A. OCL Rules

In this appendix we present the OCL rules that, together with the UML classes, define our models in DECOR.

A.1. MAPE-K Control Architecture Meta-Model

The following OCL rules are linked to the *MAPE-K Control Architecture Meta-Model* (see Figure 7).

The OCL invariant `hasAtLeastOneMapeKComponent` (see Listing 3) defined for *Self-Adaptive Unit* specifies that a *Self-Adaptive Unit* must contain at least one *MAPE-K Component*. The invariant retrieves the MAPE-K components of the *Self-Adaptive Unit* (e.g., `self.monitor` in line 2), and checks if at least one of them is not null.

Listing 3: invariant hasAtLeastOneMapeKComponent

```
1  invariant hasAtLeastOneMapeKComponent:
2          self.monitor <> null or
3          self.analyze <> null or
4          self.plan <> null or
5          self.execute <> null or
6          self.knowledge <> null;
```

The other invariants of the *MAPE-K Control Architecture Meta-Model* concern the interactions. All of these require that the target and context of the interaction are not null. This property is stored as an attribute `targetAndContextNotNull` in the *Interaction* class (see Listing 4).

Listing 4: attribute targetAndContextNotNull

```
1  attribute targetAndContextNotNull : Boolean[1] {derived readonly transient volatile} {
2          initial: self.context <> null and self.target <> null;
3  }
```

The OCL invariant `isInterComponent` is linked to *InterComponentInteraction*. This invariant (see Listing 5) first checks if the *context* and *target* entities of the *Interaction* are not null (line 2), then checks if they are not of the same type (i.e., they belongs to different *MAPE-K Component*s) (line 3). The `oclType()` is an OCL function that returns the class to which a calling object belongs to.

Listing 5: invariant isInterComponent

```
1  invariant isInterComponent:
2          if (targetAndContextNotNull)
3                  then context->oclType() <> target->oclType()
4          else true
5          endif;
```

The OCL invariant `isIntraComponent` is linked to *IntraComponentInteraction*. This invariant (see Listing 6) first checks if the *context* and *target* entities of the *Interaction* are not null (line 2), then it checks if they are of the same type (i.e., they belongs to the same *MAPE-K Component*s) (line 3). Notice that in the OCL syntax the symbol "="stands for the equality operator (usually represented as "==").

Listing 6: invariant isIntraComponent

```
1  invariant isIntraComponent:
2          if(targetAndContextNotNull)
3                  then context->oclType() = target->oclType()
4          else true
5          endif;
```

The OCL invariant `targetIsKnowledge` relates to *ReadWriteInteraction*. This invariant (see Listing 7) checks if the *target* entity of the *ReadWriteInteraction* belongs to the *Knowledge* class. The question mark "`?`" is the safe object navigator operator, which was introduced in OCL 2.4 to avoid the navigation through a null object yielding an invalid value. The `oclIsKindOf()` is an OCL function that returns true if the calling object belongs to the class specified through the argument (it is equivalent to Java's `instanceOf`).

Listing 7: invariant targetIsKnowledge

```
1 invariant targetIsKnowledge:
2         self.target?.oclIsKindOf(Knowledge);
```

The OCL invariant `isInverse` relates to *CoordinationInteraction*. This invariant (see Listing 8) checks if the *inverse Coordination*'s target is equal to the calling *Coordination*'s context, if the *inverse*'s context is equal to the calling's target and finally if the inverse of the inverse *Coordination* is the calling.

Listing 8: invariant isInverse

```
1 invariant isInverse:
2         isInverse: inverse?.target = self.context and
3         inverse?.context = self.target and
4         self = self.inverse?.inverse;
```

CAME makes use of pattern-specific OCL rules that support the definition of MAPE-K control architectures blueprints.

To this end, the *ConcretePattern* entity extends the *MAPEKControlArchitecture* by defining the OCL rules of all the available patterns. *PatternType* is an `enum` containing all the patterns' names that are available in DECOR, i.e., *Master/Slave*, *Information Sharing*, *Coordinated Control*, *Hierarchical Control*, *Regional Planning*, a *CustomPattern* with no predefined rules and a *NoPattern* default invalid *PatternType*, which indicates that the user has not yet selected a pattern.

The OCL invariant `noPatternSelected` (see Listing 9, line 3) relating to *ConcretePattern* checks if the user has selected a valid *PatternType*.

Listing 9: ConcretePattern entity

```
1 class ConcretePattern extends MAPEKControlArchitecture {
2     attribute type : PatternType[1];
3     invariant noPatternSelected: if(type = PatternType::NoPattern) then false else true endif;
4     ...
5     ...
6 }
```

To develop a new pattern in DECOR, the developer has therefore two possibilities: (*i*) using the *CustomPattern PatternType*, which does not place any constraints on the MAPE-K control architecture and thus will not be validated through OCL, or, more formally, (*ii*) extending the CAME by adding a new *PatternType*, and defining the OCL rules that characterize the new pattern. In the latter (and suggested) alternative the newly defined pattern can be validated with OCL according to the developer's rules.

In the following we show the specification of the standard patterns available in the tool: *Master/Slave*, *Information Sharing*, *Coordinated Control*, *Hierarchical Control* and *Regional Planning*.

A.2. Master/Slave Pattern

The Master/Slave pattern is composed by the *SelfAdaptiveUnit Master* entity and one or more *Slaves*. Listing 10 shows the *Master* `class` with all the invariants that characterize it. The *Master* has no *Monitor* (line 2) and no *Execute* (line 5) *MapeKComponents*, while it has *Analyze* (line 3) and *Plan* (line 4). The invariant *analyzeSpeaksWithPlan* (line 6) validates if the *Analyze* and *Plan Master*'s *MapeKComponents* communicate through an *InterComponentInteraction*. First, it obtains all the *Master*'s *IntercomponentInteractions* and then it selects those with context of type *Analyze* (line 7) and target of type *Plan* (line 8), and

then the invariant checks if their cardinality is 1. The function `getMyContextInterComponentInteractions` (line 6) is defined for the *SelfAdaptiveUnit* class and it returns all the *InterComponentInteraction*s for which the caller instance is the context.

Listing 10: Master entity

```
1  class Master extends SelfAdaptiveUnit {
2          invariant hasNOMonitor: self.monitor = null;
3          invariant hasAnalyze: self.analyze <> null;
4          invariant hasPlan: self.plan <> null;
5          invariant hasNOExecute: self.execute = null;
6          invariant analyzeSpeaksWithPlan: getMyContextInterComponentInteractions
7                  ->select(context.oclIsKindOf(Analyze))
8                  ->select(target.oclIsKindOf(Plan))->size()=1;
9  }
```

Listing 11 represents the *Slave* `class` with all its invariants. The *Slave* has no *Analyze* (line 4) and no *Plan* (line 5) *MapeKComponents*, while it has *Monitor* (line 3) and *Execute* (line 6).

The invariants `checkONEManaged` (line 7), `actONEManaged` (line 8) and `checkAndAct` (line 9) verify that (*i*) the *Slave*'s *Monitor* is connected to one *LocalManagedSystem*, (*ii*) the *Slave*'s *Execute* acts on one *LocalManagedSystem* and, (*iii*) that they both manage the same *LocalManagedSystem*'s instance. The boolean auxiliary attributes `hasOneManagedCheck` (see Listing 12) and `hasOneManagedAct` are defined for the *Monitor* and *Execute MapeKComponent*s, respectively, and verify if the *MapeKComponent*s manage one instance only of type *LocalManagedSystem*.

Finally, the invariants `slaveMonitorSpeaksWithMasterAnalyze` (see Listing 11, line 9) and `masterPlanSpeaksWithSlaveExecute` (see Listing 11, line 11) verify that there is one *InterComponentInteraction* that connects the *Slave*'s *Monitor* with an *Analyze MapeKComponent* and one *InterComponentInteraction* that connects a *Plan MapeKComponent* with the *Slave*'s *Execute*, respectively. Notice that there is no need to specify that the *Analyze* and *Plan MapeKComponent*s belong to the *Master*, since this constraint is implicitly satisfied by the union of all the defined invariants. The function `getMyTargetInterComponentInteractions` (see Listing 11, line 11) is defined for the *SelfAdaptiveUnit* class and it returns all the *InterComponentInteraction*s for which the caller instance is the target.

Listing 11: Slave entity

```
1  class Slave extends SelfAdaptiveUnit {
2          invariant hasMonitor: self.monitor <> null;
3          invariant hasNOAnalyze: self.analyze = null;
4          invariant hasNOPlan: self.plan = null;
5          invariant hasExecute: self.execute <> null;
6          invariant checkONEManaged: self.monitor?.hasOneManagedCheck;
7          invariant actONEManaged: self.execute?.hasOneManagedAct;
8          invariant checkAndAct: self.monitor?.check = self.execute?.act;
9          invariant slaveMonitorSpeaksWithMasterAnalyze: getMyContextInterComponentInteractions
10                 ->select(context.oclIsKindOf(Monitor))->select(target.oclIsKindOf(Analyze))->size()=1;
11         invariant masterPlanSpeaksWithSlaveExecute: getMyTargetInterComponentInteractions
12                 ->select(context.oclIsKindOf(Plan))->select(target.oclIsKindOf(Execute))->size()=1;
13 }
```

Listing 12: attribute hasOneManagedCheck

```
1  attribute hasOneManagedCheck : Boolean[1] {derived readonly transient volatile} {
2          initial:
3          self.check?->size() = 1 and self.check->select(oclIsKindOf(LocalManagedSystem))->size() = 1;
4  }
```

The invariant `exactlyOneMaster` (see Listing 13, line 1) checks that there is only one *Subsystem* of type *Master* in the *ConcretePattern*, while the invariant `atLeastOneSlave` (see Listing 13, line 6) checks that at least one *Subsystem* in the *ConcretePattern* is of type *Slave*.

Listing 13: invariants exactlyOneMaster and atLeastOneSlave

```
1  invariant exactlyOneMaster:
2          if(type = PatternType::MasterSlave)
3                  then self.subsystems->select(oclIsKindOf(Master))->size() = 1
4          else true
5          endif;
6  invariant atLeastOneSlave:
7          if(type = PatternType::MasterSlave)
8                  then self.subsystems->select(oclIsKindOf(Slave))->size() >= 1
9          else true
10         endif;
```

The invariant `maxMasterSlavePatternInteractionAllowed` (see Listing 14, line 1) limits the number of *Interaction*s allowed, to be two times the number of the *Slave* instances, i.e., all *InterComponentInteraction*s connecting the *Slave*s' *Monitor* with the *Master*'s *Analyze* and the *Master*'s *Plan* with the *Slave*s' *Execute*, plus one *Interaction*, i.e., the *InterComponentInteraction* connecting the *Master*'s *Analyze* with the *Master*'s *Plan*. The invariant `presenceOfNotAllowedElementsInMasterSlavePattern` (see Listing 14, line 7) limits the *SelfAdaptiveUnit*s of the *ConcretePattern* to be only those of type *Master* and *Slave*. The invariant first collects all the *SelfAdaptiveUnit*s of the *ConcretePattern*, rejects those of type *Master* and *Slave* (line 10 and 11), and force the cardinality of this resulting set to be 0 (line 11).

Listing 14: invariants maxMasterSlavePatternInteractionAllowed and presenceOfNotAllowedElementsInMSPattern

```
1  invariant maxMasterSlavePatternInteractionAllowed:
2          if(type = PatternType::MasterSlave)
3              then self.interactions->select(oclIsKindOf(Interaction))
4                          -> size() <= 2 * (self.subsystems->select(oclIsKindOf(Slave))->size()) + 1
5          else true
6          endif;
7  invariant presenceOfNotAllowedElementsInMSPattern:
8          if(type = PatternType::MasterSlave)
9                  then self.subsystems -> select(oclIsKindOf(SelfAdaptiveUnit))
10                 -> reject(oclIsKindOf(Master))
11                 -> reject(oclIsKindOf(Slave))->size() = 0
12         else true
13         endif;
```

A.3. Information Sharing Pattern

This paragraph describes the OCL rules defined for the information sharing pattern, which is composed only by the *SelfAdaptiveUnit Peer* entities. *Peer* extends the abstract entity *AbstractPeer* (see Listing 15) that defines the constraints on the type of *MapeKComponent*s, i.e., *Monitor*, *Analyze*, *Plan* and *Execute* (line 2-5), the *InterComponentInteraction*s allowed, i.e., *Monitor-Analize*, *Analize-Plan*, *Plan-Execute* (line 6,9 and 12), and limits the *InterComponentInteraction*s to be the three just described (line 15). The *Peer* entity (see Listing 16) extends *AbstractPeer* by only adding the constraints `checkONEManaged`, `actONEManaged` and `checkAndAct` that force the *Peer*'s *Monitor* and *Execute* to manage one *LocalManagedSubsystem* and they are equivalent of those defined for the *Slave SelfAdaptiveUnit* in Listing 11.

Listing 15: AbstractPeer entity

```
1  abstract class AbstractPeer extends patternGenerator::SelfAdaptiveUnit {
2          invariant hasMonitor: self.monitor <> null;
3          invariant hasAnalyze: self.analyze <> null;
```

```
4           invariant hasPlan: self.plan <> null;
5           invariant hasExecute: self.execute <> null;
6           invariant monitorSpeaksWithAnalyze: getMyContextInterComponentInteractions
7                   ->select(context.oclIsKindOf(Monitor))
8                   ->select(target.oclIsKindOf(Analyze))->size() = 1;
9           invariant analyzeSpeaksWithPlan: getMyContextInterComponentInteractions
10                  ->select(context.oclIsKindOf(Analyze))
11                  ->select(target.oclIsKindOf(Plan))->size() = 1;
12          invariant planSpeaksWithExecute: getMyContextInterComponentInteractions
13                  ->select(context.oclIsKindOf(Plan))
14                  ->select(target.oclIsKindOf(Execute))->size() = 1;
15          invariant maxInterInteractionAllowed: getMyContextInterComponentInteractions -> size() <= 3;
16  }
```

Listing 16: Peer entity

```
1   class Peer extends AbstractPeer {
2           invariant checkONEManaged: self.monitor?.hasOneManagedCheck;
3           invariant actONEManaged: self.execute?.hasOneManagedAct;
4           invariant checkAndAct: self.monitor.check = self.execute.act;
5   }
```

The following OCL rules relate to *CustomPattern* in the case *InfoSharing* or *CoordControl* (which will be described later) is chosen as *PatternType*.

The invariant `atLeastOnePeer` (see Listing 17, line 1) validates that there is at least one *Subsystem* of type *Peer* in the *ConcretePattern*, while the invariant `presenceOfNotAllowedElements` (see Listing 17, line 6) validates that there are no other *Subsystem*s in the *ConcretePattern* except those of type *Peer*.

Listing 17: invariants atLeastOnePeer and presenceOfNotAllowedElements

```
1   invariant atLeastOnePeer:
2           if(type = PatternType::InfoSharing or type = PatternType::CoordControl)
3                   then self.subsystems->select(oclIsKindOf(Peer)) -> size() > 0
4           else true
5           endif;
6   invariant presenceOfNotAllowedElements:
7           if(type = PatternType::InfoSharing or type = PatternType::CoordControl)
8                   then self.subsystems -> select(oclIsKindOf(SelfAdaptiveUnit))
9                   - >reject(oclIsKindOf(Peer)) -> size() = 0
10          else true
11          endif;
```

The invariant `intraComponentInteractionIsMCoord` (see Listing 18), defined for *CustomPattern* in the case *InfoSharing* is chosen as *PatternType*, validates that the only *IntraComponentInteraction*s allowed are the *Coordination*s between *MapeKComponent*s of type *Monitor* (line 6 and 7).

Listing 18: invariant intraComponentInteractionIsMCoord

```
1   invariant intraComponentInteractionIsMCoord:
2           if(type = PatternType::InfoSharing)
3                   then self.interactions->select(oclIsKindOf(IntraComponentInteraction))
4                   ->select(oclAsType(Interaction).targetAndContextNotNull)
5                   -> reject(oclIsKindOf(Coordination) and
6                   oclAsType(Interaction).context?.oclIsKindOf(Monitor) and
7                   oclAsType(Interaction).target?.oclIsKindOf(Monitor)) -> size() = 0
8           else true
9           endif;
```

A.4. Coordinated Control Pattern

The Coordinated Control pattern is similar to the Information Sharing pattern, except for the fact that in the first the *Coordination*s can be among *Monitor*s, as well as *Analyze*s, *Plan*s and *Execute*s, while in the latter the *Coordination*s are only between *Monitor MapeKComponent*s. Their strong similarity implies that most of the OCL rules are the same for both. The Coordinated Control pattern does not have the `intraComponentInteractionIsMCoord` invariant. The invariant `intraComponentInteractionIsCoord` (see Listing 19) relates to *CustomPattern* in the case *CoordControl* is choosen as *PatternType* and it validates that the only *IntraComponentInteraction*s are the *Coordination*s and thus that no *Delegation*s are present.

Listing 19: invariant intraComponentInteractionIsCoord

```
1  invariant intraComponentInteractionIsCoord:
2          if(type = PatternType::CoordControl)
3                  then self.interactions->select(oclIsKindOf(IntraComponentInteraction))
4                  ->select(oclAsType(Interaction).targetAndContextNotNull)
5                  ->select(oclIsKindOf(Delegation))->size() = 0
6          else true
7          endif;
```

A.5. Hierarchical Control Pattern

The Hierarchical Control pattern is composed only by the *SelfAdaptiveUnit HPeer* entity (see Listing 20), which extends *AbstractPeer* by just specifying with `hasOneCheckAndAct` invariant (line 2) that the *Subsystem*s checked by the *Monitor MapeKComponent* and those acted by the *Execute* are at least one and they both are the same instances. Notice that differently for the *Peer* entity (see Listing 16) in this case we are not constraining the type of managed entity to be *LocalManagedSubsystem*, since in the hierarchical control pattern *SelfAdaptiveUnit*s can manage other *SelfAdaptiveUnit*s, as well as *LocalManagedSubsystem*s.

Listing 20: HPeer entity

```
1  class HPeer extends AbstractPeer {
2          invariant hasOneCheckAndAct:
3                  self.monitor?.check->size() > 0 and
4                  self.execute?.act->size() > 0 and
5                  self.monitor?.check = self.execute?.act;
6  }
```

The following OCL rules relate to *CustomPattern* in the case *HierchicalControl* is choosen as *PatternType*.

The invariant `atLeastOneHPeer` (see Listing 21) validates that there is at least one *Subsystem* in the *ConcretePattern* of type *HPeer*. The invariant `presenceOfNotAllowedElementsInHCPattern` (see Listing 22, line 1) validates that there are no other *Subsystem*s in the *ConcretePattern* except those of type *HPeer* (line 3). Finally, the invariant `noIntraComponentInteraction` (see Listing 22, line 7) validates that there are no *IntraComponentInteraction*s.

Listing 21: invariant atLeastOneHPeer

```
1  invariant atLeastOneHPeer:
2          if(type = PatternType::HierchicalControl)
3                  then self.subsystems->select(oclIsKindOf(HPeer))->size() >= 1
4          else true
5          endif;
```

Listing 22: invariants presenceOfNotAllowedElementsInHCPattern and noIntraComponentInteraction

```
1  invariant presenceOfNotAllowedElementsInHCPattern:
2          if(type = PatternType::HierchicalControl)
3                  then self.subsystems->select(oclIsKindOf(SelfAdaptiveUnit))
```

```
4                   -> reject(oclIsKindOf(HPeer)) -> size() = 0
5           else true
6           endif;
7   invariant noIntraComponentInteraction:
8           if(type = PatternType::HierchicalControl)
9                   then self.interactions -> select(oclIsKindOf(IntraComponentInteraction)) -> size() = 0
10          else true
11          endif;
```

A.6. Regional Planning Pattern

The Regional Planning pattern is composed by one or more *SelfAdaptiveUnit RegionalPlanner* entities and one or more *UnderlyingSubsystem*s. Listing 23 represents the *RegionalPlanner* class with all its invariants. The *RegionalPlanner* has no *Monitor* (line 2), no *Analyze* (line 3) and no *Execute* (line 5) *MapeKComponent*s, while it has the *Plan MapeKComponent* (line 4).

Listing 23: RegionalPlanner entity

```
1   class RegionalPlanner extends patternGenerator::SelfAdaptiveUnit {
2           invariant hasNoMonitor: self.monitor = null;
3           invariant hasNoAnalyze: self.analyze = null;
4           invariant hasPlan: self.plan <> null;
5           invariant hasNoExecute: self.execute = null;
6   }
```

Listing 24 represents the *UnderlyingSubsystem* entity with all its invariants. The *UnderlyingSubsystem* has *Monitor* (line 2), *Analyze* (line 3) and *Execute* (line 5) *MapeKComponent*s, while it has no *Plan MapeKComponent* (line 4). The invariants `checkONEManaged` (line 6), `actONEManaged` (line 7) and `checkAndAct` (line 8) force the *UnderlyingSubsystem*'s *Monitor* and *Execute* to manage exactly one *LocalManagedSubsystem*.

The invariants `monitorSpeaksWithAnalyze`, `analyzeSpeaksWithPlan`, and `planSpeaksWithExecute` define the constraints on the *InterComponentInteraction*s allowed (lines 9,12 and 14, respectively), i.e., *Monitor-Analyze*, *Analize-Plan*, *Plan-Execute*. The invariant `maxInterInteractionAllowed` limits the *InterComponentInteraction*s to be the three just described.

Finally, the invariant `hasOneOnlyAssociatedPlanner` (line 18) binds the *Plan MapeKComponent* of the *Analyze-Plan InterComponentInteraction* to belong to the same *RegionalPlanner* instance of the one of the *Plan-Execute* interaction.

Listing 24: UnderlyingSubsystem entity

```
1   class UnderlyingSubsystem extends patternGenerator::SelfAdaptiveUnit {
2           invariant hasMonitor: self.monitor <> null;
3           invariant hasAnalyze: self.analyze <> null;
4           invariant hasNoPlan: self.plan = null;
5           invariant hasExecute: self.execute <> null;
6           invariant checkONEManaged: self.monitor.hasOneManagedCheck;
7           invariant actONEManaged: self.execute.hasOneManagedAct;
8           invariant checkAndAct: self.monitor.check = self.execute.act;
9           invariant monitorSpeaksWithAnalyze: getMyContextInterComponentInteractions
10                  -> select(context.oclIsKindOf(Monitor)) -> select(target.oclIsKindOf(Analyze))
11                  -> size() = 1;
12          invariant analyzeSpeaksWithPlan: getMyContextInterComponentInteractions
13                  -> select(context.oclIsKindOf(Analyze)) -> select(target.oclIsKindOf(Plan)) -> size() = 1;
14          invariant planSpeaksWithExecute: getMyTargetInterComponentInteractions
15                  -> select(context.oclIsKindOf(Plan)) -> select(target.oclIsKindOf(Execute)) -> size() = 1;
16          invariant maxInterInteractionAllowed: getMyContextInterComponentInteractions->size() <= 2
17                  and getMyTargetInterComponentInteractions->size()<=2;
18          invariant hasOneOnlyAssociatedPlanner:
```

```
19                  if(hasAssociatedContextPlanner and hasAssociatedTargetPlanner)
20                          then self.getMyContextInterComponentInteractions
21                          -> any(target.oclIsKindOf(Plan)).target.oclContainer().oclAsType(RegionalPlanner)
22                              = self.getMyTargetInterComponentInteractions
23                          -> any(context.oclIsKindOf(Plan)).context.oclContainer()
24                              .oclAsType(RegionalPlanner)
25                  else true
26                  endif;
27  }
```

The OCL rules in Listing 25 relate to *CustomPattern* in the case *RegionalPlanning* is chosen as *PatternType*. The rules `atLeastOneUnderlyingSubsystem` (line 1) and `atLeastOneRegionalPlanner` (line 6) specify that in the *CustomPattern* subsystems there must be at least one *UnderlyingSubsystem* and at least one *RegionalPlanner* entities. The invariant `presenceOfNotAllowedElementsInRegionalPlanningPattern` (line 11) limits the *Subsystem*s of the *CustomPattern* to be only those of type *RegionalPlanner* or *UnderlyingSubsystem*. This invariant first selects all the *SelfAdaptiveUnit*s instances of the *CustomPattern*'s subsystems (line 13), then it rejects all *RegionalPlanner*s and *UnderlyingSubsystem*s, and force the cardinality of this resulting set to be equal to 0 (line 15). Finally, the Invariant `intraComponentInteractionIsPCoord` (line 18) limits the *IntraComponentInteraction*s of the *CustomPattern* to be only the *Coordination*s between the *Plan MapeKComponent*s of the *RegionalPlanner*s.

Listing 25: Regional Planning Pattern invariants

```
1   invariant atLeastOneUnderlyingSubsystem:
2           if(type = PatternType::RegionalPlanning)
3                   then self.subsystems->select(oclIsKindOf(UnderlyingSubsystem)) -> size() > 0
4           else true
5           endif;
6   invariant atLeastOneRegionalPlanner:
7           if(type = PatternType::RegionalPlanning)
8                   then self.subsystems->select(oclIsKindOf(UnderlyingSubsystem)) -> size() > 0
9           else true
10          endif;
11  invariant presenceOfNotAllowedElementsInRegionalPlanningPattern:
12          if(type = PatternType::RegionalPlanning)
13                  then self.subsystems->select(oclIsKindOf(SelfAdaptiveUnit))
14                  -> reject(oclIsKindOf(RegionalPlanner))
15                  -> reject(oclIsKindOf(UnderlyingSubsystem)) -> size() = 0
16          else true
17          endif;
18  invariant intraComponentInteractionIsPCoord:
19          if(type = PatternType::RegionalPlanning)
20                  then self.interactions->select(oclIsKindOf(IntraComponentInteraction))
21                  -> select(oclAsType(Interaction).targetAndContextNotNull)
22                  -> reject(oclIsKindOf(Coordination) and
23                      oclAsType(Interaction).context.oclIsKindOf(Plan) and
24                      oclAsType(Interaction).target.oclIsKindOf(Plan)) -> size() = 0
25          else true
26          endif;
```