# MIEIC – Formal Methods in Software Engineering 2018/19
## Mini-Test - 28/11/2018 - 90 minutes

| Name: | Number: |
|---|---|

**Notes:**

- **Wrong answers discount $\frac{1}{4}$ of the quotation.**
- **All the questions have identical quotation.**
- **Consultation is not allowed (except for the attached consultation sheet).**

**1.** Consider the following relations

A = {(a1,b1), (a2,b1), (a2,a1), (b1,a2)}
B = {(b1,b1),(a2,b1), (a2,a1), (b1,a2)}
C = {(a1),(b1)}

Which of the following is **TRUE**?

| | |
|---|---|
| | `(B :> C) = (A :> C)` |
| | `C.A = C` |
| | `C.^B = {(b1)}` |
| X | `iden in ^A` |
| | `None of the above` |

**2.** Consider the following relations

D = {(x1,y2), (x2,y3), (x2,y2), (x3,x1)}
E = {(x1),(x2),(y2)}
F = {(x1,y1), (x2,y1)}

Which of the following is **TRUE**?

| | |
|---|---|
| | `D.E = E.D` |
| | `[D ++ (E -> E)] = [E -> E]` |
| | `[(D - F):> E] = F` |
| | `*D = D.~F` |
| X | `None of the above` |

**3.** Consider relation **r:A->B | #r.B < #r[A]**
Which of the following is a relation that makes this property **TRUE**?

| | |
|---|---|
| | `some a:A \| #r[a] > 1` |
| | `some ~r.r - iden` |
| | `some r.~r & iden` |
| | `the first 2 are correct` |
| X | `None of the above` |

**4.** **[QUESTION CANCELLED, because should be (~r).r]** Consider relation **r:A->B | r.~r in iden**
Which of the following is a relation that makes this property **TRUE**

| | |
|---|---|
| | `sig A {r: one B}` |
| | `sig A {r: lone B}` |
| | `sig A {r: some B}` |
| | `sig A {r: set B}` |
| X | `the first 2 are correct` |

**5.** Which of the following is **FALSE** about generic relations **r** and **s**?

- [ ] `s.r = r[s]`
- [ ] `r.~r in iden` means `r is injective`
- [ ] `~r.r in iden` means `r is functional`
- [X] `~r in r` means that `r is transitive`
- [ ] `None of the above`

**6.** **"A Directory may have other Directories inside but not itself"**. What is the **WORNG** specification of this constraint?

- [x] `sig Directory {content: set Directory} fact {no content & this}`
- [ ] `sig Directory {content: set Directory - this}`
- [ ] `sig Directory {content: set Directory} {this not in content}`
- [ ] `sig Directory {content: set Directory} fact {no content & iden}`
- [ ] `None of the above`

**7.** Considering **sig Directory {content: set Directory}** and the property **"A Directory cannot be inside an inner Directory"**. Which of the following **does not check** if the property holds?

- [ ] `check {no iden & ^content}`
- [ ] `check {no d:Directory | d in d.^content}`
- [X] `check {all disj d1,d2:Directory | d1 not in d2.^content}`
- [ ] `check {all d:Directory | d not in d.^content}`
- [ ] `None of the above`

**8.** Consider **one sig Root extends Directory {}.** Which of the following does **NOT** guarantee that all Directories (except the Root) are reachable from the Directory Root ?

- [ ] `all d:Directory-Root | d in Root.^content`
- [ ] `Root.^content = (Directory - Root)`
- [X] `*content = (^content + iden - Root)`
- [ ] `no d:Directory-Root | d not in Root.^content`
- [ ] `None of the above`

**9.** How can you generate instances of this model with 4 levels deep?

- [ ] `fact {#(Root.^content-Root)=4} run {} for 5`
- [ ] `fact {all d:Directory | one d.content} run {} for 5`
- [ ] `run {} for 3 but exactly 5 Directory`
- [X] `fact {#(Root.^content-Root)=4} run {all d:Directory | lone d.content} for 5`
- [ ] `All of the above`

**10.** How can you generate the set of Directories that we need to traverse from the Root to achieve a specific directory?

- [ ]
```
pred path1[d:Directory] set Directory {
  {e:Directory | e in d.^(~content)}  }
```
- [ ]
```
fun path2[d:Directory]: set Directory {
  {e:Directory | e in d.^(~content)}  }
```
- [ ]
```
fun path3[d:Directory]: set Directory {
  { (^content).d }                    }
```
- [X] `Previous 2 options (path2 and path3) are correct`
- [ ] `None of the above`

Name: Number:

**11.** Assume the following Alloy definitions:

```
sig Task {}
sig Worker {}
sig Project {
   tasks: ? Task,              -- a project has one or more tasks
   workers: ? Worker,          -- a project has one or more workers
   dependencies: tasks->tasks -- pairs (t1,t2), meaning that t1
                               -- cannot start before t2 finishes, i.e.,
                                  -- t1 depends on t2
}{ -- dependencies are acyclic  }
```

What should be the multiplicity of **tasks** and **workers**?

[ ] set [x] some [ ] one [ ] lone

**12.** (Continued) Which expression correctly imposes the constraint above (dependencies are acyclic)?

- [x] `no ^dependencies & iden`
- [ ] `no *dependencies & iden`
- [ ] `not ^dependencies in iden`
- [ ] `not *dependencies in iden`

**13.** (Continued) Which of the following gives you the pairs (Project, Task) in which the Task does **NOT depend on** any other task

- [ ] `{p:Project, t:Task | no p.dependencies[t]}`
- [x] `tasks - dependencies.Task`
- [ ] `previous two are correct`
- [ ] `none of the above are correct`

**14.** (Continued) Assume we also define in Alloy:

```
sig ProjectState {
   project: Project,
   finished: set Task,
   ongoing: Task ? -> ? Worker -- pairs (t,w), meaning that t
                               -- is being carried out by w
}{
   -- a) finished and ongoing tasks must belong to the project
   finished + ongoing.Worker in project.tasks
   -- b) finished and ongoing tasks must be disjoint
   no finished & ongoing.Worker
   -- c) dependencies of ongoing tasks must be finished
   no project.dependencies[ongoing.Worker] & finished
   -- d) workers of ongoing tasks must belong to the project
   Task.ongoing in project.workers
}
```

What should be the multiplicity to write on both sides of the **ongoing** relation, so that a task cannot have multiple workers, and a worker cannot have multiple tasks?

[ ] set [x] lone [ ] some [ ] one

**15.** (Continued) Which constraint expression is **INCORRECT** above?

[ ] a) [ ] b) [x] c) [ ] d)

**16.** (Continued) Which one is **NOT** a valid pre-condition below?

```
-- Task t is started by worker w, changing project state from s to s'.
pred startTask[s, s': ProjectState, t: Task, w: Worker] {
   -- pre-conditions
   t in s.project.tasks                       -- a)
   t not in s.finished + s.ongoing            -- b)
   Task->w not in s.ongoing                   -- c)
   s.project.dependencies[t] in s.finished -- d)
   -- post-conditions
   s'.project = s.project and s'.finished = s.finished
   s'.ongoing = s.ongoing + t -> w
}
```

[ ] a) [ ] b) [x] c) [ ] d)

**17.** (Continued) Which one is **NOT** a valid pre or post-condition below?

```
-- Task t is finished by worker w, changing project state from s to s'.
pred finishTask[s, s': ProjectState, t: Task, w: Worker] {
   -- pre and post-conditins
   t->w in s.ongoing                  -- a)
   s'.project = s.project             -- b)
   s'.finished = s.finished + t       -- c)
   s'.ongoing = t <: s.ongoing :> w -- d)
}
```

[ ] a) [ ] b) [ ] c) [x] d)

**18.** (Continued) Which constraint is **NOT** correctly defined below?

```
open util/ordering[ProjectState]
...
-- The ordered instances of ProjectState must represent a valid
-- execution of a project.
fact validOrdering {
   -- a) in the first state, there are no finished tasks
   no first.finished
   -- b) in the first state, there are no ongoing tasks
   no first.ongoing
   -- c) consecutive states are related by startTask or finishTask
   s: ProjectState, s' : s.next | some t: Task, w: Worker |
      startTask[s, s', t, w] or finishTask[s,s', t, w]
   -- d) in the last state, all tasks are finished
   all last.finished
}
```

[ ] a) [ ] b) [ ] c) [x] d)

**19.** (Continued) Which command finds valid executions of a project with dependencies?

[x] `run {some dependencies} for 5`
[ ] `check {some dependencies}`
[ ] `run {one dependencies}`
[ ] `check {some dependencies} for 5`