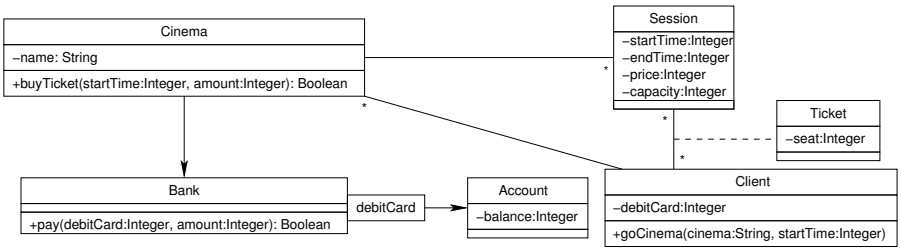

Figure 1: Class diagram for the cinema example.

A simple UML model for a Ticket Sale System

We will show how to use the mOdCL evaluator by means of a simplified model for a *Tickets Sale System* which includes its class diagram (with invariants and pre- and post- conditions) and some object diagrams.

A `Cinema` has a `name` and offers a number of sessions, for which it sells tickets, managing the payment through a single `bank`. A `Session` has a `capacity`, a ticket `price`, start/finish times (`startTime`/`endTime`), and the tickets sold to clients. A `Client` knows some cinemas, pays with a `debitCard` and has access to the tickets he has bought. The association class `Ticket` represents a ticket bought by a client for a given session; each ticket has a `seat` number. A `Bank` knows the accounts that are associated with the debit cards, modeled as a qualified association with the `debitCard` number as key. Each `Account` has a `balance`.

The class diagram includes two invariants: the invariant `avoid-overlapping` specifies that a client cannot buy two tickets for overlapping sessions.

```
context Cliente inv avoid-overlapping :
  tickets -> forAll(T1 | tickets
            -> forAll(T2 | (T1 = T2)
                 or (T1.session.endTime < T2.session.startTime)
                 or (T2.session.endTime < T1.session.startTime))))
```

an the invariant `seats-in-session` states that the number of tickets sold for a session does not exceed its capacity.

```
context Session inv seats-in-session :
    capacity >= tickets -> size()
```

The class diagram includes three public operations: `goCinema`, in the `Client` class, allows a client to purchase a ticket for a given cinema, for a given time; `buyTicket`, in the `Cinema` class, allows to buy one ticket for the session starting at a given time; and `pay`, in the `Bank` class, charges a given `amount` to the account associated to a given `debitCard`. The behavior of the operations is not fully specified. It could be done, for example, by using sequence diagrams as we show for the `buyTicket` operation in Figure 2

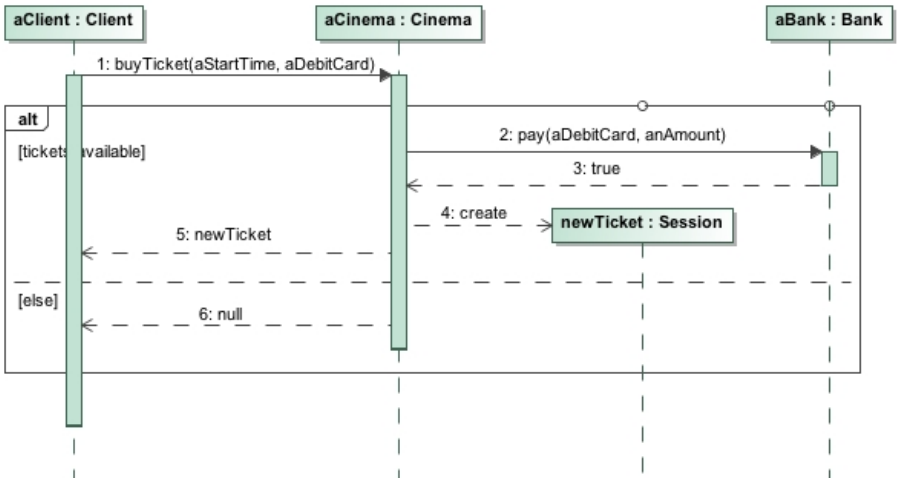

Figure 2: Class diagram for the cinema example.

Finally, we will illustrate the use of pre- and post- conditions of operations by using the `buyTicket` operation. We impose as a precondition that the cinema must offer a session at the time requested, and as a postcondition that the returned value should be `null` or a new ticket, which in turn should be the only ticket added to the tickets list of the requested session.

```
context Cinema::buyTicket(startTime:Integer,aClient:Client):Ticket
pre : sessions -> select(S | S.startTime = startTime)->size() = 1
post: (result = null)
       or
      -- tickets of the session must include the result ticket
      (sessions -> select(S | S.startTime = startTime).tickets
          -> includes(result)
       and
      -- the number of tickets increases in 1 unit
      ((sessions -> select(S |
               S.startTime = startTime).tickets->asSet())
        - (sessions -> select(S |
               S.startTime = startTime).tickets@pre->asSet()))
        ->size() = 1)
```

1 Object diagrams

The object diagram represents a snapshot of the system state at a given time, and is described with similar elements to those used for class diagrams, but now the boxes refer to concrete instances of the classes which contains specific values for their attributes and associations. An object diagram can be used, for

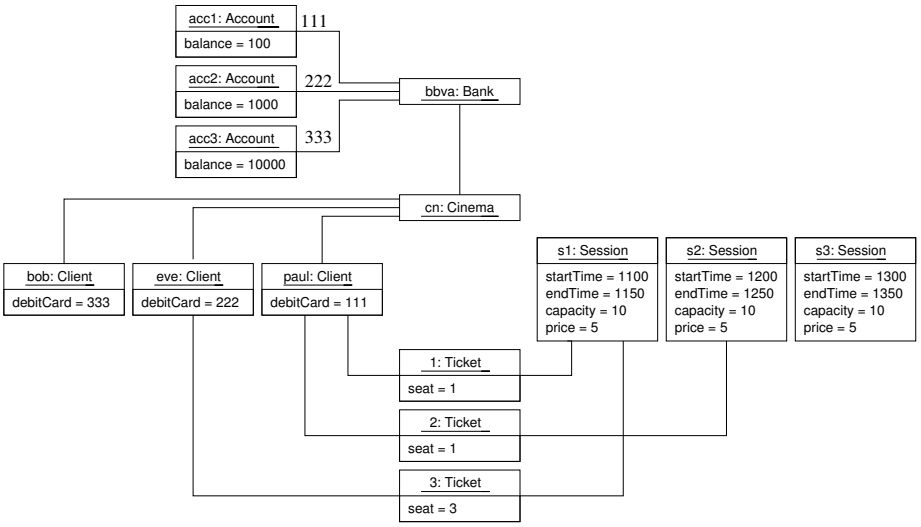

Figure 3: An objets diagram for the cinema example.

example, to describe situations which must fulfill given constraints, expressed as OCL expressions.

For example, Figure 3 shows an object diagram of the cinema model, which will be used to illustrate the use of mOdCL. In this diagram a cinema (`cn`) offers three sessions (`s1`, `s2` and `s3`), three active clientes (`eve`, `bob` and `paul`): `eve` bought one ticket for the session `s1` and `paul` bought two tickets for the sessions `s1` and `s2`.