Validating UML and OCL Models in USE by Automatic Snapshot Generation

Martin Gogolla, Jörn Bohling, Mark Richters

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
`[gogolla|joebo|mr]@informatik.uni-bremen.de`

Abstract. We study the testing and certification of UML and OCL models as supported by the validation tool USE. We extend the available USE features by introducing a language for defining properties of desired snapshots and by showing how such snapshots are generated. Within the approach, it is possible to treat test cases and validation cases. Test cases show that snapshots having desired properties can be constructed. Validation cases show that given properties are consequences of the original UML and OCL model.
Key words: UML, OCL, Model Validation, Model Testing, Reasoning about Models, Class diagram, Invariant, Pre- and postcondition, Test case, Snapshot.

1 Introduction

The UML [OMG03] is accepted today as a de-facto standard for software development. The Object Constraint Language (OCL) [WK98] is an important part of the UML, and software support for OCL is growing (see the Dresden OCL compiler [HDF00] integrated within ArgoUML [Arg03], the Boldsoft tool [Bol02], work based on C++ in [Chi01], the KeY approach [ABB+00], or the NEPTUNE tool [CCJ+03]). One of the first systems supporting OCL was our UML Specification Environment (USE) [RG98,RG01]. Among other offered functionalities, USE allows to validate UML and OCL models by constructing snapshots representing system states at a particular point in time with objects, attribute values, and links. Such snapshots are represented as object diagrams [RG00].

However, up to now these snapshots had to be constructed by giving an explicit sequence of commands. The motivation for the current work is to construct snapshots in a more declarative way by specifying properties the desired snapshot has to fulfill. For this means, we have developed the language ASSL (A Snapshot Sequence Language) allowing the construction of snapshots apart from giving command sequences. Thus one can specify properties the resulting snapshots have to satisfy. Furthermore, the USE functionality has been extended in order to dynamically load invariants. USE and its ASSL extension afford the certification of properties about UML models, i.e., the developer can formally check properties against provided scenarios regarded as relevant. In terms of practical merits, our approach allows to construct large and non-trivial snapshots being consistent with a complex class diagram and illustrating complex scenarios.

The importance of validation and testing in object-oriented software development has been recognized for long, see for example [Amb97]. Thus, our work has connections to other related approaches (apart from the ones we have mentioned above in connection directly with OCL). For example, the approach in [OK99] focuses on animation. Although the starting point of [JGP99] is distributed software, the paper discusses a set of general techniques like verification, simulation or testing for validation purposes. The paper also introduces the UMLAUT tool which can manipulate the UML representation of a system and which integrates different validation techniques. The approach in [DdB00] emphasizes the importance of having multi-formalisms by translating UML models to Z and Lustre specifications which then are validated by means of a prover and a testing environment. [Mut00] studies validation of UML models by means of theorem proving, in particular in connection with PVS. Snapshots were also generated from dynamic models such as Use Cases or State Charts. For instance, [FL00] derive a sequence of messages from Use Cases by transforming them into planning problems. The Alloy constraint analyzer [JSS00] is a tool that allows the generation of snapshots for a model based on relational logic. However, an Alloy model does not support significant UML and OCL features like attributes and its data types. Approaches concentrating on testing of UML models have been considered for special applications like smart cards [Mar99], and with emphasis on special UML language features like sequence diagrams [GR00] or collaboration [AO00] diagrams. The approach in [OA99] adapts state-based specification test data generation criteria to generate test cases from UML statecharts obeying classical test coverage criteria. In contrast to these works, our approach in USE is the only one which focusses on OCL and which allows to describe test cases, i.e., snapshots, in a descriptive way using the same language (OCL) which is used for describing the models itself.

The structure of the rest of this paper is as follows. Section 2 gives an overview of an example scenario which is used in this paper to point out the new ideas of our approach. Section 3 introduces in detail the class diagram including operations and invariants on which we rely and the ASSL constructs needed for automatic snapshot generation. Section 4 shows how to apply ASSL elements to test cases and something what we call validation cases. Section 5 explains a denotational semantics for our language ASSL. The paper ends with concluding remarks in Sect. 6. Appendix A shows one additional example for an ASSL procedure, and Appendix B offers more details of the semantics of single ASSL commands.

2 Overview by a Deployment Diagram

Fig. 1 shows a UML deployment diagram picturing the files and their dependencies for the example scenario used in this paper. We try to strictly adhere to the

UML notation in this diagram. Files are regarded as components being displayed as rectangles with two smaller rectangles protuding from its side, and comments are shown as rectangles with bent upper-right corner (dog-ear). Comments are attached by undirected lines to components, and dependencies between components are shown as directed dashed lines. These directed dashed lines express that, when the component where the dashed arrow ends is modified, this may affect the component where the dashed arrow starts. In concrete terms, when, e.g., the file `percom.use` changes this may effect the file `twoRoleTC.cmd`.

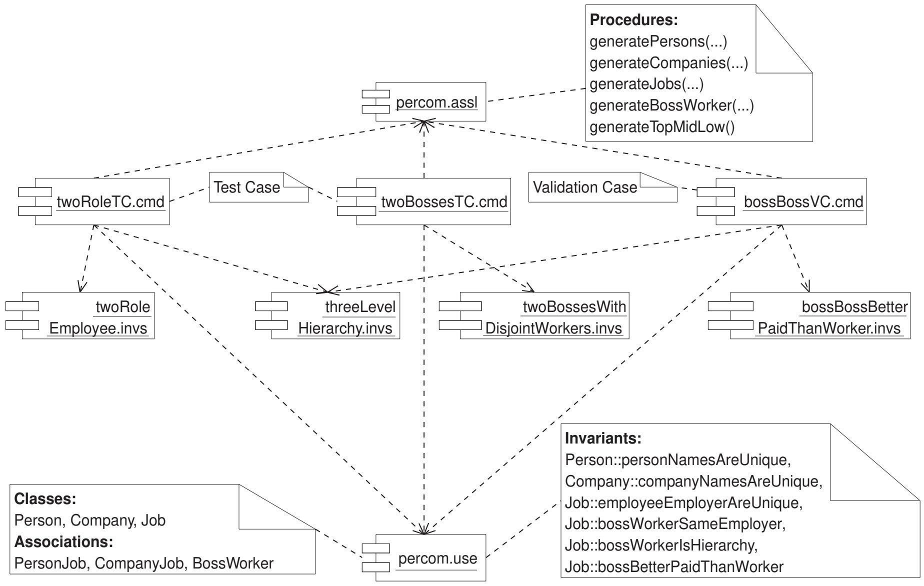

Fig. 1. Deployment Diagram with `use`,`invs`, `cmd`, and `assl` Files

At the bottom, one recognizes the file `percom.use` (PERson and COMpany) containing the USE model of the classes, associations, and invariants as indicated by the comments on both sides. On the next higher level, the names of four `invs` files are shown, each one containing one invariant. These invariants are not contained in the `use` file. On the top, the file `percom.assl` can be seen which contains several procedures used for constructing snapshots consisting of objects and links. Below this file, three command files with ending `cmd` are displayed. Each command file is responsible for performing either a test case or something what we call validation case which is explained below. A command file first loads the `percom.use` file into the USE system in order to make the classes, associations, and invariants known. A command file afterwards calls procedures from the `percom.assl` file and thereby creates objects, attribute values, and links. In addition to using the invariants from the `use` file, a command file may dynamically load further invariants from `invs` files. These invariants determine the properties of the created objects and links in the snapshot. The particular procedure calls construct the desired snapshots. But it is not required that each separate snapshot is created by a separate procedure. Skilful parametrization of procedures can provide general building blocks that can be re-used for the construction of several snapshots.

But let us come back to the test and validation cases. On the one hand, a test case has the task to certify that it is allowed to construct a certain snapshot on the basis of the already present invariants and the dynamically loaded invariants. On the other hand, a validation case certifies that a dynamically loaded invariant is a semantic consequence of the already present invariants by showing that it is not possible to construct a snapshot satisfying the already present invariants and the negation of the dynamically loaded invariant. These ideas are detailed below.

3 USE Definitions, ASSL Procedures, and Dynamic Invariants

The USE tool and its ASSL extension are a lightweight formal method following the classification in [AL99]. That means, USE allows analysis of specifications rather than focusing entirely on correctness proofs. The purpose of the USE tool is to animate, test, and validate a UML class diagram and its OCL constraints and to focus on the early stages of the development process. Thus defects in a UML class diagram, e.g., whether a class cannot have instances or an association cannot have links, can be detected early before the implementation starts. Extending the USE tool with automated system state generation allows the generation of snapshots and, most important, testing a large number of system states against additional OCL constraints. The purpose is still to find defects, thus USE and ASSL are still a lightweight formal method.

3.1 Class Diagram and Test and Validation Cases

The UML class diagram in Fig. 2 shows an example from the UML Notation Guide [OMG03] but specifies some additional details. This example is contained in the file `percom.use` of Fig. 1. Persons may have jobs at several companies receiving from each company a salary. In each company, a set of workers may be managed by a single boss. We expect the following invariants to hold. Persons and companies are identified by their names. A Job object is determined by its employee and employer links. A boss and a worker belong to the same company, the links in the association BossWorker constitute a hierarchy, and a boss is better paid than a worker. In addition to the role names boss and worker, the operations bossPlus() and workerPlus() make the transitive closure of the respective role names available. Note that the BossWorker hierarchy is built on jobs, not on persons.

As an example, let us consider the snapshot in Fig. 3. Links in the BossWorker association are always displayed by placing the boss job above its worker jobs.

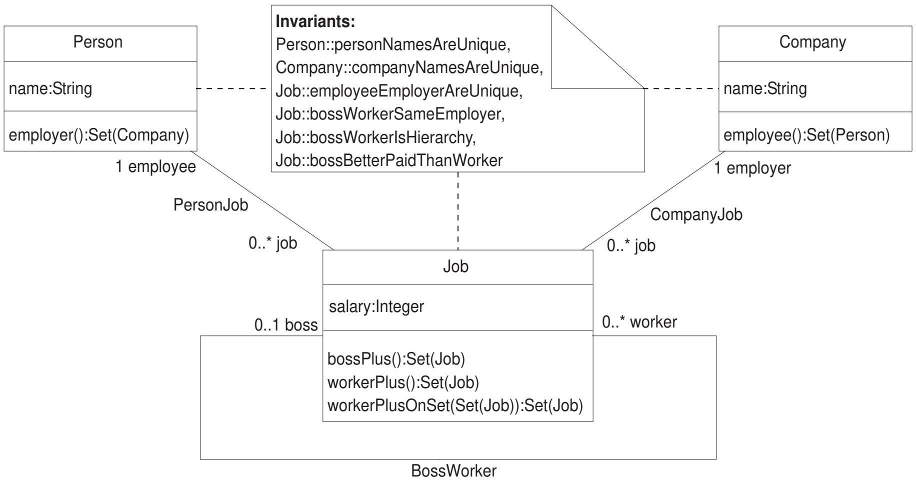

Fig. 2. Class Diagram

One can also identify the boss and the worker by considering the salary which is always higher for the boss. In Fig. 3, Cher works for Sony and Lex. Within Sony, Cher has Tom and Nick as workers. Within Lex, Cher's boss is Tom and Cher's worker is Nick. Coming back to the deployment diagram in Fig. 1, the goal of the test case `twoRoleTC` is to construct a snapshot with a three level job hierarchy where a single person plays two roles, i.e., the person has a boss job in one company and a worker job in the other company. The goal of the test case `twoBossesTC` is to show a snapshot where two bosses with disjoint worker sets exist. The validation case `bossBossVC` shows that, as a consequence from the specified invariants, not only the boss of a worker is better paid than the worker, but also the boss of the boss (the bossBoss) of the worker is better paid.

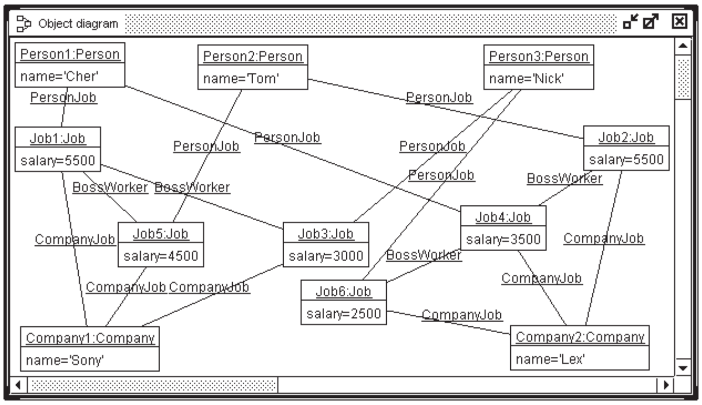

Fig. 3. Snapshot generated by `twoRoleTC.cmd`

3.2 USE Definitions

The following USE definitions show details not covered in Fig. 2. First, the five operations from the class diagram are defined by OCL expressions.

```
Person::employer():Set(Company)=self.job.employer->asSet()
Company::employee():Set(Person)=self.job.employee->asSet()
Job::bossPlus():Set(Job)=if boss.isDefined
  then boss.bossPlus()->including(boss)
  else oclEmpty(Set(Job)) endif
Job::workerPlus():Set(Job)=workerPlusOnSet(worker)
Job::workerPlusOnSet(s:Set(Job)):Set(Job)=
  let oneStep:Set(Job)=s.worker->asSet in
  if oneStep->exists(j|s->excludes(j))
    then workerPlusOnSet(s->union(oneStep)) else s endif
```

Note that we have defined the transitive closure here by employing recursively defined operations. But OCL allows a quite general `iterate` construct which makes it possible to define the transitive closure without recursion.

Second, the invariants informally mentioned above are formally characterized by OCL formulae.

```
context p1:Person inv personNamesAreUnique:
  Person.allInstances->forAll(p2|p1.name=p2.name implies p1=p2)
context c1:Company inv companyNamesAreUnique:
  Company.allInstances->forAll(c2|c1.name=c2.name implies c1=c2)
context j1:Job inv employeeEmployerAreUnique:
  Job.allInstances->forAll(j2|
    j1.employee=j2.employee and j1.employer=j2.employer
      implies j1=j2)
context top:Job inv bossWorkerSameEmployer:
  top.worker->forAll(low|low.employer=top.employer)
context j:Job inv bossWorkerIsHierarchy:
  j.workerPlus()->excludes(j)
context top:Job inv bossBetterPaidThanWorker:
  top.worker->forAll(low|low.salary<top.salary)
```

We assume the operation bossPlus() is only called in valid states where the invariants hold, in particular there should be no cycles in the association BossWorker. We do not assume this for the operation workerPlus() since this operation is used in the invariant bossWorkerIsHierarchy to ensure that no cycles exist. We could have defined bossPlus() analogously to workerPlus(), but in order to keep the definitions short we have preferred the current solution.

We are aware of the fact that our example is rather complex: first, there is a reflexive association (i.e., an association where one class participates twice);

second, the example involves the transitive closure of the two roles for that reflexive association; third, the BossWorker hierarchy is formally defined on the class Job, but one tends to think of it in terms of Person objects. However, this complexity allows us to study some rather complicated invariants and snapshots.

3.3 ASSL Procedures

The task of an ASSL procedure is to construct a snapshot or part of a snapshot. A call to the following ASSL procedure `generateCompanies` generates as many companies as indicated by its parameter `count`. The `CreateN` expression returns a sequence of length `count` of newly generated Company objects. Afterwards, within a loop ranging over all these new objects, the procedure tries to assign the attribute name for each new Company object. A choice of ten names is offered, but this choice is restricted with `reject` to the names not already present in the current set of Company objects. Note that the OCL expression `Sequence{...}->reject(...)` in the loop is evaluated newly every time it is reached within the loop. Assuming no Company object is present, calling, for example, `generateCompanies(3)` will construct a snapshot with three Company objects with distinct names from the offered choice. If one alternatively calls `generateCompanies(11)`, the ASSL generator will not construct a valid snapshot because the eleventh company cannot obtain a name distinct from the existing company names because only ten choices are offered.

```
procedure generateCompanies(count:Integer)
var theCompanies:Sequence(Company);
begin
theCompanies:=CreateN(Company,[count]);
for p:Company in [theCompanies]
  begin
  [p].name:=Any([Sequence{'AMD','DEC','HP','IBM','Lex','Miro',
    'NEC','SAP','Sony','Sun'}
    ->reject(n1|Company.allInstances.name->exists(n2|n1=n2))]);
  end;
end;
```

It is essential in procedure `generateCompanies` to reduce the search space by the `reject` expression in order to achieve the result efficiently. If we drop this `reject` expression, the desired snapshots are still within the described search space, because the invariant `companyNamesAreUnique` rejects duplicate names, but it will take much longer to receive a desired snapshot. Completely analogously to `generateCompanies`, the file `percom.assl` contains a procedure `generatePersons` offering a choice of 26 person names (one name for each letter in the alphabet).

Let us now discuss the generation of links in the procedure `generateJobs` given below. First, a fixed number of new Job objects is created. As the multiplicity in the class diagram requires, each of these Job objects must be linked to one employee aPerson and one employer aCompany. As possible employee candidates, first the set of all Person objects is considered, but then this is directly restricted by the following select to those Person objects linked currently to the minimal number of Job objects. As employer candidates, the set of all Company objects is considered, but this is restricted by the reject to those companies not already linked to the chosen employee candidate. As in the case of employee candidates, this choice is further restricted to those companies linked currently to the minimal number of Job objects. Both restricting select expressions contribute to fairness of Job distribution (one tries to give the same number of jobs to all companies) and make the resulting snapshot look more balanced. In contrast to this, the reject expression contributes to invariant satisfaction by reducing the search space. This reject expression guarantees that the invariant employeeEmployerAreUnique will never be violated by a constructed snapshot (which is still to be checked against the invariants). The calculated employee aPerson and employer aCompany are then inserted into the respective associations.

```
procedure generateJobs(count:Integer)
var theJobs:Sequence(Job), aPerson:Person, aCompany:Company;
begin
theJobs:=CreateN(Job,[count]);
for j:Job in [theJobs]
  begin
  aPerson:=Try([Person.allInstances->asSequence
               ->select(p|Person.allInstances->forAll(p2|
                           p.job->size<=p2.job->size))]);
  aCompany:=Try([Company.allInstances->asSequence
                ->reject(c|aPerson.employer()->includes(c))
                ->select(c|Company.allInstances->forAll(c2|
                            c.job->size<=c2.job->size))]);
  Insert(PersonJob,[aPerson],[j]);
  Insert(CompanyJob,[aCompany],[j]);
  end;
end;
```

After a complete snapshot has been constructed, the invariants are checked. If one invariant fails, backtracking occurs and an alternative choice is taken considering the last Try expression offering still alternatives. The Try and Any expressions syntactically look identical. However, the Any expressions do not allow backtracking, whereas the Try expressions do so.

With the same ASSL language elements as described above one can define procedures to generate links in the BossWorker association. The procedure generateBossWorker mentioned in Fig. 1 is given in Appendix A and will be used in the command files. The procedure generateTopMidLow also mentioned in Fig. 1 will be explained below.

Before we discuss the invariants to be loaded dynamically, let us make some more technical remarks on ASSL. An ASSL procedure is a language construct that defines, in the context of a given initial snapshot and given actual parameter values, a sequence of snapshots. Upon starting a procedure, the ASSL interpreter iterates through that sequence of snapshots. If a snapshot is found that fulfills the model-inherent constraints and the OCL invariants, then the search stops. The output is a sequence of USE commands that transforms the initial snapshot into the found snapshot. If a snapshot that fulfills the constraints and invariants is not found, then the ASSL interpreter reports `No valid state found`. An ASSL procedure has also parameter variables and local variables. Thus an ASSL command is evaluated in the context of the current snapshot and the variables. The tuple consisting of a snapshot and variable values is called a configuration. For example, the first ASSL command of a procedure is evaluated in the configuration consisting of the initial snapshot and the actual values of the parameters.

Some ASSL commands such as `Try` are producing a sequence of configurations. For instance, after evaluating `i:=Try([Sequence{17,15,13}])` the variable `i` has the value `17` in the first configuration, the value of `i` in the second configuration is `15`, and finally, in the third configuration the value of `i` is `13`. Because an ASSL procedure is a sequence of ASSL commands, executing an ASSL procedure is done by a depth-first strategy through a tree consisting of configurations. Only the snapshots of the configuration tree leafs are checked against the OCL constraints. That means, an ASSL procedure does not stop in the middle of its execution with a `Valid state found` result. OCL expressions may be part of ASSL procedures. There, OCL expressions are always contained in square brackets. An OCL expression is evaluated in the context of the current configuration. The language ASSL was first introduced in [Boh01].

3.4 Dynamically Loaded Invariants

As already mentioned above, command files can dynamically load invariants in order to generate snapshots with certain properties or to certify given properties. All these invariants will be referenced in the following by their names. The command files will employ the invariant twoRoleEmployee demanding that there exists a Person object with a job playing the boss role in one company and with a second job playing the worker role in a second company.

```
context Person inv twoRoleEmployee:
  Person.allInstances->exists(p |
    p.job->exists(low, top |
      low.boss.isDefined and top.worker->notEmpty and
      low.employer<>top.employer))
```

The invariant threeLevelHierarchy demands that there are three jobs arranged in a hierarchical fashion in the BossWorker association.

```
context Job inv threeLevelHierarchy:
  Job.allInstances->exists(top, mid, low |
    top.worker->includes(mid) and mid.worker->includes(low))
```

The invariant twoBossesWithDisjointWorkers requires that there are two bosses with disjoint worker sets (direct or indirect workers).

```
context Person inv twoBossesWithDisjointWorkers:
  Person.allInstances->exists(top1, top2 |
    let top1Worker:Set(Job)=top1.job.workerPlus()->asSet in
    let top2Worker:Set(Job)=top2.job.workerPlus()->asSet in
    top1Worker->notEmpty and top2Worker->notEmpty and
    top1Worker->intersection(top2Worker)->isEmpty)
```

Last, the invariant bossBossBetterPaidThanWorker assures that the boss of the boss of a worker is better paid than the worker.

```
context Job inv bossBossBetterPaidThanWorker:
  Job.allInstances->forAll(low, mid, top |
    low.boss=mid and mid.boss=top implies low.salary<top.salary)
```

It is worth to mention that the negated version of the previous invariant is equivalent to the invariant NEGATEDbossBossBetterPaidThanWorker.

```
context Job inv NEGATEDbossBossBetterPaidThanWorker:
  Job.allInstances->exists(low, mid, top |
    low.boss=mid and mid.boss=top and low.salary>=top.salary)
```

3.5 ASSL Commands

In this subsection we summarize the available ASSL commands.

- `Create(cls)` returns one new object in class `cls`.
- `CreateN(cls,expr)` returns a sequence of length `expr` with new objects in class `cls`.
- `Insert(assoc,object-1,...,object-n)` inserts the tuple `(object-1, ..., object-n)` into the association `assoc`.
- `Any(seq-expr)` returns an arbitrary value from the sequence `seq-expr` without allowing backtracking.
- `Sub(seq-expr)` returns an arbitrary subsequence from the sequence `seq-expr` without allowing backtracking.
- `Sub(seq-expr,size-expr)` returns an arbitrary subsequence of length `size-expr` from the sequence `seq-expr` without allowing backtracking.

– `Try(seq-expr)` returns a single value from the sequence `seq-expr` with the possibility of backtracking where the choices in the backtracking process are determined by the order in the sequence `seq-expr`.
– `Try(assoc,seq-expr-1,...,seq-expr-n)` empties the association `assoc` and successively assigns (with allowing backtracking) all possible link sets to the association `assoc` which can be build from the sequence expressions `seq-expr`: first the empty link set, then the link sets having one link, then the link sets with two links and so on.

4 Test and Validation Cases

This section applies ASSL elements to test and validation cases. A test case certifies that it is allowed to construct a snapshot fulfilling the invariants. A validation case certifies that a dynamically loaded invariant is a consequence of given invariants.

4.1 Test Case `twoRoleTC`

The task of the test case `twoRoleTC` is to generate some companies, persons, and jobs. Furthermore the association links have to be established in such a way that a three level BossWorker hierarchy in at least one company and a two role employee, i.e., a person possessing a boss job and a worker job in different companies, become part of the snapshot. The central steps of the command file `twoRoleTC.cmd` are shown below (we here omit some details of the command file which show the result of the last generation process and which actually generate the new snapshot).

```
open percom.use
gen start percom.assl generateCompanies(2)
gen start percom.assl generatePersons(3)
gen start percom.assl generateJobs(6)
gen load threeLevelHierarchy.invs
gen load twoRoleEmployee.invs
gen start percom.assl generateBossWorker(4)
```

The command file demands to read the respective USE file and to start the generation process of 2 companies, 3 persons, and 6 jobs which means that every person will work in every company. Up to now we have generated Company, Person, and Job objects and CompanyJob and PersonJob links. Now we load the two invariants which guide the generation of BossWorker links into the desired direction. We generate 4 BossWorker links meaning that every employee in every company will participate in at least one such link. The result of the automatic generation process is pictured in Fig. 3. The company with the three level BossWorker hierarchy is Lex, and Cher and Tom are two role employees.

4.2 Test Case `twoBossesTC`

The task of the test case `twoBossesTC` is to generate a BossWorker hierarchy with two bosses having disjoint, non-empty sets of direct or indirect workers (workerPlus) but this time within a single company. Again, we only show the central steps of the command file `twoBossesTC.cmd`.

```
open percom.use
gen start percom.assl generateCompanies(1)
gen start percom.assl generatePersons(7)
gen start percom.assl generateJobs(7)
gen load twoBossesWithDisjointWorkers.invs
gen start percom.assl generateBossWorker(6)
```

We generate 1 company, 7 persons, and a job for each person. We load the controlling invariant and request the generation of 6 BossWorker links. The result generated automatically is displayed in Fig. 4. Max and Ida are the bosses with disjoint, non-empty worker sets. Without loading the controlling invariant twoBossesWithDisjointWorkers a flat hierarchy would be generated. For the example in Fig. 4 this would mean that Dan would be the only boss and all other persons would be direct workers for him.

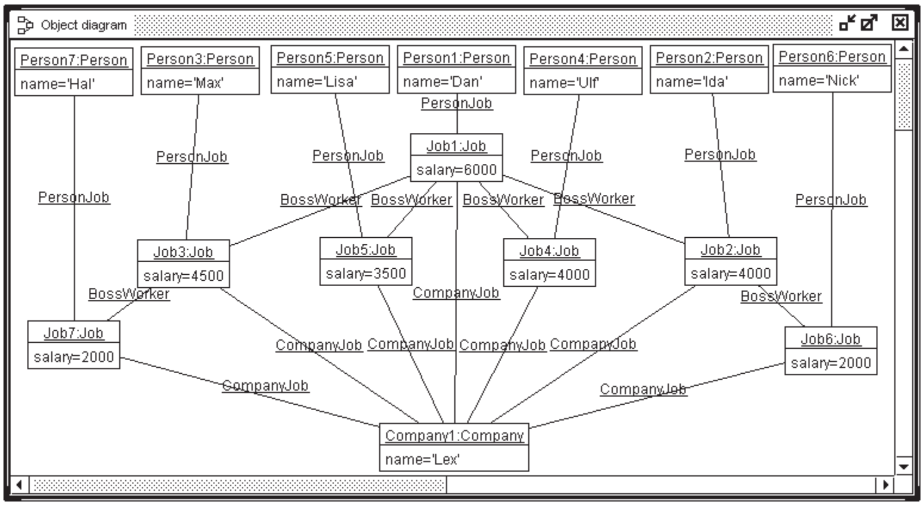

Fig. 4. Snapshot generated by `twoBossesTC.cmd`

4.3 Validation Case `bossBossVC`

As said above, a validation case certifies that a dynamically loaded invariant is a consequence of already present invariants by showing that it is not possible to construct a snapshot satisfying the already present invariants P-INVS and the negation of the dynamically loaded invariant L-INV:

$$(\text{P-INVS} \Rightarrow \text{L-INV}) \Leftrightarrow \neg (\text{P-INVS} \wedge \neg \text{L-INV}).$$

The validation case `bossBossVC` shows that, as a consequence from the specified invariants, not only the boss of a worker is better paid than the worker, but also the boss of the boss (the bossBoss) of the worker is better paid. The following protocol file shows the central steps in command file `bossBossVC.cmd` and indicates responses from USE.

```
use> open percom.use
use> gen start percom.assl generateCompanies(1)
use> gen start percom.assl generatePersons(3)
use> gen start percom.assl generateJobs(3)                (A)
use> gen load threeLevelHierarchy.invs
     Added invariants: Job::threeLevelHierarchy
use> gen load bossBossBetterPaidThanWorker.invs
     Added invariants: Job::bossBossBetterPaidThanWorker
use> gen flags Job::bossBossBetterPaidThanWorker +n       (B)
use> gen start percom.assl generateTopMidLow()
use> gen result
     Checked 162 snapshots.
     Result: No valid state found.                        (C)
```

We generate 1 company, 3 persons, and 3 jobs. Fig. 5 shows the result at point `(A)` and at point `(C)`. Note that no BossWorker links have been generated at point `(C)`. After point `(A)` we load the two desired controlling invariants. Afterwards we set the negate flag of invariant `bossBossBetterPaidThanWorker` directly before point `(B)` meaning that not the invariant itself is currently valid but its logical negation. This is equivalent to explicitly loading the invariant `NEGATEDbossBossBetterPaidThanWorker`. Then the ASSL procedure `generateTopMidLow` given below is started (`gen start`) and we ask to present the result (`gen result`).

The USE system responds that it has checked 162 snapshots but no valid state was found. Why do we end up with 162 snapshots? Let us have a more detailed look at procedure `generateTopMidLow`.

```
procedure generateTopMidLow()
var top:Job, mid:Job, low:Job, jobs:Sequence(Job);
begin
jobs:=[Job.allInstances->asSequence];
top:=Try([jobs]);
mid:=Try([jobs->excluding(top)]);
low:=Try([jobs->excluding(top)->excluding(mid)]);
```

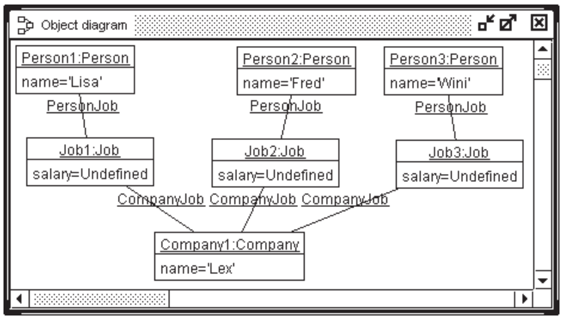

Fig. 5. Snapshot generated by `bossBossVC.cmd`

```
Insert(BossWorker,[top],[mid]);
Insert(BossWorker,[mid],[low]);
[top].salary:=Try([Sequence{2000,3000,4000}]);
[mid].salary:=Try([Sequence{2000,3000,4000}]);
[low].salary:=Try([Sequence{2000,3000,4000}]);
end;
```

The procedure `generateTopMidLow` assumes that 3 Job objects are present and makes a choice for the Job objects `top`, `mid`, and `low` via the 3 assignments `{top|mid|low}:=Try(...)`. Note that we have 3 choices for `top`, 2 choices for `mid`, and 1 choice for `low`. This makes 6 choices and backtracking possibilities for these Job assignments. Now consider the salary assignments. For every salary we offer 3 possibilities, which makes $3 * 3 * 3 = 27$ cases for the salary assignments. In the total, we get $6 * 27 = 162$ combinations and snapshots. None of them satisfies the negated invariant `bossBossBetterPaidThanWorker`. This certifies that it is not possible to construct a snapshot with three hierarchically arranged BossWorker links such that the bossBoss is less paid than the worker or equally paid to the worker. By certification we refer to the process that the developer gains more insight into her UML and OCL model and convinces herself through a test or validation case that her model possesses a certain property. We are aware of the fact that this process is not a complete formal proof, but nevertheless the intuitively suspected property of the modeler is formally checked by means of USE and ASSL against the scenarios the developer regards as relevant.

Up to now we have used `Try` with its backtracking possibilities for attributes only. But one can use backtracking for link generation as well. It is also possible to use `Try(BossWorker, ...)` instead of `Insert(BossWorker, ...)`. If we use `Try`, we have to give for each argument of the association a list of possible objects like `Try(BossWorker, [Sequence{top}], [Sequence{mid}])`. The USE system then generates all possible link combinations for the complete association. In the example above, this would be the empty link set and the link set consisting of the single pair having the current `top` value as first and the current `mid` value as second component. Backtracking is enabled again, if we use `Try` instead of `Insert`.

Above in the example we have used concrete values like `2000` or `3000`. But the argumentation will be the same if we use instead abstract values and abstract interpretation techniques. This is subject to future research.

5 Formal Semantics of ASSL

This section surveys the semantics of ASSL. Subsection 5.1 describes a transformation we have to perform on ASSL procedures before the proper semantic function can be applied. This transformation resolves nested ASSL commands. Subsection 5.2 afterwards characterizes the basic steps to be taken for the denotational semantics. After explaining the used formalization of OCL we map commands and command sequences into configurations and study how snapshot sequences result from evaluating ASSL procedures.

5.1 Unnesting of ASSL Expressions

The syntax of ASSL allows nested ASSL expressions. For example, in the procedure `generateCompanies` the variable `theCompanies` can be omitted, if we include the `CreateN(...)` command directly in the loop:

```
for p:Company in (CreateN(Company,[count]))
  begin ... end
```

For the semantics of ASSL we only consider simple and unnested ASSL expressions, thus the first step of defining the semantics is to translate a given ASSL procedure with nested ASSL expressions to a semantically equivalent procedure of an ASSL core having no nested ASSL expressions. Because this transformation is simple and because we want to focus on a precise denotational semantics, this transformation is explained by an example only. In procedure `generateCompanies` the body of the loop

```
[p].name:=Any([Sequence{'AMD','DEC','HP','IBM','Lex','Miro',
  'NEC','SAP','Sony','Sun'}
  ->reject(n1|Company.allInstances.name->exists(n2|n1=n2))]);
```

is translated to:

```
vv:=Any([Sequence{'AMD','DEC','HP','IBM','Lex','Miro',
  'NEC','SAP','Sony','Sun'}
  ->reject(n1|Company.allInstances.name->exists(n2|n1=n2))]);
[p].name:=[vv];
```

The translation has inserted a new variable `vv` which must defined in the declaration section of the procedure.

A procedure of the ASSL core only has commands given in Appendix B. In the next section we define a denotational semantics for these single commands and finally for procedures of the ASSL core.

5.2 Denotational Semantics

Formalization for OCL Expressions: An ASSL procedure with parameters is executed in the context of a given UML class model, an initial snapshot, and actual parameter values. An ASSL procedure may contain OCL expressions denoted within squared brackets. Thus, in order to develop a semantics for ASSL we first need a formalization for UML class models, snapshots, and OCL expressions.

The required definitions, especially those for the semantics of OCL expressions, are given in [RG98]. There, a snapshot (system state) S is defined as a labeled hypergraph (V, E, *nodes*, *roles*, *ostate*) where V is a finite set of nodes (objects, i.e., instances of a class), E is the finite set of edges (links, i.e., instances of an association), *nodes* is a function connecting edges with nodes, *roles* is a function labeling each edge with a tuple of role names and *ostate* is a function labeling each node with an attribute-value mapping. For example, the snapshot represented in Fig. 5 is defined as:

```
V = { Person1, Person2, Person3, Job1, Job2, Job3, Company1 }
E = { J1P1, J2P2, J3P3, C1J1, C1J2, C1J3 }
nodes(J1P1) = ( Job1, Person1 )
nodes(J2P2) = ( Job2, Person2 )
nodes(J3P3) = ( Job3, Person3 )
nodes(C1J1) = ( Company1, Job1 )
nodes(C1J2) = ( Company1, Job2 )
nodes(C1J3) = ( Company1, Job3 )
roles(J1P1) = roles(J2P2) = roles(J3P3) = (job, employee)
roles(C1J1) = roles(C1J2) = roles(C1J3) = (employer, job)
ostate(Person1) = { (name, 'Lisa' ) }
ostate(Person2) = { (name, 'Fred' ) }
ostate(Person3) = { (name, 'Wini' ) }
ostate(Job1) = ostate(Job2) = ostate(Job3) = { (salary, undefined ) }
ostate(Company1) = { (name, 'Lex' ) }
```

In [RG98], a definition for the semantics of OCL expressions is given as well. Please note that the additional semantics in this paper is based on that work and we do not repeat all details from that paper here. Let B be the set of all variable assignments and C the set of all classifiers. The semantics of an OCL expression $e \in \mathit{Expr}(c)_s$ in the context of a class $c \in C$ is a function $I[e] : B \times C \to I(s)$ in [RG98].

With little modification we can reuse these definitions for the semantics of ASSL procedures. The main difference here is that OCL expressions in ASSL procedures are not evaluated in the context of a class c. But we can achieve this, if we formally add a new class without associations, attributes and operations to the class diagram, create an instance of this new class, and provide the class c as the second parameter in $I[e]$. However, we ommit this second parameter in the following definitions and use it as $I[e] : B \to I(s)$.

Note that this new function $I[e] : B \to I(s)$ does not have the snapshot as a parameter. In [RG98] the snapshot is given by the frame of the definitions, but we need it explicitly as a parameter. In addition to the present notions, let *Snapshot* be the set of all snapshots of a given class diagram. Now, fitting for our purpose, the semantics of an OCL expression $e \in \mathit{Expr}(c)_s$ can be defined by a function $I[e] : B \times \mathit{Snapshot} \to I(s)$.

Formalization of Commands: The next step is to formalize ASSL procedures. The body of an ASSL procedure is a sequence of ASSL commands. Let *Command* be the set of commands defined by the ASSL core. The set of all command sequences will further on be called $\mathit{Command}^*$.

Now let $\mathit{proc} \in \mathit{Command}^*$ be the command sequence of an ASSL procedure and let $\beta_0 \in B$ be the initial variable assignment. This means, this assignment associates the actual values of the procedure call with the parameter variables of the ASSL procedures. Because *proc* is evaluated on an initial snapshot $\sigma_0 \in \mathit{Snapshot}$ and an initial variable assignment β_0, the semantics of *proc* is a function:

$$M : (\mathit{Command}^* \times B \times \mathit{Snapshot}) \to \mathit{Snapshot}^*$$

This function M produces the sequence of snapshots needed for automatic snapshot generation. The goal of the next sections is to define the function M for $M(\mathit{proc}, \beta_0, \sigma_0)$.

Evaluation of ASSL Commands on a Configuration K: ASSL commands are evaluated in the context of a snapshot and a variable assignment. For example, the first command in *proc* is evaluated in the context of β_0 and σ_0. Such a tuple (β, σ) is called *configuration*. Let K be the set of configurations.

Because commands such as `Try` may produce many configurations, the semantics of a command $i \in \mathit{Command}$ is a function A returning sequences of configurations, i.e., elements of K^*:

$$A[i] : K \to K^*$$
$$A : \mathit{Command} \to (K \to K^*)$$

Now we define the semantic function A for the commands of the ASSL core. Here, in the running section, we only define A for the variable assignment *var*`:=[`*expr*`]`.

The semantics of the other commands is given in Appendix B. Let $expr \in Expr_s$ be an OCL expression and let $var \in Var_s$ be a variable with $s \in S_{Expr}(S)$. We now consider $oclexpr(expr, var) \in Command$.

$$A[oclexpr(expr, var)](\beta, \sigma) =< (\beta_{[var/I[expr](\beta,\sigma)]}, \sigma) >$$

The command `var:=[expr]` produces a sequence with one configuration, which has an unchanged snapshot σ and a changed variable assignment $\beta_{[var/I[expr](\beta,\sigma)]}$. That means, the value of var will be equal to the evaluation result of the OCL expression $expr$ in the context of (β, σ).

Evaluation of ASSL Commands on a Configuration Sequence K^*: Starting with an initial configuration (β_0, σ_0) the first command of the ASSL procedure produces a sequence of configurations. The second command must now be evaluated in the context of *each* configuration. So we would get a sequence of sequences of commands. n commands are producing a configuration tree with at least n levels. In order to get the configuration of a level as a flattened sequence we need a function $flatten : (K^*)^* \to K^*$ defined as:

$$\begin{aligned} &flatten(<< k_{1,1}, \dots, k_{1,n_1} >, < k_{2,1}, \dots, k_{2,n_2} >, \cdots, < k_{m,1}, \dots, k_{m,n_m} >>) = \\ &\qquad < k_{1,1}, \dots, k_{1,n_1}, k_{2,1}, \dots, k_{2,n_2}, \cdots, k_{m,1}, \dots, k_{1,n_m} > \end{aligned}$$

Let $i \in Command$ be given. The semantics of a command evaluated on a configuration sequence is defined as:

$$\begin{aligned} &Q[i] : K^* \to K^* \\ &Q[i] < k_1, \dots, k_n > = flatten(A[i](k_1), \dots, A[i](k_n)) \end{aligned}$$

Now the semantics of a command sequence can be defined using recursion. Let $seq \in K^*$ and $i_j \in Command$ with $j \in \{1, \dots, n\}$ be given.

$$\begin{aligned} &Q[(< i_1, \dots, i_n >)](seq) = \\ &\quad \begin{cases} seq & \text{if } n = 0 \\ Q[i_n](Q[< i_1, \dots, i_{n-1} >](seq)) & \text{if } n > 0 \end{cases} \end{aligned}$$

Semantics of an ASSL Procedure: The semantic function M only needs the snapshots of the produced configuration sequence, which are collected by the function $snaps : (B \times Snapshot)^* \to Snapshot^*$

$$snaps(< (\beta_1, \sigma_1), (\beta_2, \sigma_2), \dots, (\beta_n, \sigma_n) >) =< \sigma_1, \sigma_2, \dots, \sigma_n >$$

Summarizing we can say that the semantics of an ASSL procedure consisting of the command sequence $proc \in Command^*$ applied to the initial variable assignment β_0 and the initial state σ_0 is:

$$M(proc, \beta_0, \sigma_0) = snaps(Q[proc](<(\beta_0, \sigma_0)>))$$

6 Conclusion

We have presented an extension of the USE tool for automatic generation of complex snapshots, i.e., system states consisting of objects possessing attribute values and links. Our approach employs OCL expressions for various tasks apart from traditional OCL tasks like definition of operations, invariants, pre- and postconditions: (1) OCL expressions are used in ASSL procedures (1.a) to reduce the search space for snapshots and (1.b) to formulate properties of the desired snapshot, and (2) OCL invariants are dynamically loaded during command file execution (2.a) to guide the snapshot search and (2.b) to certify an invariant implication. It is an interesting observation for us that a descriptive language like OCL can be used in task (1.a) for improving efficiency.

The presented approach can be seen in two ways: First, as a means for purely generating consistent data constellations and consistent snapshots, and second, as a possibility for providing an abstract, i.e., programming-independent, implementation of operations in UML class diagrams. This paper has been focussing on the first aspect. Regarding the second aspect, one could use ASSL to systematically implement each operation from the UML class diagram by writing a procedure for each operation. The possibilities of ASSL go far beyond the current possibilities provided by USE command files for operation implementation.

Our approach is scalable with respect to larger snapshots. We have employed the approach in various case studies and have constructed, for example, snapshots with about 200 objects and 400 links in acceptable execution time (about 30 seconds). Our motivation for the development of complex test cases in early development stages is to ensure the quality of the developed models and to give early feedback to the developer. We also see the possibility to carry over test and validation cases from early phases to the later implementation phases. For example, the 162 snapshots from the above validation case correspond to 162 negative test cases which could also be run against the implementation. Thus, the effort spent in the early phases will be rewarded by high quality results in later phases.

Acknowledgment

The authors would like to thank the referees for their fruitful and constructive criticism and their overall helpful remarks.

References

[ABB+00] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I.P. de Guzmán, G. Brewka, and L. M. Pereira, editors, *Proc. 8th European Workshop Logics in AI (JELIA'2000)*, LNCS 1919, pages 21–36. Springer, Berlin, 2000.

[AL99] Sten Agerholm and Peter Gorm Larsen. A Lightweight Approach to Formal Methods. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullmann, editors, *Proc. Int. Workshop Current Trends in Applied Formal Methods (FM-TRENDS'1998)*, LNCS 1641, pages 168–183. Springer, Berlin, 1999.

[Amb97] Scott W. Ambler. *Building Object Applications: Patterns, Architecture, Design, Construction, and Testing.* Prentice Hall, 1997.

[AO00] Aynur Abdurazik and Jeff Offutt. Using UML collaboration diagrams for static checking and test generation. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*, LNCS 1939, pages 383–395. Springer, 2000.

[Arg03] ArgoUML Team. The ArgoUML Tool. `www.argouml.tigris.org`, 2003.

[Boh01] Jörn Bohling. Generation of Snapshots for the Validation of UML Class Diagrams (In German). Diploma Thesis, University of Bremen, 2001.

[Bol02] Boldsoft. The Boldsoft OCL Tool Model Run. `www.boldsoft.com`, Boldsoft, Stockholm, 2002.

[CCJ+03] Augusti Canals, Yannick Cassaing, Antoine Jammes, Laurent Pomies, and Etienne Roblet. How You could Use NEPTUNE in the Modelling Process. *Journal of Object Technology*, 2(1):69–83, 2003. `www.jot.fm`.

[Chi01] Dan Chiorean. Using OCL Beyond Specifications. In Andy Evans, Robert France, Ana Moreira, and Bernhard Rumpe, editors, *Proc. UML'2001 Workshop Rigorous Development*, pages 57–68. Lecture Notes in Informatics (LNI), German Informatics Society, 2001.

[DdB00] S. Dupuy and L. du Bousquet. A multi-formalism approach for the validation of UML models. *Formal Aspects of Computing*, 12(4):228–230, 2000.

[FL00] Peter Fröhlich and Johannes Link. Automated Test Case Generation from Dynamic Models. In Elisa Bertino, editor, *Proc. 14th European Conf. Object-Oriented Programming (ECOOP'2000)*, pages 472–491. Springer, Berlin, LNCS 1850, 2000.

[GR00] Peter Graubmann and Ekkart Rudolph. Hypermscs and sequence diagrams for use case modelling and testing. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*, LNCS 1939, pages 32–46. Springer, 2000.

[HDF00] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proc. 3rd Int. Conference Unified Modeling Language (UML'2000)*, LNCS 1939, pages 278–293. Springer, 2000.

[JGP99] Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneach. Validating distributed software modeled with the Unified Modeling Language. In Jean Bézivin and Pierre-Alain Muller, editors, *Proc. 1st Int. Workshop Unified Modeling Language (UML'1998)*, LNCS 1618, pages 365–377. Springer, 1999.

[JSS00] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: The Alloy constraint analyzer. In *Proc. Int. Conf. Software Engineering (ICSE'2000)*, pages 730–733. ACM, New York, 2000.

[Mar99] H. Martin. Using test hypotheses to build a UML model of object-oriented smart card applications. In Jean-Claude Rault, editor, *Proc. Int. Conf. Software and Systems Engineering and their Applications (ICSSEA'1999)*, 1999.

[Mut00] Darmalingum Muthiayen. *Real-Time Reactive System Development - A Formal Approach Based on UML and PVS*. PhD thesis, Department of Computer Science at Concordia University, Montreal, Canada, January 2000.

[OA99] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In Robert France and Bernhard Rumpe, editors, *Proc. 2nd Int. Conf. Unified Modeling Language (UML'1999)*, LNCS 1723, pages 416–429. Springer, Berlin, 1999.

[OK99] I. Oliver and S. Kent. Validation of object-oriented models using animation. In *Proc. EuroMicro, Vol. 2*, pages 2237–2243. IEEE, Los Alamitos, 1999.

[OMG03] OMG, editor. *OMG Unified Modeling Language Specification, Version 1.5*. OMG, March 2003. OMG Document `formal/03-03-01`, `www.omg.org`.

[RG98] Mark Richters and Martin Gogolla. On Formalizing the UML Object Constraint Language OCL. In Tok-Wang Ling, Sudha Ram, and Mong Li Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'1998)*, pages 449–464. Springer, Berlin, LNCS 1507, 1998.

[RG00] Mark Richters and Martin Gogolla. Validating UML Models and OCL Constraints. In Andy Evans and Stuart Kent, editors, *Proc. 3rd Int. Conf. Unified Modeling Language (UML'2000)*, pages 265–277. Springer, Berlin, LNCS 1939, 2000.

[RG01] Mark Richters and Martin Gogolla. OCL - Syntax, Semantics and Tools. In Tony Clark and Jos Warmer, editors, *Advances in Object Modelling with the OCL*, pages 43–69. Springer, Berlin, LNCS 2263, 2001.

[WK98] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

Appendix A: `generateBossWorker`

```
procedure generateBossWorker(count:Integer)
var top:Job, low:Job, depth:Integer, addition:Integer;
begin
for i:Integer in [Sequence{1..count}]
  begin -- establish BossWorker links
  top:=Try([Job.allInstances->asSequence]);
  low:=Try([Job.allInstances->asSequence->reject(low|
    low.boss.isDefined or low=top or low.workerPlus()->includes(top) or
    top.workerPlus()->includes(low) or top.employer<>low.employer)]);
  Insert(BossWorker,[top],[low]);
  end;
depth:=[0]; -- calculate depth of BossWorker hierarchy
for j:Job in
  [Job.allInstances->select(j|j.salary.isUndefined)->asSequence]
```

```
  begin depth:=[if j.bossPlus()->size>depth
                  then j.bossPlus()->size else depth endif];
  end;
for j:Job in -- establish salaries using depth and random values
  [Job.allInstances->select(j|j.salary.isUndefined)->asSequence]
  begin addition:=Any([Sequence{0,500,1000,1500}]);
  [j].salary:=[1000+2000*(depth-j.bossPlus()->size)+addition];
  end;
end;
```

Appendix B: Semantics of Further Single ASSL Commands

This appendix contains the semantics of single ASSL commands $i \in Command$ by defining $A[i] : K \to K^*$. We are defining the semantics for various commands showing that the formalization is able to incorporate the main features of ASSL which are (A) backtracking (by building a configuration tree), (B) snapshot changes, and (C) variable assignment changes.

• **Variable assignment with an OCL expression:** `var := [ expr ]`

Let $expr \in Expr_s$ be the OCL expression and let $var \in Var_s$ be the variable with $s \in S_{Expr}(S)$. Let $oclexpr(expr, var) \in Command$.

$$A[oclexpr(expr, var)](\beta, \sigma) =< (\beta_{[var/I[expr](\beta,\sigma)]}, \sigma) >$$

• **Variable assignment with `Create`:** `var := Create( c )`

Let $c \in C$ be the class of the object to create. Let $s \in S_{Expr}(S)$ be the object type of class c. Let $var \in Var_s$. $o \in I(c)$ is a object identifier. Let $create(c, var) \in Command$.

$$\begin{aligned} A[create(c, var)]\ (\beta, (V', E, nodes, roles, ostate)) = \\ < (\beta_{[var/o]},\ (V \cup \{o\}, E, nodes, roles, ostate)) > \\ \text{with } o \in I(c) \setminus V \end{aligned}$$

• **Insertion of a link:** `Insert( A , var`$_1$ `, ... , var`$_n$ `)` $(n \in N,\ n > 1)$

Let $s_i \in S_{Expr}(S)$ be the object type of the class of the i^{th} association end of association A. Then it is $var_i \in Var_{s_i}$. Now let $rn_i \in A_{role}$ be the role name of the i^{th} association end of A. $(i \in N,\ i \leq n)$. Let $insert(< rn_1, \ldots, rn_n >, < var_1, \ldots, var_n >) \in Command$.

$$A[insert(< rn_1, \ldots, rn_n >, < var_1, \ldots, var_n >)]$$
$$(\beta,\ (V, E, nodes, roles, ostate)) =$$
$$< (\beta,\ (V, E \cup \{e\}, nodes', roles', ostate)) >$$
$$\text{with } nodes' = nodes \cup \{e \mapsto\ < I[var_1](\beta, \sigma), \ldots, I[var_n](\beta, \sigma) >\}$$
$$\text{and } roles' = roles\ \cup \{e \mapsto\ < rn_1, \ldots, rn_n >\}$$
$$\text{and } e \notin E$$

- **Variable assignment with `Try`: `var := Try( seqvar )`**

Let $seqvar \in Var_{Sequence(s)}$ and $var \in Var_s$ with $s \in S_{Expr}(S)$. Let $try(seqvar, var) \in Command$.

$$A[try(seqvar, var)]\ (\beta, \sigma) = < (\beta_{[var/e_1]}, \sigma), \sigma), \ldots, (\beta_{[var/e_n]}, \sigma) >$$
$$\text{with } \beta(seqvar) = < e_1, \ldots, e_n >$$

- **Variable assignment with `Any`: `var := Any( seqvar )`**

Let $seqvar \in Var_{Sequence(s)}$ and $var \in Var_s$. $(s \in S_{Expr}(S))$. Let $any(seqvar, var) \in Command$. Let $random : Var_{Sequence(s)} \rightarrow I(s)$

$$A[any(seqvar, var)]\ (\beta, \sigma) = < (\beta_{[var/random(seqvar)]}, \sigma) >$$

The function *random* is a function which could be defined as an OCL term with `seqvar->asSet->asSequence->at(1)`. Because a set has no order, the term `asSequence->at(1)` returns a random element of *seqvar*. For a deterministic and exact semantics of *random* the configuration $(\beta, \sigma) \in K$ must be extended with a third element $(\beta, \sigma, path) \in K$, where $path \in N^*$ represents the path in the configuration tree. For example, $path =< 1, 3, 8, 4 >$ refers to the 4^{th} subconfiguration of the 8^{th} subconfiguration of the 3^{rd} subconfiguration of the initial configuration. The deterministic function $random(seqvar, path)$ now returns an element of *seqvar* depending on *path*. Let $rand : N \times N^* \rightarrow N$ be a given function with $1 \leq rand(max, path) \leq max$, then $random(seqvar, path) = random(< e_1, e_2, \ldots, e_n >, path) = e_{rand(n,path)}$.

A configuration K is defined as a tuple $(\beta, \sigma) \in K$ without *path* in order to reduce the complexity of the semantics.

- **Attribute assignment: `[ objectvar ]. attrname`$_c$ `:= [ valuevar ]`**

Let $c \in C$, $objectvar \in Var_c$, $valuevar \in Var_s$ and $(attrname_c : c \rightarrow s) \in ATT_c$. Let $attrAssign(objectvar, attrname_C, valuevar) \in Command$ and $\sigma = (V, E, nodes, roles, ostate)$.

$$A[attrAssign(objectvar, attrname_C, valuevar]\ (\beta, \sigma) =$$
$$< (\beta, (V, E, nodes, roles, ostate')) >$$

$$\text{with } ostate'(v, a_c) = \begin{cases} I[valuevar](\beta, \sigma) & \text{if } I[objectvar](\beta, \sigma) = v \\ & \quad \wedge\ attrname_C = a_c \\ ostate(v, a_c) & \text{else} \end{cases}$$

$(v \in V, a_c \in ATT_c)$