Chapter 7

Modeling Relationships

Overview of Mappings

- A *mapping* is a data structure describing a relationship between the values of one set (called the *domain* of the mapping) and the values of another set (called the *range* of the mapping).
- Each domain element maps to the corresponding range element.
- Each domain element can occur only once in a mapping.

Overview of Mappings (cont)

- Mapping values are represented using curly brackets ("{ " and "} ") as delimiters, with individual values (called *maplets*) separated by commas.
- A maplet consists of two values (a *domain* and a *range* value) separated by the map symbol ("|->").
- For instance, the value

  `{"Paul" |-> 7, "Peter" |-> 5, "John" |-> 8, "Nico" |-> 8}`

  is a mapping consisting of four maplets: `{"John" |-> 8}`, `{"Nico" |-> 8}`, `{"Paul" |-> 7}`, and `{"Peter" |-> 5}`.
- The type of this value is, therefore, *map from strings to natural numbers*, in VDM++ written as:

  ```
  map (seq of char) to nat
  ```

Overview of Mappings (cont)

- In general, a mapping type in VDM++ can be written as

  ```
  map <type1> to <type2>
  ```

  by which a mapping from elements of `<type1>` to elements of `<type2>` is meant.
- `<type1>` is the *domain* type of the mapping and `<type2>` is the *range* type of the mapping.

Overview of Mappings (cont)

- An *injective* mapping from elements of `<type1>` to elements of `<type2>` is written as:

```
inmap <type1> to <type2>
```

- The difference is that with a 'normal' mapping all elements in the domain of the mapping are unique, but not necessarily the elements in its range. An injective mapping also has unique elements in its range.
- For example

```
{ "John" |-> 8, "Nico" |-> 8}
```

  is a valid mapping but not a valid injective mapping, and

```
{ "Peter" |-> 7, "Peter" |-> 5}
```

  is not a valid mapping because there are two different maplets with the same domain value.

Overview of Mappings (cont)

- The *empty mapping*, i.e., the mapping which has no maplets, is written as:

```
{ |-> }
```

Overview of Mappings (cont)

- A mapping comprehension has the general form:
  {<maplet> | <bindings> & <predicate>}
- Such an expression builds up a mapping as follows:
  1. Values are taken from <bindings>.
  2. These values are evaluated against an (optional) <predicate>.
  3. If the predicate yields true then the value is used to form <maplet>, and that maplet is added to the mapping.
- For example, the result of the mapping comprehension
  {i |-> i * i | i : nat1 & i <= 4}
  would be:
  { 1 |-> 1, 2 |-> 4, 3 |-> 9, 4 |-> 16}

Operator	Name	Type
dom m	Domain	(map A to B) → set of A
rng m	Range	(map A to B) → set of B
m1 munion m2	Merge	(map A to B) * (map A to B) → map A to B
m1 ++ m2	Override	(map A to B) * (map A to B) → map A to B
merge ms	Distributed merge	set of (map A to B) → map A to B
s <: m	Domain restrict to	(set of A) * (map A to B) → map A to B
s <-: m	Domain restrict by	(set of A) * (map A to B) → map A to B
m :> s	Range restrict to	(map A to B) * (set of B) → map A to B
m :-> s	Range restrict by	(map A to B) * (set of B) → map A to B
m(d)	Map apply	(map A to B) * A → B
m1 comp m2	Map composition	(map B to C) * (map A to B) → map A to C
m ** n	Map iteration	(map A to A) * nat → map A to A
m1 = m2	Equality	(map A to B) * (map A to B) → bool
m1 <> m2	Inequality	(map A to B) * (map A to B) → bool
inverse m	Map inverse	inmap A to B → inmap B to A

Table 7.1: Summary of VDM++ mapping operators.

Overview of Mappings (cont)

- The mapping application operator takes a mapping m and a domain value d as its argument, where d must be in the domain of the mapping and returns the range value of the maplet with the corresponding domain value.
- Mapping application is written in the same way as function and sequence application.
- For example,

```
{<red>|->1,<green>|->4,<blue>|->9} (<blue>)
```

  yields the value 9.
- Suppose mapping m has the value

```
{"Paul" |-> 7, "Peter" |-> 5, "John" |-> 8,
 "Nico" |-> 8}
```

  then

```
m ("Nico")
```

  yields the value 8.

Congestion Warning System Revisited

- The road network model was quite simplistic. Adding a new congestion monitor, for example, required inserting a new element in the sequence modeling the road network. Therefore, each time this was done, the 'location' (modeled by the index of the sequence) of the elements in the sequence after the newly inserted one changed.
- There was no notion of *lanes* on a motorway. Multiple sensors can be used at a given location, each gathering the data from vehicles passing over different lanes on the motorway. The congestion monitor combines the data from the different sensors at a location and decides whether congestions exists at that location or not.
- Having a single actuator manager is not realistic when implementing the system in a somewhat larger area. In that case a collection of actuator managers is used, each controlling a specific number of locations. At the borders of these areas it may be the case that actuator managers need to communicate about the signals being shown in a different area, or by sending a signal to another actuator manager, to be shown in the area controlled by that actuator manager. In order to keep track of which actuator manager controls which area, a *name server* is used.
- Congestion monitors, corresponding sensors and actuators were added to the system in one go. In practice the infrastructure for the system (sensors and actuators) are added first, then actuator managers and congestion monitors are added.

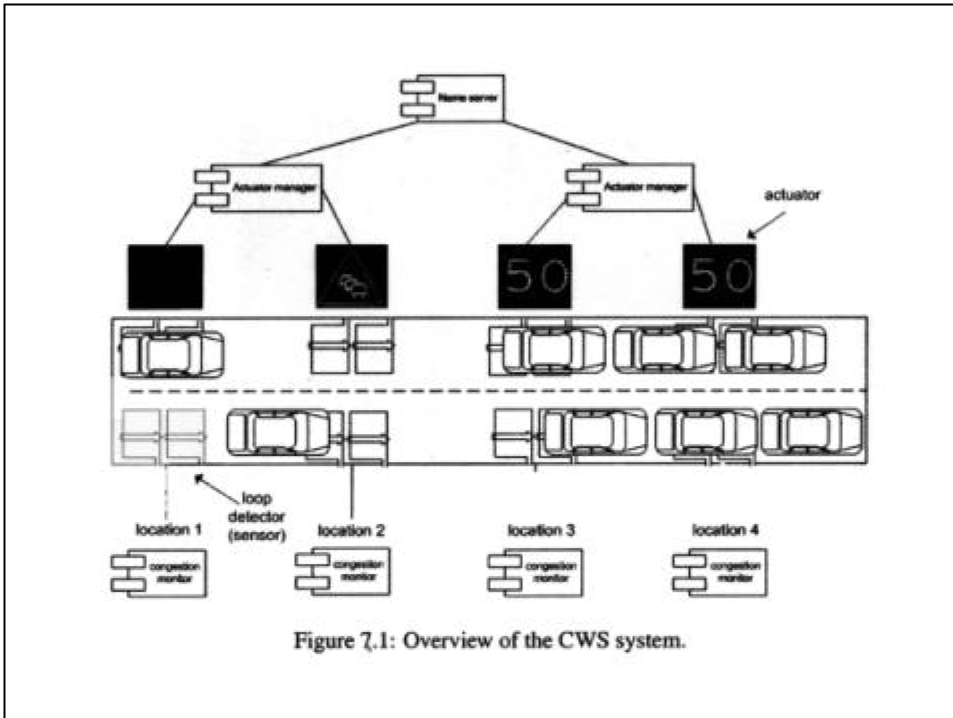

Figure 7.1: Overview of the CWS system.

```
class CWS
. . .
instance variables
  roadNetwork :
    inmap Location to CongestionMonitor
      := { |-> }
end CWS

class CWS
. . .
instance variables
  sensors :
    inmap Location to
      (inmap Lane to PassageSensor)
        := { |-> };
end CWS
```

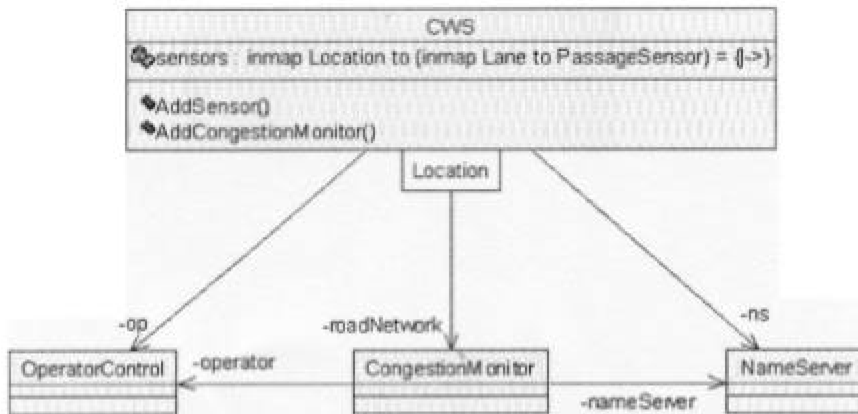

Figure 7.2: UML class diagram for CWS.

VDM++ Domain Operator

- The VDM++ *domain operator* takes a mapping as its argument and yields the *set of values* which are in its domain.
- It is generally written as
  `dom <map expression>`
- For example, suppose that mapping `m` has the value

```
{"Breakfast" |-> "Banana",
 "Lunch" |-> "Sandwich",
 "Dinner" |-> "Steak",
 "Snack" |-> "Sandwich"}
```

  then
  `dom m`
  will yield the set
  `{"Breakfast", "Lunch", "Dinner", "Snack"}`

VDM++ Range Operator

- The VDM++ *range operator* returns the set of values in its range of the mapping. It is written as
  `rng <map expression>`
- Given the same definition of mapping `m` as above,

```
{"Breakfast" |-> "Banana",
 "Lunch" |-> "Sandwich",
 "Dinner" |-> "Steak",
 "Snack" |-> "Sandwich"}
```

the result of
  `rng m`
will be the set
`{"Banana", "Sandwich", "Steak"}`

CWS Example (cont)

- To test whether or not the mapping `sensors` already uses a location `loc`, a simple set membership test can be used on the domain of *`sensors`*:

```
 if loc in set dom sensors
    . . .
```

- If the location is in the domain of `sensors`, then the range element of `sensors` at `loc` needs to be updated.
- A *state designator* can be used for this purpose at the left hand side of an assignment:
  `sensors (loc) := . . .`
- Then the mapping at `sensors (loc)` needs to be updated with the information for the sensor which is added.

CWS Example (cont)

- The VDM++ *mapping merge operator* takes two mappings with the same domain and range types as its arguments and returns a *single* mapping consisting of all the maplets in both arguments.
- The operator is written as an infix operator
  ```
  <map1> munion <map2>
  ```
- If `<map1>` and `<map2>` contain maplets with the same domain argument then it is still possible to apply the merge operator to them, providing that these maplets also have the same range elements. If not, then the resulting value is `undefined`.

CWS Example (cont)

For example

```
{1 |-> “Apple”, 3 |-> “Banana”} munion
  {2 |-> “Strawberry”, 3 |-> “Banana”}
```

yields

```
{1 |-> “Apple”, 2 |-> “Strawberry”,
 3 |-> “Banana”}
```

but

```
{1 |-> “Apple”, 3 |-> “Banana”} munion
   {2 |-> “Strawberry”, 3 |-> “Cherry”}
```

will yield

```
 undefined
```

because `3` maps to two different values: `“Banana”` and `“Cherry”`.

CWS Example (cont)

- The mapping merge operator is now applied as follows.
- The left hand side of the operator is a *mapping application* expression, representing the current value of `sensors (loc)`.
- The right hand side consists of a *mapping enumeration,* consisting of one element: the new value which binds the given lane to newly defined passage sensor:

```
sensors (loc) := sensors (loc) munion
  {lane |-> passageSensor}
```

- If the location is not in the domain of `sensors` the entire `sensors` mapping needs to be updated as follows:

```
sensors (loc) := sensors munion
  {loc |-> {lane |-> passageSensor}}
```

CWS Example (cont)

```
class CWS
. . .
operations

public AddSensor :  Location * Lane ==> ()
AddSensor (loc, lane) ==
  def passageSensor = new PassageSensor (loc, lane)
  in
    let sensorAtLane = {lane  |-> passageSensor}
    in
      if loc in set dom sensors
      then
        sensors (loc) := sensors (loc) munion sensorAtLane
      else
        sensors := sensors munion {loc |-> sensorAtLane};
end CWS
```

CWS Example (cont)

```
class NameServer
. . .
instance variables
  am : map ActuatorManager to (set of CWS`Location):={|->}
end NameServer
```

CWS Example (cont)

- The *mapping override* operator takes two mappings `<map1>` and `<map2>` as its arguments and yields a mapping whose maplets are all those maplets which are either maplets of `<map2>`, or all those maplets of `<map1>` whose domain element is not in the domain of `<map2>`.
- This is written as

```
<map1> ++ <map2>
```

- For example, the expression

```
{"Peter"|-><vegetarian>, "Nico"|-><meat>}++
   {"Peter"|-><meat>, "John"|-><vegetarian>}
```

yields

```
{"Peter" |-> <meat>,
 "Nico"  |-> <meat>,
 "John"  |-> <vegetarian>}
```

```
class NameServer
. . .
operations

public SetActuatorManager :
  ActuatorManager * set of CWS`Location ==> ()
SetActuatorManager (actuatorManager, locations) ==
  am := am ++ {actuatorManager |-> locations};
end NameServer
```

CWS Example (cont)

- The *inverse* operator takes an injective mapping and yields a mapping in which all domain values have been replaced by the corresponding range values and vice versa.
- The mapping inverse operator is written as
  `inverse <map>`
- For example
  ```
  inverse
     {"Peter"|-><vegetarian>, "Nico"|-><meat>}
  ```
  yields
  `{<vegetarian>|->"Peter", <meat>|->"Nico"}`

```
class NameServer
. . .
public
  GetActuatorManager:[CWS`Location]==>[ActuatorManager]
  GetActuatorManager (loc) ==
    if loc = nil
    then return nil
    else let locations = inverse am
         in
           let locationSet in set dom locations be st
               loc in set locationSet
           in
             return locations (locationSet);
end NameServer
```