OCL Constraints Laboratory Report

Project Overview: Untangle Task Management System

This report documents the development of a UML model with OCL constraints for a task management system called "Untangle". The system allows users to manage projects, tasks, and organize work through sprints, providing features to track project history and task status.

Model Structure

The model consists of 5 interrelated classes:

1. **Project** - Represents a project with tasks and sprints
2. **Task** - Represents individual work items with priorities and statuses
3. **User** - Represents system users who own tasks and manage projects
4. **Date** - Utility class for date handling and calculations
5. **Note** - Represents text notes with creation and update timestamps

Additionally, the model includes two enumeration types:

- **Priority** (Low, Medium, High) - For task prioritization
- **Status** (Open, InProgress, Done) - For tracking task progress

Class Relationships

The model defines three key associations:

- **Owns** - Users own tasks (User[*] — Task[1])
- **Manages** - Users manage projects (User[*] — Project[1])
- **Contains** - Projects contain tasks (Project[0..1] — Task[*])

Complex Data Structures

The model implements advanced data structures including:

- **Collection of collections** in Project's

```
sprints : Sequence(Set(Task))
```

- **Tuples** in Project's

```
history : Set(Tuple(date:Integer, action:String))
```

OCL Constraints Implementation

Invariants

The model includes 17 invariants across classes. Here are key examples with code snippets:

Project Invariants (10)

1. **uniqueId** - Ensures all projects have unique identifiers

```
inv uniqueId:
  -- Ensures that all projects have unique identifiers to prevent
duplicates
  Project.allInstances()->forAll(p1, p2 | p1 <> p2 implies p1.id <>
p2.id)
```

2. **validTitle** - Ensures projects have non-empty titles

```
inv validTitle:
  -- Ensures that project title is not empty, as titles are required
for identification
  self.title.size() > 0
```

3. **validDateRange** - Ensures project end date is after start date

```
inv validDateRange:
  -- Ensures that project has a valid timeframe (end date must be
after start date)
  self.endDate > self.startDate
```

4. **chronologicalHistory** - Ensures project history events are chronologically ordered

```
inv chronologicalHistory:
  -- Ensures that project history events are ordered chronologically
by date
  self.history->asSequence() = self.history->asOrderedSet()-
>sortedBy(h | h.date)
```

5. **limitHighPriorityTasks** - Limits high priority tasks to 5 per project

```
inv limitHighPriorityTasks:
  -- Limits the number of high priority tasks to 5 to maintain focus
and prevent priority inflation
  self.task->select(t | t.priority = Priority::High)->size() <= 5
```

6. **uniqueTasksInSprints** - Ensures tasks appear in at most one sprint

```
inv uniqueTasksInSprints:
  -- Ensures that each task can only be in one sprint at a time
  self.sprints->forAll(s1, s2 |
    s1 <> s2 implies s1->intersection(s2)->isEmpty()
  )
```

Task Invariants (6)

1. **uniqueID** - Ensures system-wide unique task IDs

```
inv uniqueID:
  -- Ensures that all tasks in the system have unique identifiers
  Task.allInstances()->forAll(t1, t2 | t1 <> t2 implies t1.id <>
t2.id)
```

2. **importantTasksHaveDueDate** - Ensures high priority tasks have deadlines

```
inv importantTasksHaveDueDate:
  -- Ensures that high priority tasks always have a specified deadline
  self.priority = Priority::High implies not
self.dueDate.oclIsUndefined()
```

3. **highPriorityTasksHaveProjectAssigned** - Ensures high priority tasks are assigned to projects

```
inv highPriorityTasksHaveProjectAssigned:
  -- Ensures high priority tasks are always assigned to a project for
accountability
  self.priority = Priority::High implies not
self.project.oclIsUndefined()
```

User Invariants (2)

1. **idRequired** - Ensures users have non-empty identifiers

```
inv idRequired:
  -- Ensures that all users have a non-empty identifier
  not self.id.oclIsUndefined() and self.id <> ''
```

2. **emailRequired** - Ensures users have valid email addresses

```
```

```
  inv emailRequired:
    -- Ensures that all users have a valid email address for
  communication
    not self.email.oclIsUndefined() and self.email <> ''
```

Pre-conditions and Post-conditions

The model implements 17 pre-conditions and 20 post-conditions across various operations, ensuring operations maintain the system's integrity. Here are key examples with code snippets:

Project Operations

addTask - Adds a task to a project with validation

```
context Project::addTask(task : Task)
  pre taskExists:
    -- Ensures the task parameter is not null before adding to project
    not task.oclIsUndefined()

  pre taskNotAlreadyAdded:
    -- Prevents adding duplicate tasks to the project
    not self.task->includes(task)

  post taskAdded:
    -- Verifies that the task has been successfully added to the project
    self.task->includes(task)

  post taskSizeIncrease:
    -- Ensures that the number of tasks has increased by exactly one
    self.task->size() = self.task@pre->size() + 1
```

createNewSprint - Creates a new empty sprint

```
context Project::createNewSprint() : Integer
  post sprintCreated:
    -- Verifies that a new sprint has been added to the sequence of
sprints
    self.sprints->size() = self.sprints@pre->size() + 1

  post newSprintIsEmpty:
    -- Ensures that the newly created sprint starts empty without tasks
    self.sprints->at(self.sprints->size())->isEmpty()

  post returnsNewSprintIndex:
    -- Ensures the operation returns the index of the newly created sprint
    result = self.sprints->size()
```

Task Operations

changeStatus - Updates task status

```
context Task::changeStatus(status : Status)
  post statusChanged:
    -- Confirms that the status has been updated to the new value
    self.status = status

  post oldStatusDifferent:
    -- Ensures that the status has actually changed (not just set to same
value)
    self.status <> self.status@pre
```

isOverdue - Checks if a task is past its due date

```
context Task::isOverdue() : Boolean
  pre notAlreadyDone:
    -- Only checks overdue status for tasks that are not already completed
    self.status <> Status::Done

  pre hasDueDate:
    -- Ensures that the task has a due date that can be checked against
    not self.dueDate.oclIsUndefined()

  post dueDateInThePast:
    -- Determines if the task is overdue by comparing its due date with
current date
    let currentDate : Date = Date.allInstances()->any(d | d.now = 20199),
        taskDate : Integer = self.dueDate
    in
      result = currentDate.now > taskDate
```

Special OCL Features

The model implements several advanced OCL features with example code snippets:

Collection Operations

At least 9 different collection operations are used throughout the constraints:

select - Used to filter collections

```
-- Example: Getting high priority tasks
highPriorityTasks() : Set(Task) =
  self.task->select(t | t.priority = Priority::High)
```

collect - Used to transform collections

```
-- Example: Extracting all task titles
getAllTaskTitles() : Set(String) =
  self.task->collect(t | t.title)
```

forAll - Used for universal quantification

```
-- Example: Ensuring unique IDs
Project.allInstances()->forAll(p1, p2 | p1 <> p2 implies p1.id <> p2.id)
```

intersection & isEmpty - Used to find common elements and check emptiness

```
-- Example: Ensuring tasks appear in at most one sprint
self.sprints->forAll(s1, s2 |
  s1 <> s2 implies s1->intersection(s2)->isEmpty()
)
```

sortedBy - Used for ordering collections

```
-- Example: Ensuring chronological history
self.history->asSequence() = self.history->asOrderedSet()->sortedBy(h |
h.date)
```

@pre Usage in Post-conditions

Multiple post-conditions use the @pre operator to reference attribute values at operation start:

```
-- Example: Tracking collection size changes
post taskSizeIncrease:
  self.task->size() = self.task@pre->size() + 1

-- Example: Comparing old and new values
post oldStatusDifferent:
  self.status <> self.status@pre

-- Example: Ensuring only specific parts are modified
post otherSprintsUnchanged:
  Sequence{1..self.sprints->size()}->excluding(sprintIndex)->forAll(i |
    self.sprints->at(i) = self.sprints@pre->at(i)
  )
```

Queries

The model defines several query operations:

```
-- Example: Query using select
highPriorityTasks() : Set(Task) =
  self.task->select(t | t.priority = Priority::High)

-- Example: Query using collect
getAllTaskTitles() : Set(String) =
  self.task->collect(t | t.title)
```

Local Variables

Local variables are used in several operations to improve readability:

```
-- Example: Local variables in date calculations
day() : Integer =
  let dayOfYear : Integer = self.now - ((self.year() - 1970) * 365),
      monthVal : Integer = self.month(),
      daysBeforeMonth : Integer = ((monthVal - 1) * 365) div 12
  in
    dayOfYear - daysBeforeMonth

-- Example: Local variables in overdue checking
post dueDateInThePast:
  let currentDate : Date = Date.allInstances()->any(d | d.now = 20199),
      taskDate : Integer = self.dueDate
  in
    result = currentDate.now > taskDate
```

Collection of Collections

```
-- Example: Sequence of Sets in Project class
sprints : Sequence(Set(Task))
```

Tuples

```
-- Example: Tuple type in attributes
history : Set(Tuple(date:Integer, action:String))

-- Example: Tuple literals in operations
self.history->includes(Tuple{date = 20222, action = 'Create'})
```

Conclusion

The Untangle task management system demonstrates the power of OCL constraints in maintaining data integrity and enforcing business rules in a complex software system. The constraints ensure that projects and tasks maintain proper relationships, have required attributes, and that operations maintain the system's consistency.

The model showcases various OCL features including collections, tuples, queries, and the @pre operator, demonstrating how these can be used effectively to model real-world business requirements and constraints.