A computer system contains file systems. A set of different users can access the computer system ressources generically named folderItems. Users may own or only access folderItems (files or folders). Each user has an acces permission for each forderItem. Each folderItem has a name which is unique in its namespace and a size. Let suppose that the permission to access a fileItem can be: write, read and so forth. Each folderItem knows its owner which is a user. Folders and files are structured in a hierarchical manner. Each folder may have attached a file system. Each file system may refer many folders.

b. Using the OCL, please specify an invariant requiring that all ownedItems by a user have the permission access write.
c. Using the OCL, please specify an observer infering the Set of tuple type of users having the biggest sum of ownedItems size.

```
context User

    inv ownedItems:

        self.accesableItems->select(fI:FolderItem | fI.permission.value =
PermValue::write)->includesAll(self.ownedItems)

context ComputerSystem

def impUsers:

        let bigestSize:Integer = self.users->collect(us:User |
Tuple{u=us,ts=us.ownedItems.size->sum})->sortedBy(t | t.ts)->last.ts

        let impUsers:Set(TupleType(u:User,ts:Integer)) = self.users->

collect(us:User | Tuple{u=us,ts=us.ownedItems.size->sum})->select(t | t.ts =
bigestSize)->asSet
```

A Sudoku puzzle is shown in the figure below. To complete this puzzle requires the puzzler to fill every empty cell with an integer between 1 and 9 in such a way that every number from 1 up to 9 appears once in every row, every column and every one region of the small 3 by 3 boxes highlighted with thick borders. Each cell corresponds to a rowIndex, which represents the row of the cell, a columnIndex, which represents the column of the cell and has a value, which represents the content of the cell. A cell belongs to a region; each region is defined by the topLeft cell and by the contained cells. Initially, the puzzle contains predefined cells whose values remain unchanged and by empty cells, also named whiteCells. During the game, the puzzler starts by setting the value of empty cells trying to comply with the constraints mentioned in the beginning. Once the value of an empty cell was set, the type of the cell is considered to be a new kind, named PotentialValue. If later in the game the puzzler notices that the value of the PotentialValue cell is incorrect, then this must be changed. In order to support new potential changes of the value, the player keeps changes history in a sequence of so named ChangedValue cells.

b. Using OCL please specify in the context of a cell an invariant constraining the value of the cell which is not empty, to be unique both on his row, column and region. 1 pt

```
context Cell
    inv uniqueValue:
        if self.value.isUndefined
            then true
            else
                (self.row.cell->reject(c:Cell     |     c.value.isUndefined)-
>select(c:Cell | c.value = self.value)->size = 1 and
                self.column.cell->reject(c:Cell    |    c.value.isUndefined)-
>select(c:Cell | c.value = self.value)->size = 1 and
                self.region.containedCells->reject(c:Cell |
c.value.isUndefined)->select(c:Cell | c.value = self.value)->size = 1)

        endif
```

A faculty has a set of admitted students and a set of teachers which give courses. Each student is inscribed at a part of courses proposed by the faculty. At the end of each course, students inscribed are evaluated. The evaluation result is an integer mark in the interval {1, 2, …, 9, 10}. The evaluations are done at a planned date, usually by teachers giving the courses. If a student is absent at an evaluation, then his mark will be 0 (by default). Students and teachers are characterized by names and Ids. Students and teachers Ids values are unique in the context of a faculty. Courses are characterized by name and credits which are natural numbers in the interval {1, ..., 6}. The faculty has a name.

b. Using the OCL, please complement the model represented by the class diagram with:

I. an invariant specified in the Student class which states that each student Id must be unique in the context of the faculty,
II. another invariant specifying that teachers doing the evaluation are among teachers giving the course associated with the evaluation,
III. an observer returning true if a student passed (marks >= 5) all evaluations of his courses.

```
  context Student
      inv uniqueStudId:
          self.faculty.students.id->count(self.id) = 1

  context Evaluation
      inv aprTeachers:
          self.courses.teachers->includesAll(self.evaluationTeachers)

  context Course
      inv courseEvaluationTeachers:
          self.teachers->includesAll(self.evaluation.evaluationTeachers)

  context Student
      def passedCourseEvaluations:
          let passCE:Boolean = self.courses->reject(c | c.evaluation->exists(e | e.mark >= 5 and e.students=self))->isEmpty

context Student
      def passedCourseEvaluations2:
          let passCE2:Boolean = self.courses->reject(c | c.evaluation.mark->exists(m | m >= 5))->isEmpty
```

Let us consider a network containing nodes and links between two nodes. A network may contain also other networks, nodes, and links which connect two nodes. Networks are organized in a structured manner. There is a unique root network which is not contained in any other network. Excepting the root network, all the other networks are directly contained (owned) by a network. Nodes can/may be connected by links. Links may be included in a network or the may connect two different networks.

b. Using the OCL, please specify an invariant imposing that in case of links connecting two networks, the nodes of the link are from the two networks connectd.

c. Using the OCL, please specify in the context Network observers returning the root network, and the set of all networks contained in the nested networks in which is included the current network.

```
context Link

    inv linkBetween2Networks:

        self.nodes.networkContainer->asSet->size = 2

context Network

    def allNetworks:

        let allChildNetworksFL:Set(Network) = (self.containedModules->select(m:Module | m.oclIsTypeOf(Network)).oclAsType(Network))->asSet

        let allChildNetworksDE:Set(Network) = allChildNetworksFL->union(allChildNetworksFL->closure(n:Network | n.allChildNetworksFL))

        let networkRoot: Network =  if self.networkContainer->isEmpty
                                        then self
                                        else self.networkContainer.networkRoot
                                    endif

        let allNetworks:Set(Network) = networkRoot.allChildNetworksDE->including(networkRoot)
```

Persons may have as author or editor different kind of publications – (both roles are excluded). Each publication has a title, a publication date and is associated with a publisher which has a name. Persons are characterized by name and affiliation. There are different kind of publications: article, journal, conference paper, book, proceedings, compendium and compendium chapter. Articles written by author(s), are included in a journal and are characterized by their first and last page. Journals are edited by editors and are characterized by volume and number. A book is written by author(s) and is featured by the number of pages. Compendium and proceedings are two special kind of books. A compendium contains a set of compendium chapters, each featured by a chapter number, a first and last page. By consequence, each compendium is edited (by editors) and each compendium chapter is authored. Proceedings contain conference papers accepted and presented to a conference. Proceedings are edited (by editors) and are featured by the conference name, conference's start and end date. Each conference paper is characterized by first and last page.

b. In the context of Publication class, please specify an invariant stating that each publication may authored exclusive or edited, and in case in which is edited, the publication kind must be one of the: Journal, Compendium, Proceedings.

c. In the context of Person, please specify an observer returning the articles authored by the current person as unique author.

```
context Publication

    inv editors:

        (not    self.editors->isEmpty    implies    ((self.oclIsTypeOf(Journal)    xor
self.oclIsTypeOf(Compendium))    xor    self.oclIsTypeOf(Proceedings)))    xor    (not
self.authors->isEmpty)


context Person

    def pubWithUniqueAuth:

      let articlesWithUniqueAuth:Set(Article) = self.publications->select(p:Publication

          | p.authors->size = 1 and p.oclIsTypeOf(Article)).oclAsType(Article)->asSet
```

A person may be employed by companies. An employed person has a job at each employer. Any job is characterized by a title {designer, programmer, tester, other}, a start date and a salary. A person has a name, a birth date, a gender {female, male}, an information confirming that he/she is married. Each company is characterized by its name and maximum numberOfEmployee. Each married person knows the identity of the partner (wife or husband).

b. using OCL, please specify in the appropriate context an invariant checking that any employee of a company having a title different from designer cannot have a salary bigger than any other employee from the same company having the title designer;

```
context Person
    inv isMaried:
        self.isMaried implies
           if sex = Gendre::female
               then self.husband.sex = Gendre::male
               else self.wife.sex =  Gendre::female
           endif

context Company
    inv designersSalary:
        let lowestDesignerSalary:Integer = self.job->select(j:Job | j.title = JobTitle::designer).salary->sortedBy(s:Integer| s)->first
        let highestNonDesignersSalary:Integer = self.job->reject(j:Job | j.title = JobTitle::designer).salary->sortedBy(s:Integer | s)->last in
        lowestDesignerSalary > highestNonDesignersSalary

context Company
    def numOfEmployedFamilies:
        let numOfEmployedFamilies:Integer = self.employee->select (e | e.isMaried and e.sex = Gendre::male)->select( e | e.husband.employer->includes(self))->size
```

An airline operates flights. Each airline has an ID and a name.
Each flight has an ID a departureAirport and an arrivalAirport, a departureTime, an arrivalTime, a date and a status (onGround, onAir, landed). Each flight has a pilot and a least a coPilot, and uses an aircraft of a certain type, AircraftType. An airline owns a set of aircrafts of different types. Each aircraft ha a name and a state (ready, inRevision, inRepair). A company has a set of pilots: each pilot has anexperience level and a set of skils. There are 3 categories of pilots: 1, 2 and 3. Pilots cat 3 are the most experienced pilots. The captain of an aircraft must be a pilot category 3. Each type of aircraft needs a number of pilots. A captain and one or morecoPilots. All the pilots of an aircraft must be certified for the aircraftType corresponding to the aircraft.

b. Using OCL, please specify an observer returning the set of tuples (airport, nbOfDepartures) of airline airports having the most departures.

```
context Flight
    inv apropiatePilots:
        self.drivenBy->select(p | p.oclIsTypeOf(PilotCat3)).oclAsType(PilotCat3).captainOf->includesAll(self.drivenBy.navigatorOf)
        and (self.drivenBy.navigatorOf->reject(at | at.name=self.uses.aicraftType.name))->isEmpty

context Airline
    def airportsWithMostDepartures:

        let airports:Set(Airport)=self.flights.departsFrom->asSet
        let airportsNbDepartures:Set(TupleType(arp:Airport, fn:Integer))=airports->collect( a | Tuple{arp=a, fn=a.flightD->size})->asSet
        let anAirportWithMostDepartures:TupleType(arp:Airport, fn:Integer)=airports->collect( a | Tuple{arp=a, fn=a.flightD->size})->sortedBy(t | t.fn)->last
        let airportsWithMostDepartures:Set(TupleType(arp:Airport, fn:Integer))=airportsNbDepartures->select(t | t.fn = anAirportWithMostDepartures.fn)
```

A library has a catalog of books, each of which has a number of copies (we allow the fact that there might not be any copies of a book), a number of members and a way of keeping track of current and historical loans of those members. At any moment, a member may have at most 3 current loans, each corresponding to a copy. When a loaned copy is not returned in time, an instance of historical loan ponting towards that copy will be created and the corresponding current loan will be deleted. After returning a loaned copy, the state of that copy will be updated at available. Each book has a title and a bookId which is unique. A copy has a copyID, a state, a bReserve and an eReserve attributes. A state can be: available, loaned or reserved. When a currentLoan is created, the state of the associated copy will be modified at loaned. A loan is characterized by the attributes loanDate, returnDate and numberOfDays (representing the number of days for that loan). The value of returnDate will be updated at the currentDay of the returning day, irrespective if it's about a currentLoan or a historicalLoan.

Using OCL, please specify:
a) An invariant requiring that the books associated to the libray have a unique bookId. When the invariant is violted, the specification must support to infer the set of books having the same bookId. 1p
b) A precondition attached to the Member::loan(bId:String, bDate:Date, nD:Integer), where bId is the bookId of the book associated with the copy we intent to loanm, bDate – the beginData of the loan and nD – the number of days of the loan. The precondition cmust check that there are not historicalLonas unreturned and that the number of currentLoans are less than 3. Another constraint is that there is at least an available copy of the book we intent to loan.

```
context Member::loan( bId:String, lDate:Date, nD:Integer)
    pre loanIsPossible:
        if self.historicalLoans->size > 0 then not self.historicalLoans->exists(l | l.returnDate.isUndefined) else true endif and
        self.currentLoans->size < 3 and
        self.memberOf.catalog->any(b | b.bookId = bId).copies->exists(c | c.state = CopyState::available)


context Book
    inv uniqueBookId:
      let bookId:Bag(String) = self.library.catalog.bookId in
        if self.library.catalog->reject(b:Book | b.bookId <> self.bookId)->size = 1
            then true
            else false
        endif
```

A state Machine contains states, transitions between states, events triggering transitions and guards that can be associated to states. States are named elements. Guards are assertions that enable transitions if and only if the mentioned event appears and the guard is evaluated at **true**. Like guards, events are associated to transitions. There are different kind of states: SimpleStates, CompositeStates (that may contain other states) and PseudoStates. There are three different kinds of PseudoStates. InputStates, OutputStates and HistoryStates. Each HistoryState has a reference towards the last SimpleState from which a transition was triggered when an event appeared at the level of the CompositeState in which the HistoryState is nested.

b. Using the OCL, please specify appropriate invariants in classes: OutputState and InputState mentioning that: there are not transitions starting from outputStates, only transitions entering in outputStates; and only a transition starting from each inputState. There are not transitions entering in inputStates.

c. In the context of the CompositeState please specify an observer computing the number of simpleStates contained in that compositeState.

```
context OutputState
    inv: not self.incoming->isEmpty and self.outgoing->isEmpty


context InputState
    inv: self.incoming->isEmpty and self.outgoing->size = 1

context CompositedState
    def numOfSimpleStates:
    let numOfSimpleStates:Integer = self.containedStates->select(s | s.oclIsKindOf(SimpleState))->size
```