Modelling using sets

- Sets:
    - The finite set type constructor
    - Value definitions: enumeration, subrange, comprehension
    - Operators on sets
- Case Study: the explosives store

To define a type:

- *a type constructor*
- *ways of writing down values*
- *ways of operating on values*

The idea of a set ...

An unordered collection of values:

The order doesn’t matter:

Nor do duplicates:

Would you represent each of the following as a set? Why?

The departures display board at an airport, showing outgoing flights in order of departure.

A database keeping track of who is in a restricted area at any given time.

The set type constructor

The finite set type constructor is: `set of _`

What are the types of the following expressions?

```
{1, -3, 12}

{ {9, 13, 77}, {32, 8}, {}, {77} }
```

The set type constructor

The type `set of X` is the class of all possible finite sets of values drawn from the type `X`. For example:

`set of nat1`	sets of non-zero Natural numbers
`set of Student`	sets of student records
`set of (seq of char)`	sets of sequences of characters (e.g. sets of names)
`set of (set of int)`	sets of sets of integers, e.g. `{ {3,56,2},{-2},{},{-33,5} }`

Defining sets ...

(0) Empty Set: `{}`

(1) Enumeration:

(2) Subrange (integers only): `{integer1,...,integer2}`

e.g. `{12,...,20} =`

`{12,...,12} =`

`{9,...,3} =`

Defining sets ...

(3) Comprehension

{ *expression* | *binding* & *predicate* }

The set of values of the expression under each assignment of values to bound variables satisfying the predicate.

Consider all the values that can be taken by the variables in the binding.

Restrict this to just those combinations of values which satisfy the predicate.

Evaluate the expression for each combination. This gives you the values in the set.

e.g. `{ x**2 | x:nat & x < 5 }`

Defining sets ...

Examples of Comprehensions:

```
{x | x:nat & x < 5}

{y | y:nat & y < 0}

{x+y | x,y:nat & x<3 and y<4}
```

Defining sets ... Finiteness

In VDM-SL, sets should be finite, so be careful when writing comprehensions that you don’t define a predicate that could be satisfied by an infinite number of values.

Example:

Operators on Sets

There are plenty of built-in operators on sets.

Each one has a *signature* defining the number and types of operand expected, e.g. the set union operator:

```
_ union _ : set of A * set of A -> set of A
```

Name of the operator. The underscores show where the arguments go when the operator is used.

Types of the inputs, in order. The "*" separates each input type.

The type of the result.

What can you tell about the union operator from this signature?

Operators on Sets

```
_ union _ : set of A * set of A -> set of A
```

Are the following expressions legal, according to the signature?

```
union({4, 7, 9} {23, 6})
```

```
3 union {7, 1, 12}
```

```
{12,…,15} union {x-y | x,y:nat & x<4 and y<10}
```

```
{} union {}
```

```
{12} union {x**y | x,y:nat & x<4 and y>2}
```

Operators on Sets

In order to use operators effectively, you have to know the number and types of operands that they expect, and the types of value that they return as a result.

This information is given in the notes and the text, and *you really need to know it* in order to use the language effectively.

Operators on Sets

```
_ union _    : set of A * set of A -> set of A
_ inter _    : set of A * set of A -> set of A
_ \ _        : set of A * set of A -> set of A
dunion       : set of (set of A)   -> set of A
dinter       : set of (set of A)   -> set of A
card         : set of A            -> nat
_ in set _   : A * set of A        -> bool
_ subset _   : set of A * set of A -> bool
```

Note: *we don't show the underscores when the operator is normally used in a prefix form, e.g.*

```
card {12, 45, 12, 3} = 4
```

Operators on Sets

distributed operators

The most common operators have special forms in which they are extended to a whole set of arguments, not just two.

```
dunion    : set of (set of A)   -> set of A
```

```
dinter    : set of (set of A)   -> set of A
```

Operators on Sets selecting elements

In side a function definition, we may need to select an *arbitrary* element from a set, not caring how it is selected. We can do this by using a local definition, i.e. in the body of the function say

```
let x in set S in …
```

(now `x` *stands for some arbitrary member of* `S`*)*

Alternatively, we could just define a general function for selecting an element from a set. Since we are not interested in the means of selection, we could do this by an implicit function definition:

```
Select (s:set of X) result:X

pre s <> {}

post result in set s
```

… and now we can use `Select(_)` whenever we want to select an element of a set.

Case Study: the explosives storage example

• The system to be modelled is part of a controller for a robot that positions explosives such as dynamite and detonators in a store.

• The store is a rectangular building. Positions within the building are represented as coordinates with respect to one corner designated the origin. The store's dimensions are represented as maximum x and y coordinates.

• Objects in the store are rectangular packages, aligned with the walls of the store. Each object has dimensions in the x and y directions. The position of an object is represented as the coordinates of its lower left corner. All objects must fit within the store and there must be no overlap between objects.

Case Study: the explosives storage example

The positioning controller must provide functions to:

1. return the number of objects in a given store;

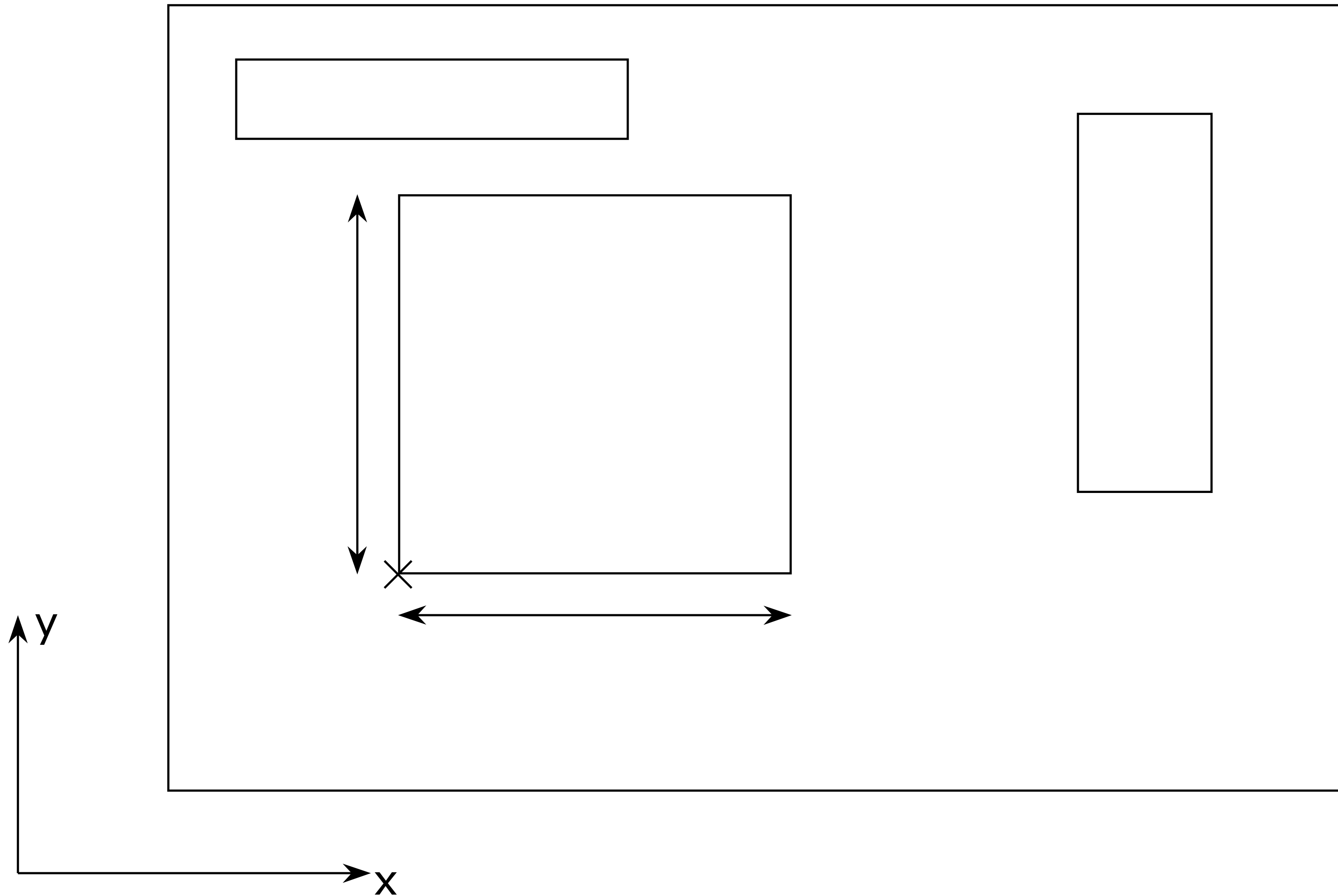
2. suggest a position where a given object may be accommodated in a given store;

3. update a store record to note that a given object has been placed in a given position;

4. update a store record to note that all the objects at a given set of positions have been removed.

Purpose of the model: to clarify the rules for the storage of explosives.

Case Study: the explosives storage example



Case Study: the explosives storage example

```
Store  :: contents :
          xbound   :
          ybound   :

Object :: position :
          xlength  :
          ylength  :
```

Case Study: the explosives storage example

```
Store   :: contents :
           xbound   :
           ybound   :

inv mk_Store(contents,xbound,ybound) ==
```

Case Study: the explosives storage example

```
Store   :: contents :
           xbound   :
           ybound   :

inv mk_Store(contents,xbound,ybound) ==

     forall o in set contents &
           InBounds(o,xbound,ybound) and
     not exists o1, o2 in set contents &
           o1 <> o2 and Overlap(o1,o2)

InBounds: Object * nat * nat -> bool
InBounds(o,xb,yb) ==
```

Case Study: the explosives storage example

```
Store   :: contents :
           xbound   :
           ybound   :

inv mk_Store(contents,xbound,ybound) ==

     forall o in set contents &
           InBounds(o,xbound,ybound) and
     not exists o1, o2 in set contents &
           o1 <> o2 and Overlap(o1,o2)
```

Overlap:

```
Overlap(o1,o2) ==
```

Case Study: the explosives storage example

1. return the number of objects in a given store;

```
NumObjects: Store -> nat
```

2. suggest a position where a given object may be accommodated in a given store;

```
SuggestPos: nat * nat * Store -> Store
```

3. update a store record to note that a given object has been placed in a given position;

```
Place: Object * Store Point -> Store
```

4. update a store record to note that all the objects at a given set of positions have been removed.

```
Remove: Store * set of Point -> Store
```

Case Study: the explosives storage example

```
NumObjects: Store -> nat

NumObject(s) ==
```

Case Study: the explosives storage example

```
SuggestPos: nat * nat * Store -> Store

SuggestPos(xlength,ylength,s) == ???
```

There might be any number of viable positions, but the requirements are not specific about which one ought to be returned - any point with sufficient space will do.

Since we do not have to give a specific point, there is not need to give an algorithm for finding a suitable point: we can use an *implicit function definition* instead.

Case Study: the explosives storage example

An implicit definition does not have a body, but does describe the result by means of a postcondition.

functionName (input vars & types) result & type

pre *precondition*

post *postcondition*

```
sqrt(x:real) r:real

pre   x >= 0

post  r*r = x
```

Case Study: the explosives storage example

```
SuggestPos: nat * nat * Store -> Store

SuggestPos(xlength,ylength,s) == ???
```

```
SuggestPos(xlength:nat, ylength:nat, s:Store) p:[Point]

post  -- if there is a point with enough room
      -- then return some point where there is
      --      enough room
      -- else return nil

      if exists poss:Point &
                RoomAt(x,length,ylength,s,poss)

      then RoomAt(xlength,ylength,s,p)

      else p = nil
```

Case Study: the explosives storage example

```
RoomAt: Object * Store * Point -> bool

RoomAt(o,s,p) ==

      let new_o = mk_Object(p,o.xlength,o.ylength) in
```

Case Study: the explosives storage example

3. update a store record to note that a given object has been placed in a given position;

```
Place: Object * Store Point -> Store

Place(o,s,p) ==

    let new_o = mk_Object(p,o.xlength,o.ylength) in

    mk_Store (




              )

    pre
```

Case Study: the explosives storage example

An extension - Suppose we have a site which consists of a collection of stores:

```
Store :: name : token
         ...

Site = set of Store
inv site ==
  forall store1, store 2 in set site &
       store1. name = store2.name => store1 = store2
```

and we need to take an inventory of the site:

```
Inventory = set of inventoryItem
InventoryItem :: store : token
                 item  : Object
```

Case Study: the explosives storage example

We could take the union of the individual inventories of each store:

```
SiteInventory: Site -> Inventory
SiteInventory(site) ==
  dunion{StoreInventory(store) | store in set site}


StoreInventory: Store -> Inventory
StoreInventory(store) ==
  {mk_InventoryItem(store.name,o) |
         o in set store.contents}
```

Review

The set constructor:

Individual sets can be defined by:

- enumeration
- subrange
- comprehension

Operators on sets: defined with signatures; include distributed versions of union and intersection.

The explosives example:

- use implicit specification (postcondition) when there is no need to give a particular result;
- use auxiliary function definitions to break down and simplify the task of building the model.