic idea is that some actions require the presence of other actions within the same execution path in order to leave the system in a consistent state at the end of the execution. Therefore, to be executable, a path *p* must satisfy all action dependencies for every action *ac* in *p*. Dependencies for a particular action are drawn from the structure and constraints of the class diagram and from the kind of modification performed by the action type. For example, the *dismiss* operation is not weakly executable because its single path (see Fig.4.3) is not executable since the action *DestroyLink(WorksIn,person,p,department,d)* must be always followed by *CreateLink(WorksIn,person,p,department,d')* or *DestroyObject(p)* to avoid violating

[3] It is also important to detect and repair all non-executable paths. Otherwise, all executions of the operation that follow one of those paths will irremediably fail.

the minimum multiplicity. The single path includes none of these actions and thus it is not executable.

To determine if a path *p* is weakly executable, we proceed by (1) computing the action dependencies for each action in *p* and (2) checking that those dependencies are satisfied in *p*. If all dependencies are satisfied, then, we may conclude that *p* is weakly executable. In the following, we explain in detail these two steps and provide an algorithm that combines them to determine the executability of a path.

5.1 Computing the Dependencies

A dependency from an action ac_1 (the *depender* action) to an action ac_2 (the *dependee*) expresses that ac_2 must be included in all execution paths where ac_1 appears to avoid violating the constraints of the class diagram. It may happen that ac_1 depends on several actions (AND-composition) or that we have different alternatives to keep the consistency of the system after executing ac_1 (OR-composition; as long as one of the possible dependee actions appears in the path, the dependency is satisfied).

Table 5.1 provides the rules to compute the dependencies for each kind of action, linked with the AND and OR operators, if necessary. These rules are adapted from [4]. The third column (*Shareable*) determines, for each dependency, if two or more dependee actions can be mapped (i.e. share) to the same depender action in the path.

As an example, according to the table 5.1, a *CreateLink* action needs (when the rule condition is true) a *DestroyLink*, a *CreateObject* or a *ReclassifyObject* action in the same execution path. The first dependency is not shareable, since each *CreateLink* needs a different *DestroyLink* to keep the system consistent. Instead, the alternative dependency *CreateObject* (*ReclassifyObject*) is shareable since several create links may rely on the same new (reclassified) object to satisfy the cardinality constraints.

Note that, to determine the dependencies we just take into account cardinality constraints and disjoint and complete generalization constraints. Other constraints do not affect the weak executability property, since we can always find a system state and/or a set of arguments for which the execution of an action results in a consistent state with respect to those constraints. For instance, constraints restricting the value of the attributes of an object may be satisfied when passing the appropriate arguments as parameters for the action (and similarly with constraints restricting the relationship between an object and related objects). As seen before, *MaxPapersSent* constraint (Fig. 2.1) does not affect the weak executability of *submitPaper*. It certainly restricts the set of people that may be passed as authors for the submitted paper but it is easy to see that there are many system states (and many possible values for the *authors* parameter) over which the operation can be successfully executed.

Table 5.1. Dependencies for modification actions. *Min(c_i,as)* and *max(c_i,as)* denote the minimum (maximum) multiplicity of c_i in *as* (for reflexive associations we use the role name).

Depender Action	Dependee Actions	Share-able
o := CreateObject(c)	*AddStructuralFeature(o,at,v)* for each non-derived and mandatory attribute at of *c* or of a superclass of *c*	No
	AND <*min(c,as),CreateLink(as,p,o,p_2,o_2)*> for each non-derived association *as* where *c* or a superclass of *c* has mandatory participation	No
DestroyObject(o:c)	<*min(c,as),DestroyLink(as,p,o,p_2,o_2)*> for each non-derived as where *c* or a superclass of *c* has a mandatory participation	No
CreateLink(as,p_1,o_1:c_1, p_2,o_2:c_2) (when *min(c_1,as) = max(c_1,as)*) to be repeated for the other end	*DestroyLink(as,p_1,o_1,p_3,,o_3)* (if *min(c_2,as) <> max(c_2,as)*)	No
	OR *CreateObject(o_1)*	Yes
	OR *ReclassifyObject(o_1,[c_1],[])*	Yes
DestroyLink(as,o_1:c_1,o_2:c_2) (when *min(c_1,as) = max(c_1,as)*) to be repeated for the other end	*CreateLink(as,p_1,o_1,p_3,,o_3)* (if *min(c_2,as) <> max(c_2,as)*)	No
	OR *DestroyObject(o_1)*	Yes
	OR *ReclassifyObject(o_1,[],[c_1])*	Yes
AddStructuralFeature(o,at,v)	-	-
ReclassifyObject(o,[nc],[oc])	*AddStructuralFeature(o,at,v)* for each non-derived and mandatory attribute at of each class *c* ∈ *nc*	No
	AND <*min(c,as),CreateLink(as,p,o,p_3,o_3)*> for each *c* ∈ *nc* and for each non-derived association *as* where *c* has a mandatory participation	No
	AND {*ReclassifyObject(o,[],[c_1])* **OR** *ReclassifyObject(o,[],[c_n])*} for each *c* ∈ *nc* such that *c* is a subclass in a disjoint and complete generalization *G(superclass, c, c_1,…,c_n)* and not ∃ i | c_i ∈ *nc*	Yes
	AND <*min(c,as),DestroyLink(as,p,o,p_3,o_3)*> for each *c* in *oc* and for each non-derived association *as* where *c* has a mandatory participation	No
	AND {*ReclassifyObject(o,[c_1],[])* **OR** *…ReclassifyObject(o,[c_n],[])*} for each *c* ∈ *oc* such that *c* is a subclass in a disjoint and complete generalization *G(superclass, c, c_1,…c_n)* and not ∃ i | c_i ∈ *oc*	Yes

5.2 Mapping the Dependencies

Each single dependency *d*=<*number,action*> computed for a path must be satisfied. Otherwise, *d* must be returned as a feedback to the user to help him/her to repair the inconsistency. A dependency is satisfied if it can be successfully mapped to one of the actions in the path.

A dependency *d* can be mapped onto a node *n* in the path when the following conditions are satisfied: (1) *d.action* and *n.action* are the same (e.g. both are *CreateLink* actions), (2) the model elements referenced by the actions coincide (e.g. both create new links for the same association), (3) all instance-level parameters of *d.action* can be bound to the parameters in *n.action* (free variables introduced by the rules may be bound to any parameter value in *n.action*, while fixed ones must have the same identifier in *d* and *n)* and (4) $d.number \leq 1$ (for actions that are shareable) or $d.number \leq n.number$ (for non-shareable actions). This comparison may include positive integer abstract variables (when *n* is part of a loop, see Section 4). In those cases, *d* can be mapped iff there is a possible instantiation of the abstract variables that satisfies the inequality comparison $d.number\text{-}n.number \geq 0$. This can be easily expressed (and solved) as a constraint satisfaction problem [16].

5.3 Algorithm to Determine the Weak Executability of a Path

In the following, we present an algorithm for determining the weak executability of an execution path *path* on a class diagram *cd*. For non-executable paths, the algorithm returns a set of possible repair action alternatives (output parameter *requiredActions*) that could be included in the path to make it executable[4].

```
function weakExecutability (
in: path: List(<number:Integer,action:Action>),
in: cd: <Set(Class),Set(StructuralFeature),Set(Association),
Set(GeneralizationSet),Set(Constraint)>,
out: requiredActions: Set(List<number:Integer,action:Action>)): Boolean
{  node: <number:Integer,action:Action>;
   depLists: Set(List<number:Integer,action:Action>):=ø;
   //Loop 1: Computing the dependencies
   i: Integer:=1;
   while i ≤ path->size() do
    node:=getNode(path,i);
    updateDependencies(node,cd,depLists); i:=i+1;
   endwhile
   //Loop 2: Determining the required actions
   executable: Boolean:=false; i:=1;
   while i ≤ depLists->size() and ¬executable do
    requiredActions[i]:=mapping(depLists[i],path);
    if (requiredActions[i] = ø) then executable:=true;
    else i:=i+1;
    endif
   endwhile
   return executable;}
```

[4] Extending the path with this sequence is a necessary condition but not a sufficient one to guarantee the executability of the path. Actions in the sequence may have, in its turn, additional dependencies that must be considered as well.

Roughly, the algorithm works by executing two loops[5]. The first loop uses the *updateDependencies* function to compute the dependencies for each action in the input *path*. This function updates the variable *depLists* as follows: (1) computes the dependencies for the action in *node.action* as stated in Table 5.1 (2) multiplies the *number* value in each dependency by the value of *node.number* and (3) adds the dependencies to the end of all lists in *depLists* (if all dependencies for *node* are AND-dependencies) or forks all lists and adds to the end of each cloned list a different dependency (in case of OR-dependencies) to represent the different alternatives we have to satisfy the dependencies.

The second loop tries to map each dependency *d* onto the actions in *path*. The *mapping(depLists[i],path)* function copies in *requiredActions[i]* the actions of *depLists[i]* that either do not map in the path or that map with an insufficient *number* value. In this latter case, the dependency is added indicating the additional number of actions that are needed. In the former, *number* is directly extracted from *d.number*.

If at least one of the lists in *depLists* is fully satisfied the *path* is determined as weakly executable. Otherwise, the algorithm returns in *requiredActions* a list of repair actions for each possible way of satisfying the dependencies.

The execution of the *executability* function for the *submitPaper* and *dismiss* operations (Fig. 2.2) is shown in Tables 5.3.1 and 5.3.2. *EndOfReview* is detailed in Appendix C. $v_1 \ldots v_n$ represent free variables introduced by the rules.

Table 5.3.1. Weak Executability for the *submitPaper* operation.

submitPaper	*Input path*	p = [<1,p:=CreateObject(UnderReview)>, <1,AddStructuralFeature(p,title,tit)>, <N,CreateLink(IsAuthorOf,author,authors[i],paper,p)>]
	Dependencies	depLists[0] = [<1,AddStructuralFeature(p,title,v_1)>, <1,CreateLink(IsAuthorOf,person,v_2,paper,p)>, <N,DestroyLink(IsAuthorOf,person,authors[i],paper,v_3)> depLists[1] = [<1,AddStructuralFeature(p,title,v_1)>, <1,CreateLink(IsAuthorOf,person,v_2,paper,p)>, <N,p:=CreateObject(UnderReview)>]
	Output	requiredActions = ø (depLists[1] maps correctly with input path *p*) executability = TRUE

The only path of *submitPaper* operation is executable since all dependencies in *depLists[1]* are satisfied by the path (when *N* takes the value 1, the last dependency can be mapped to the first node in the path). Thus, the operation is weakly executable.

Table 5.3.2. Weak Executability for the *dismiss* operation.

dismiss	*Input path*	p = [<1,DestroyLink(WorksIn,person,self,department,self.department)>]
	Dependencies	depLists[0] = [<1,CreateLink(WorksIn,person,self,department,v_1)>] depLists[1] = [<1,DestroyObject(self)>]
	Output	requiredActions[0] = [<1,CreateLink(WorksIn,person,self,department,v_1)>] requiredActions [1] = [<1,DestroyObject(self)>] executability = FALSE

This path is not executable (and thus, neither the *dismiss* operation, since this is its only path), because removing the link violates the multiplicity '1' of *WorksIn*. Adding a new link to the dangling object (with *CreateLink(WorksIn,...)*) or destroying it (with *DestroyObject(self)*) would make the path executable, as reported by our method.

[5] We could also mix both loops by checking partial satisfiability of *depLists* after each node (more efficient but with a poorer feedback since only part of the required actions would be returned).

6 Completeness

Users evolve the system state by executing the set of write actions defined in the behavior elements of the UML model (the operations in the class diagram in our case). Intuitively, we say that the set of actions in an UML model is *complete* when, all possible changes (inserts/updates/deletes/…) on all parts of the system state can be performed through the execution of those actions. Otherwise, there will be parts of the system that users will not be able to modify since no behavioral elements address their modification. For instance, the set of actions in the operations defined in Fig. 2.2 is incomplete since actions to remove a person or to create and remove departments are not specified, forbidding users to perform such kind of changes on the data.

We feel this property is important to guarantee that no behavioral elements are missing in the model. Clearly, it may happen that a class diagram contains some elements that designers do not want the users to modify but then those elements should be defined, for instance, as derived information or read-only elements.

More formally, an operation set $set_{op} = \{op_1,...,op_n\}$ is complete when, for each modifiable element *e* in the class diagram and each possible action *ac* modifying the population of *e*, there is at least a weak executable path in some op_i that includes *ac*.

A simple function for checking the completeness of set_{op} is the following:

```
function completeness (in: cd: <Set(Class), Set(StructuralFeature),
Set(Association), Set(GeneralizationSet)>, in: op: Set(Operation), out:
feedback: Set(Action)): Boolean
{  requiredActionsSet, existingActionsSet: Set(Action):=∅;
   action: Action; feedback:=∅;
   existingActionsSet:=getExistingActions(op);
   requiredActionsSet:=getRequiredActions(cd);
   for each action ∈ requiredActionsSet do
    if action∉existingActionsSet then feedback:=feedback U {action};
    endif
   endfor
   return (feedback = ∅);}
```

The parameters of the *completeness* function are the model elements of the class diagram. The result indicates whether the set of operations is complete. For incomplete operations sets, the output parameter *feedback* contains the set of actions that should be included in some operation to satisfy the completeness property. *GetExistingActions* simply retrieves all different actions of weak executable paths of the operations set (*op* parameter). *GetRequiredActions* computes the set of actions that the software system should provide to its users in order to be able to modify all parts of the system state, depending on the structure and properties of the class diagram.

The set of actions returned by *getRequiredActions* is computed by first determining the modifiable model elements in the class diagram (i.e. the elements whose value or population can be changed by the user at run-time) and then deciding, for each modifiable element, the possible types of actions that can be applied on it.

A class is modifiable as long as it is not an abstract class and it is not the supertype of a complete generalization set (instances of such supertypes must be created/deleted

through their subclasses). An attribute is modifiable when it is not derived[6]. An association is modifiable if none of its member ends are derived.

For each modifiable class *c*, users must be provided with the actions *CreateObject(c)* and *DestroyObject(o:c)*[7] to create and remove objects from *c*. For each modifiable attribute *at* the action *AddStructuralFeature(o,at,v)* is necessary. For each modifiable association *as,* we need the actions *CreateLink(as,p_1,o_1,p_2,o_2)* and *DestroyLink(as,p_1,o_1,p_2,o_2)*. For generalizations, we need a set of actions *ReclassifyObject(o,nc,oc)* among the classes involved in them to specialize (generalize) the object *o* to (from) each subclass of the generalization. As an example, the result of *getRequiredActions* for our running example is provided in Appendix C.

7 Redundancy

An operation specification may be redundant at three different levels. We may have that some actions in an execution path are redundant, that the complete execution path is redundant or that the operation as a whole is itself redundant.

An action in an execution path is redundant if its effect on the system state is subsumed by the effect of later actions in the same path (e.g. two updates on the same attribute of the same object, the second overwrites the first one). We have identified several patterns that detect such redundant actions (see Appendix B for details).

An execution path e_1 is redundant with respect to an execution path e_2 (of the same or a different operation) when p_1 is subsumed by p_2, i.e. when all actions in p_1 appear in p_2 with the same or lower *number*. This may be perfectly correct (e.g. p_1 may appear in a basic operation whose behavior is also included in a more complex one) but it should be highlighted as suspicious, specially when it happens also that p_2 is redundant respect to p_1, meaning that both paths have exactly the same actions.

Finally, we say that an operation op_1 may be redundant when all its execution paths are redundant, especially when all its paths are redundant respect to the paths of the same operation op_2. Even if both operations make sense, designer could probably merge them to favor the simplicity of the schema.

8 Related work

There is a broad set of research proposals devoted to the verification of behavior specifications in UML, focusing on state machines [15], [14], [18], interaction diagrams [1], sequence diagrams [11], activity diagrams [8] or on the consistent interrelationship between them and/or the class diagram [13], [5], [10], [25], [24], [22], [6], among others. Nevertheless, many of these methods target very basic correctness properties (basically some kind of well-formedness rules between the

[6] Read-only attributes are considered modifiable because users must be able to initialize their value (and similar for read-only associations).

[7] Or a generic operation *DestroyObject(o:OclAny)* to remove objects of any class.

different diagrams) and/or restrict the expressivity of the supported UML models. Most of the methods above do not accept the specification of actions in the behavior specifications (a relevant exception is [23]), which is exactly the focus of our method.

Another major difference is the formalism used to perform the verification. To check the executability of a behavior specification (or, in general, any property that can be expressed as a Linear Temporal Logic formula - LTL [7]) previous approaches rely on the use of model-checking techniques [12]. Roughly, model checkers work by generating and analyzing all the potential executions at run-time and evaluating if for each (or some) execution the given property is satisfied.

When compared with model-checking methods, our approach presents several advantages. First of all, our analysis is static (no animation/simulation of the model is required) and, thus, our method is more efficient. Model-checking methods suffer from the state-explosion problem (i.e. the number of potential executions to analyze grows exponentially in terms of the size of the model, the domains of the parameters,…) even though a number of optimizations are available (as partial order reduction or state compression). Therefore, in general, it is not possible to explore all possible executions. This implies that results provided by these methods are not conclusive, i.e. absence of a solution cannot be used as a proof: an operation classified as not weakly executable may still have a correct execution outside the search space explored during the verification. Another advantage of our method is the kind of feedback provided to the designer when a property is not satisfied. Model-checking based proposals provide example execution traces that do (not) satisfy the property. In contrast, our method provides a more valuable feedback (for our correctness analysis) since it suggests how to change the operation specification in order to repair the detected inconsistency.

As a trade-off, our method is unable to verify arbitrary LTL properties. In this sense, we believe our method could be used to perform a first correctness analysis, basic to ensure a basic quality level in the actions specification. Then, designers could proceed with a more detailed analysis adapting current approaches presented above to the verification of behaviors specified with AS. For instance, example execution traces that lead to an error state would help designers to detect particular scenarios not yet appropriately considered.

Finally, we would like to remark that, to the best of our knowledge, our method is the first one considering the completeness, redundancy and syntactic analysis of action specifications.

9 Conclusions and Further Work

We have presented an efficient method for the verification of the correctness of AS specifications. In particular, we have focused on the verification of actions specified as part of the definition of the effect of imperative operation specifications, one of the key elements in all MDD methods. Our approach can be easily extended to cope with other kinds of behavioral specifications since all of them use AS for a fine-grained behavior specification.

Our method is based on a static analysis of the dependencies among the actions; an animation/simulation is not required. Thus, our method does not suffer from the state-explosion problem as current model-checking methods. As a trade-off, our method is not adequate for evaluating general LTL properties. Moreover, the feedback provided by our method helps designers to correct the detected errors since our method is able to suggest a possible repair procedure instead of just highlighting the problem.

Therefore, we believe that the characteristics of our method make it especially suitable for its integration in current CASE and code-generation tools, as part of the default consistency checks that those tools should continuously perform to assist designers in the definition of software models.

As a further work, we would like to complement our techniques by providing an automatic transformation between the UML AS specification and the input language of a model-checker tool (as the PROMELA language [12]) so that, after an initial verification with our techniques (simpler and which would efficiently provide a first correctness result), designers may get a more fine-grained (though partial) analysis by means of applying more complex model checking techniques. Also, we also plan to implement/integrate these techniques into a CASE tool and validate them with a more complex case study. In addition, we plan to empirically evaluate the computational cost of each technique and compare them.

Acknowledgements. Thanks to the anonymous referees and the people of the GMC group for their useful comments to previous drafts of this paper. This work has been partly supported by the Ministerio de Educación y Ciencia and FEDER under project TIN208-00444/TIN, Grupo Consolidado.

References

1. Baker, P., Bristow, P., Jervis, C., King, D., Thomson, R., Mitchell, B., Burton, S.: Detecting and Resolving Semantic Pathologies in UML Sequence Diagrams. ESEC/SIGSOFT FSE, 50-59 (2005)
2. Bollobas, B.: Modern graph theory. Springer (2002)
3. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. ASE, 547-548 (2007)
4. Cabot, J., Gómez, C.: Deriving Operation Contracts from UML Class Diagrams. MoDELS, LNCS, 4735, 196-210, (2007)
5. Gallardo, M.M., Merino, P., Pimentel, E.: Debugging UML Designs with Model Checking. Journal of Object Technology, 1(2), 101-117 (2002)
6. Egyed, A.: Instant Consistency Checking for the UML. ICSE, 381-390 (2006)
7. Emerson, E. A.: Temporal and Modal Logic. Handbook of Theoretical Computer Science, 8, 995-1072 (1990)
8. Eshuis, R.: Symbolic Model Checking of UML Activity Diagrams. ACM Transactions on Soft. Eng. and Methodology, 15(1), 1-38 (2006)
9. Garousi, V., Briand, L., Labiche, Y.: Control Flow Analysis of UML 2.0 Sequence Diagrams. ECMDA-FA, LNCS, 3748, 160-174 (2005)
10. Graw, G., Herrmann, P.: Transformation and Verification of Executable UML Models. Electronic Notes in Theoretical Computer Science, 101, 3-24 (2004)
11. Grosu, R., Smolka, S. A.: Safety-Liveness Semantics for UML 2.0 Sequence Diagrams. ACSD, 6-14 (2005)

12. Holzmann, G. J.: The spin model checker: Primer and reference manual. Addison-Wesley Professional (2004)
13. Knapp, A., Wuttke, J.: Model Checking of UML 2.0 Interactions. MoDELS Workshops, LNCS, 4364, 42-51 (2006)
14. Latella, D., Majzik, I., Massink, M.: Automatic Verification of a Behavioural Subset of UML Statechart Diagrams using the SPIN Model-Checker. Formal Aspects of Computing, 11(6), 637-664 (1999)
15. Lilius, J., Paltor, I. P.: Formalising UML State Machines for Model Checking.UML, LNCS, 1723, 430–445 (1999)
16. Marriott, K., Stuckey, P. J.: Programming with Constraints: An Introduction. MIT Press (1998)
17. Mellor Stephen J., Balcer Marc J.: Executable UML: A foundation for model-driven architecture. Addison-Wesley (2002)
18. Ober, I., Graf, S., Ober, I.: Validating Timed UML Models by Simulation and Verification. Int. Journal on Software Tools for Technology Transfer, 8(2), 128-145 (2006)
19. Object Management Group (OMG): UML 2.0 Superstructure Specification. OMG Adopted Specification (ptc/07-11-02) (2007)
20. Object Management Group (OMG): Semantics of a Foundational Subset for Executable UML Models RFP (ad/2005-04-02) (2005)
21. Olivé, A.: Conceptual Schema-Centric Development: A Grand Challenge for Information Systems Research.CAISE, LNCS, 3520 (2005)
22. Rasch, H., Wehrheim, H.: Checking Consistency in UML Diagrams: Classes and State Machines. FMOODS, LNCS, 2884, 229-243 (2003)
23. Turner E., Treharne H., Schneider S., Evans N.: Automatic Generation of CSP || B Skeletons from xUML Models. ICTAC, LNCS, 364-379 (2008)
24. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using Description Logic to Maintain Consistency between UML Models. UML, LNCS, 2863, 326–340 (2003)
25. Xie, F., Levin, V., Browne, J. C.: Model Checking for an Executable Subset of UML. ASE, 333-336 (2001)

Appendix A. Analyzing Syntactic Consistency

Our analsysis of the WFRs relevant to the Action Packages has detected several flaws that compromise the usefulness of these WFRs to ensure the syntactic correctness of action specifications. Some example errors are the following:

- Syntactic errors: References to "*forall*" (instead of "*forAll*") and "*oclisKindOf*" (for "*oclIsKindOf*") operations.
- UML 1.5 related errors: WFRs restricting the multiplicity of input and output pins refers to a *multiplicity* attribute that does not longer exist in the UML metamodel (in UML 2.0, pins are subtypes of *MultiplicityElement* and thus we should use the *upper* and *lower* attributes instead). There is also a reference to the now inexistent *NavigableEnd* metaclass.
- Semantic errors: The constraint *"context WriteStructuralFeatureAction inv: self.value.type = self.structuralFeature.featuringClassifier"* forces the type of the new value for the structural feature to be equal to the type of the Classifier owning the feature. Clearly, this is plain wrong. The type of the new value should be the same as the type of the structural feature, i.e. *"self.value.type = self.structuralFeature.type"*.
- It is not clear the relationship between the *InstanceSpecification*, *ValueSpecification* and *Pin* metaclasses. Since input and output pins must hold *InstanceSpecification* (e.g. in the *CreateObjectAction* action) and *ValueSpecification* (e.g. *WriteStructuralFeature* actions) values, both kind of values need to be converted to instances of the *Pin* metaclass which is not possible with the current metamodel structure.

Besides, several required WFRs are not predefined in the metamodel. For instance, in actions of type *WriteStructuralFeatureAction*, we should check that the type of the input object (i.e. the object whose feature will be modified) is compatible with the classifier owning the feature (OCL definition: *context WriteStructuralFeatureAction inv: self.value.type = self.structuralFeature.type*). Also, in *CreateObjectAction*, the input classifier cannot be the supertype of a covering generalization set (in a covering generalization, instances of the supertype cannot be directly created). Similar WFRs must be defined to restrict the possible *newClassifiers* in the *ReclassifyObjectAction*. For instance, we should check that the *newClassifiers* set and the *oldClassifiers* set are disjoint sets. Additional rules are needed to check that values of readOnly attributes are not updated after their initial value has been assigned, that the intersection between the old and new classifiers in an object reclassification is empty and so forth.

Once the previous ill-defined WFRs are fixed and the new ones are added to the metamodel specification, we can successfully analyze the well-formedness of action specifications as a first step of our verification process.

Appendix B. Redundancy Patterns

As we have seen in section 7, an action in an execution path is redundant if its effect on the system state is subsumed by the effect of later actions in the same path, that is, the final system state when executing the operation following that path would be exactly the same with or without the redundant action (e.g. two updates on the same attribute of the same object, the second overwrites the first one).

We have identified several patterns (see Table B.1) that detect redundant actions in execution paths (some of them are based on the concept of *net effect* for a database transaction first). For each pattern, we identify a redundant path (first column) and provide a possible non-redundant equivalent path (second column). The modification of the original paths cannot be fully automatic since, for instance, the redundant action may not be redundant in a different path also including that action or may affect the execution of other actions in the path. Nevertheless, we believe it is worth to at least point out these redundant actions to the designer.

Table B.1. Redundant patterns.

Redundant path	*Equivalent path*	*Description*
[...,AddStructuralFeature(o,at,v),..., AddStructuralFeature(o,at,v_2),...]	[..., AddStructuralFeature(o,at,v_2), ...]	The second update overwrites the first one
[...,o:=CreateObject(Cl),..., DeleteObject(o),...]	[...]	No need of creating an object that it is going to be removed within the same execution
[...,ReclassifyObject(o,[nc],[oc]),...,DeleteObject(o),...]	[...]	No need of reclassifying an object that it is going to be removed within the same execution
[...,CreateLink(as,p_1,o_1,p_2,o_2),...,DeleteLink(as,p_1,o_1,p_2,o_2),...]	[...]	No need of creating a link that it is going to be removed within the same execution
[...,ReclassifyObject(o,[Cl],[Cl_2]), ..., ReclassifyObject(o,[Cl_2],[Cl]), ...]	[...]	The last reclassification removes the effect of the first one
[...,ReclassifyObject(o,[Cl_2],[Cl]), ..., ReclassifyObject(o,[Cl_3],[Cl_2]),...]	[...,ReclassifyObject(o,[Cl_3],[Cl]), ...]	Transitive property

Appendix C. Running example

This appendix contains the complete application of our method on the running example shown in Fig. 2.1 and 2.2.

The input class diagram (Fig. C.1) represents part of a conference management system. This class diagram is complemented with the definition of three operations: *endOfReview*, *submitPaper* and *dismiss* (Fig. C.2). The first operation (*endOfReview*) reclassifies a paper as rejected or accepted depending on the *evaluation* parameter. The second one (*submitPaper*) creates a new "under review" paper and links the paper with its authors. The last one (*dismiss*) deletes the *WorksIn* link between a person and his/her department.

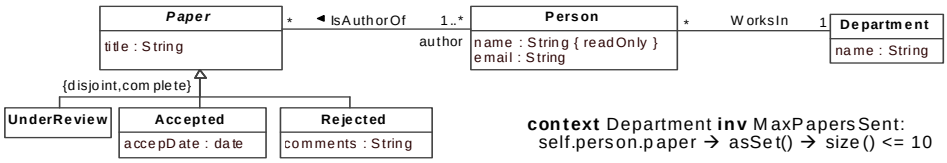

Fig. C.1. Excerpt of a conference management system class diagram.

```
context Paper::endOfReview(com:String,d:Date,
evaluation:String) {
 if self.oclIsTypeOf(UnderReview) then
  if evaluation = 'reject' then
    ReclassifyObject(self,[Rejected],[ ]);
    AddStructuralFeature(self,comments,com);
  else
    ReclassifyObject(self,[Accepted],[ ]);
    AddStructuralFeature(self,accepDate,d);
  endif
endif }
```

```
context Paper::submitPaper(tit:String, authors:Person[1..*]) {
 i: Integer := 1;
 p: Paper;
 p := CreateObject(UnderReview);
 AddStructuralFeature(p,title,tit);
 while i <= authors->size() do
  CreateLink(IsAuthorOf,author,authors[i],paper,p);
  i := i+1;
endwhile }
```

```
context Person::dismiss() {
 DestroyLink(WorksIn,person,self,department,self.department); }
```

Fig. C.2. Specification of *endOfReview*, *submitPaper* and *dismiss* operations.

As we have seen in the previous sections, our method proceeds by doing several steps: (1) analyzes the syntactic consistency of each action $ac \in op$, (2) computes all possible execution paths in *op*, (3) determines the executability of each path, (4) analyses the completeness of the whole operation set and (5) detects possible redundancies. For each detected error, possible corrective procedures are provided to the designer as a complementary feedback. In the next sub-sections, we show the execution of these steps.

Step 1: *Syntactic Correctness*

The first step of our method analyzes the syntactic consistency of each action $ac \in op$. In the running example, all actions included in operations satisfy the WFR. Thus, all operations of our example are syntactically correct.

Step 2: ***Computing the Execution Paths***

The second step of our method computes all possible execution paths in each operation of our example. Firstly, we represent each operation in a MBCFG:

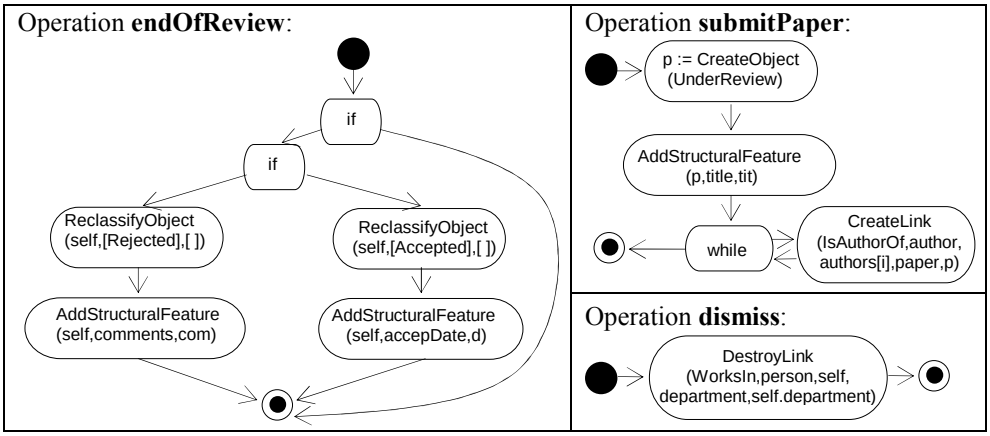

Fig. C.3. MBCFG of *endOfReview, submitPaper* and *dismiss* operations for the example.

Next, we compute all execution paths for the previous graphs:

endOfReview:
p1 = ø
p2 = [<1,ReclassifyObject(self,[Rejected],[ ])>, <1,AddStructuralFeature(self,comments,com)>]
p3 = [<1,ReclassifyObject(self,[Accepted],[ ])>, <1,AddStructuralFeature(self,accepDate,d)>]

submitPaper:
p = [<1,p:=CreateObject(UnderReview)>, <1,AddStructuralFeature(p,title,tit)>,
<N,CreateLink(IsAuthorOf,author,authors[i],paper,p)>]

dismiss:
p = [<1,DestroyLink(WorksIn,person,self,department,self.department)>]

Fig. C.4. Execution paths for *endOfReview, submitPaper* and *dismiss* operations.

Step 3: ***Weak Executability***

Applying the algorithm presented in section 5, we can determine the weak executability of each path of the operations specified in the example.

In the following, the execution of the *weakExecutability* function for the execution paths (Fig. C.4) is detailed step by step.

Table C.1. Weak Executability for the *endOfReview* operation.

endOfReview	*Input path*	$p_1 = ø$
	Dependencies	depLists = ø
	Output	requiredActions = ø executability = TRUE

endOfReview	*Input path*	p2 = [<1,ReclassifyObject(self,[Rejected],[ ])>, <1,AddStructuralFeature(self,comments,com)>]
	Dependencies	depLists[0] = [<1,AddStructuralFeature(self,comments,v_1)>, <1,ReclassifyObject(self,[ ],[UnderReview])>] depLists[1] = [<1,AddStructuralFeature(self,comments,v_1)>, <1,ReclassifyObject(self,[ ],[Accepted])>]
	Output	requiredActions[0] = [<1,ReclassifyObject(self,[ ],[UnderReview])>] requiredActions[1] = [<1,ReclassifyObject(self,[ ],[Accepted])>] executability = FALSE
endOfReview	*Input path*	p3 = [<1,ReclassifyObject(self,[Accepted],[ ])>, <1,AddStructuralFeature(self,accepDate,d)>]
	Dependencies	depLists[0] = [<1,AddStructuralFeature(self,accepDate,v_1)>, <1,ReclassifyObject(self,[ ],[UnderReview])>] depLists[1] = [<1,AddStructuralFeature(self,accepDate,v_1)>, <1,ReclassifyObject(self,[ ],[Rejected])>]
	Output	requiredActions[0] = [<1,ReclassifyObject(self,[ ],[UnderReview])>] requiredActions[1] = [<1,ReclassifyObject(self,[ ],[Rejected])>] executability = FALSE

The execution path p_1 is weak executable, since it does not contain any action. Otherwise, the execution paths p_2 and p_3 are not weak executable since they always violate the disjointness constraint of the generalization. The action required to become the paths executable is *ReclassifyObjectAction(self,Ø,UnderReview)* in both cases.

Table C.2. Weakly Executability for the *submitPaper* operation.

submitPaper	*Input path*	p = [<1,p:=CreateObject(UnderReview)>, <1,AddStructuralFeature(p,title,tit)>, <N,CreateLink(IsAuthorOf, author,authors[i],paper,p)>]
	Dependencies	depLists[0] = [<1,AddStructuralFeature(p,title,v_1)>, <1,CreateLink(IsAuthorOf,person,v_2,paper,p)>, <N,DestroyLink(IsAuthorOf,person,authors[i],paper,v_3)> depLists[1] = [<1,AddStructuralFeature(p,title,v_1)>, <1,CreateLink(IsAuthorOf,person,v_2,paper,p)>, <N,p:=CreateObject(UnderReview)>]
	Output	requiredActions = ø (depLists[1] maps correctly with input path *p*) executability = TRUE

The only path of *submitPaper* operation is executable since all dependencies in *depLists[1]* are satisfied by the path (if *N* takes the value 1, the last dependency can be mapped to the first node in the path), and thus, the operation is weakly executable.

Table C.3. Weakly Executability for the *dismiss* operation.

dismiss	*Input path*	p = [<1,DestroyLink(WorksIn,person,self,department,self.department)>]
	Dependencies	depLists[0] = [<1,CreateLink(WorksIn,person,self,department,v_1)>] depLists[1] = [<1,DestroyObject(self)>]
	Output	requiredActions[0] = [<1,CreateLink(WorksIn,person,self,department,v_1)>] requiredActions[1] = [<1,DestroyObject(self)>] executability = FALSE

This execution path is not executable (and thus, neither the *dismiss* operation since this is its only path), because removing the link violates the cardinality constraint '1' of *WorksIn*. Adding a new link to the dangling objects (with the *CreateLink(WorksIn,person,self,department,d')* action) or destroying it (with *DestroyObject(self)*) would make the path executable, as reported by our method.

Step 4: *Completeness*

Applying the algorithm presented in section 6, we can determine the completeness of the whole operation set.

In the following, the execution of the *completeness* function for our running example (Fig. C.1 and Fig. C.2) is detailed.

The operation *getRequiredActions* returns the following set of actions:

requiredActionsSet = [
//One *CreateObject(class)* action for each modifiable class of the diagram:
CreateObject(Accepted), CreateObject(Rejected), CreateObject(UnderReview), CreateObject(Person), CreateObject(Department),
//One *DestroyObject(class)* action for each modifiable class of the diagram:
DestroyObject(Accepted), DestroyObject(Rejected), DestroyObject(UnderReview), DestroyObject(Person), DestroyObject(Department),
//One *AddStructuralFeature(att)* action for each modifiable attribute *att*:
AddStructuralFeature(title), AddStructuralFeature(accepDate), AddStructuralFeature(comments), AddStructuralFeature(name), AddStructuralFeature(email),
//One *CreateLink(as)* action for each modifiable association *as*:
CreateLink(IsAuthorOf), CreateLink(WorksIn),
//One *DestroyLink(as)* action for each modifiable association *as*
DestroyLink(IsAuthorOf), DestroyLink(WorksIn),
//One *ReclassifyObject(o,[nc],[oc])* for each classes involved in the generalization to specialize (generalize) the object *o* to (from) each subclass of the generalization:
ReclassifyObject(o,[Accepted],[UnderReview]), ReclassifyObject(o,[Rejected],[UnderReview]), ReclassifyObject(o,[Rejected],[ ]), ReclassifyObject(o,[Accepted],[ ]), ReclassifyObject(o,[UnderReview],[ ])]

The operation *getExistingActions* retrieves all different actions of weak executable paths of the operations *endOfReview*, *submitPaper* and *dismiss*:

existingActionsSet = [
ur:=CreateObject(UnderReview), AddStructuralFeature(title), AddStructuralFeature(accepDate), AddStructuralFeature(comments), CreateLink(IsAuthorOf), DestroyLink(WorksIn), ReclassifyObject(o,[Rejected],[ ]), ReclassifyObject(o,[Accepted],[ ])]

Therefore, the output parameter *feedback* contains the set of actions that should be included in some operation to satisfy the completeness property.

feedback = [
CreateObject(Accepted), CreateObject(Rejected), CreateObject(Person), CreateObject(Department), DestroyObject(Accepted), DestroyObject(Rejected), DestroyObject(UnderReview), DestroyObject(Paper), DestroyObject(Department), AddStructuralFeature(name), AddStructuralFeature(email), CreateLink(WorksIn), DestroyLink(IsAuthorOf), ReclassifyObject(o,[Accepted],[UnderReview]), ReclassifyObject(o,[Rejected,[UnderReview]), ReclassifyObject(o,[UnderReview,[])]

Step 5: *Redundancy*

The last step of our method detects possible redundancies in the operations.

Following the redundant patterns described in Appendix B, we can determine that there is not any redundant path in the operations paths, and thus, we conclude that all operations are not redundant.