Deloitte & Touche Bakkenist

ExSpect

Management & ICT Consultants
Telephone + 31 20 495 23 23
Telefax + 31 20 495 23 45

Wisselwerking 46
1112 XR Diemen
The Netherlands

User Manual

© 1999 Deloitte & Touche Bakkenist, Ex*Spect* is a registered trade mark of ASPT.

Deloitte & Touche Bakkenist

ExSpect

User Manual

Contents

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Introduction

ExSpect

ExSpect, Executable Specification Tool, is a powerful business tool giving organisations the ability to model, monitor and analyse business processes effectively and efficiently. By tracking workloads and money, goods, and information flows you can use ExSpect to determine the service level of your organisation. ExSpect offers the potential for every conceivable kind of simulation and so helps you reach decisions on cost reductions and large-scale infrastructure investment program's. ExSpect has a full-graphic user-interface and a sound formal basis, developed in close co-operation with Eindhoven University of Technology, since 1980. ExSpect users are able to build executable models with ease and speed. Changing developed models is an even simpler matter, since a library of building blocks is automatically generated during development. An advantage of ExSpect is that it is possible to use application libraries specifically prepared for particular fields. Libraries are available for workflow, logistics, administrative processes and more specific business situations.

Deloitte & Touche Bakkenist

Deloitte & Touche Bakkenist offers you a helpdesk for answering your questions related to the tool and modelling. A total service package, including various standard and specially tailored courses plus a workshop is available. When desired, staff from Deloitte&Touche Bakkenist will assist you in developing your business applications.

© 1999 Deloitte & Touche Bakkenist, Ex*Spect* is a registered trade mark of ASPT.

This manual

This manual starts with two tutorial cases to get familiar with the tool ExSpect. These cases are used to show the use of ExSpect for creating executable specifications of high-level Petri nets. It is recommended to use the on line help for further explanation on the user interface. The first tutorial illustrates the creation of a small telephone number database. The resulting specification is directly executable. This first tutorial gets the user acquainted with the use of the tool. The second tutorial uses predefined building blocks to demonstrate the power of simulation.
The tool tutorials are followed by a language tutorial (Chapter 3). All features of the ExSpect language are discussed and illustrated with useful examples.

Chapter 4 contains a formal description of the syntax of the ExSpect language. Chapter 5 is a reference manual containing all library functions.

The last chapter is the bibliography. It contains numerous useful references to related material.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

1. Tutorial 1: Building a phone book

Introduction

This tutorial will give an introduction to the way a specification of a simple information system is built. The purpose of this tutorial is to show the reader what considerations and steps are taken when building an actual system.

Description of the case

The information system we are going to discuss is a phone book. A phone book is a table with information about phone numbers. In Table 1 an example of this information is given. The phone book is a very simple example of an information system. But nevertheless this example contains most ingredients that are relevant in larger systems.

Name	Phone number
Jack	020-2210922
Gary	+39-227-644413
Mary	040-2471127
Frank	045-5678230
Patrick	+01-518-2766261

Table 1

A user of a phone book can do four things:

- Look up a phone number.
- Add a new phone number to the phone book.
- Remove an existing phone number from the phone book.
- Change an existing number in the phone book.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Let's examine these four actions:

- If a user searches a phone number (s)he has to enter a name. The answer of the phone book will be the number belonging to this name. If the name doesn't exist in the phone book, the output of the phone book will be: 'not found'.
- If the user wants to *add* a phone number (s)he has to enter a name and a phone number.
- If a user wants to *remove* a phone number, only the name belonging to the phone number has to be given.
- To *change* a phone number a name and a phone number have to be given.

To keep this example simple we assume the following:
First of all, phone numbers that are to be changed or removed are always in the phone book. Second, phone numbers that are added are not already in the phone book. Finally when a search for a phone number is performed the entered name is uniquely present in the phone book. A more realistic system won't be based on these assumptions and will have to act accordingly in these kinds of events.

Getting started

Before we start with the actual building of a system some general actions have to be taken.

1. Start ExSpect .
2. Next a new file has to be defined for the specification. Selecting New from the File menu does this.
3. By selecting Save As from the File menu we can give our file an unique name, we have chosen the name: phone.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

4. Every specification needs at least one system, so we have to define a system for our phone book. To do this we select System Definitions from the Components menu. Now a window opens with all defined systems (this window is empty in our case for the simple reason that we have not yet defined a system). Now we click with the left mouse button on the New button; a window opens in which we can give a name to our system (the default name is 'System'), we choose the name Phone_book (has to be one word). After clicking the OK button with the left mouse button the System Definition window of the Phone_book system will open.

Building the system

In figure 1 the process we are going to design is given. But before we start defining this phone book, we recall the (informal) description earlier.

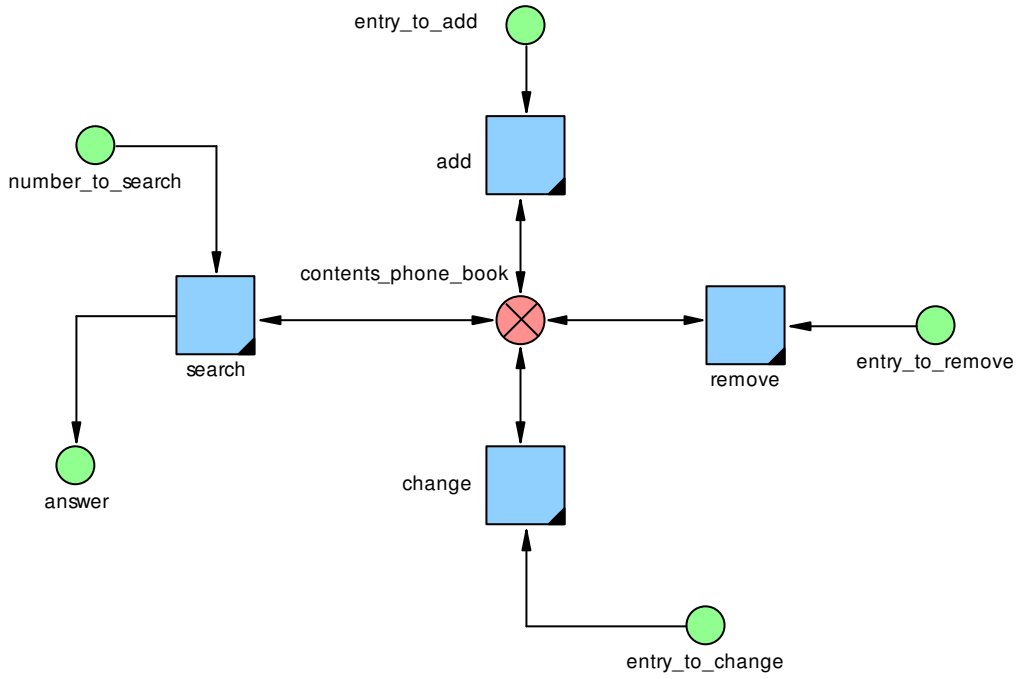

Figure 1

Placing channels

The user can initiate four actions (searching, adding, removing and changing). Therefore

channel answer.

1. Click with the left mouse button on the Channel button:  .
2. Click with the left mouse button in the System Definition window at the place you want to put the channel. Now a channel with an automatically generated name will appear.
3. We repeat 1 and 2 until we have five channels in the definition window.
4. By double clicking a channel with the left mouse button, the Channel Definition window will open. In the Name box of this window we can give each channel an appropriate name (use the names we have chosen above).
5. Press the OK button to save and close the Channel Definition window.

Placing stores

The phone book has to keep information about the phone numbers and names that are put into it. Therefore we have to define a store.

1. Click with the left mouse button on the Store button:  .
2. Click with the left mouse button in the System Definition window at the place you want to put the store. Now a store with an automatically generated name will appear.
3. By double clicking a store with the left mouse button the Store Definition window will appear. In the Name box of this window we can give the store its appropriate name. The name we use for our store is contents_phone_book.
4. Press the OK button to save and close the Store Definition window.

The contents of the resulting system definition window will be something like Figure 2.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

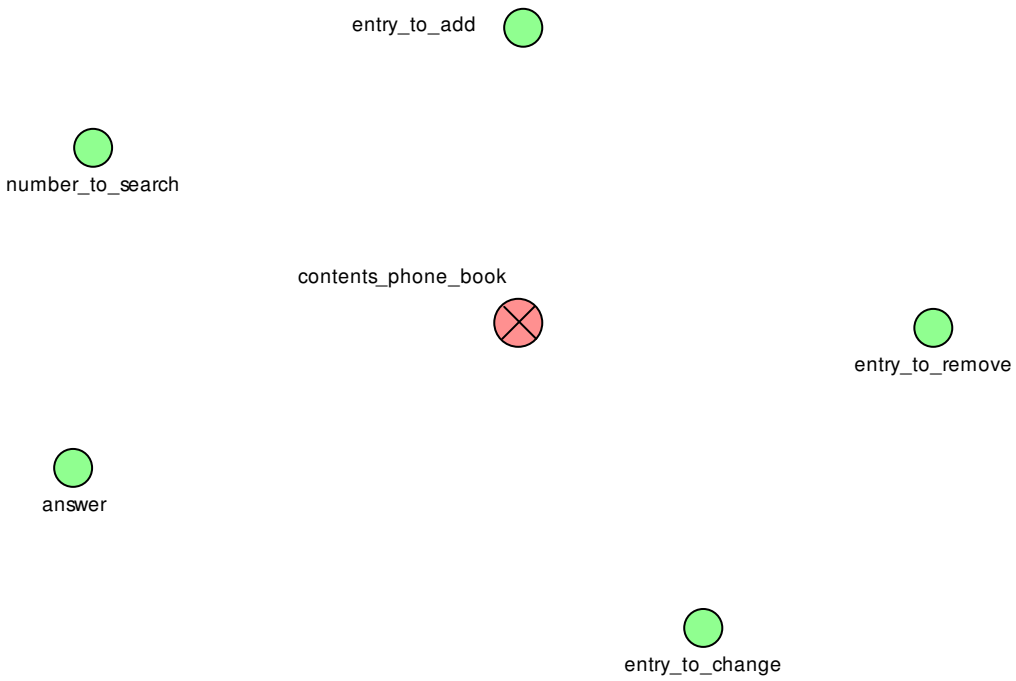

Figure 2

Defining and assigning types

In the next phase we have to think about the types that need to be assigned to the channels and stores. Through the channel number_to_search a name can be entered, so the type of this channel must be str (string). The assigning is done in the following way:

1. Double-click with the left mouse button the channel number_to_search. The Channel Definition window will open.
2. Enter str in the Type box.
3. Press OK to save and close the Channel Definition window.

The channel answer produces the searched phone number or (if no such name exists in the phone book) 'Not found', so the type we assign to answer is also str. By using the channel entry_to_add we can add a new entry (name and phone number) to the phonebook. Therefore we define a type entry as the tuple type [name:str, phone_number:str].

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

1. Select type definitions from the components menu. Now the Type Definitions window will open.
2. Add a new type definition by pressing New, in the Type Definitions window.
3. Enter entry in the name box of the Type Definition window.
4. In the definition box the type can be specified, in our case [name:str, phone_number:str].
5. Press OK to save and close the Type Definition window.
6. Press CLOSE to close the Type Definitions window.
7. Assign entry to entry_to_add.

The channel entry_to_remove can be used to enter a name to remove a name and the entry belonging to this name. Therefore we assign str to the channel entry_to_remove. The channel entry_to_change is used to change existing phone numbers, so we assign the type entry to entry_to_change. This leaves the type of the store contents_phone_book. This is a set of entries so the store contents_phone_book has the type $entry.

Placing processors

After assigning types to all the channels and stores, we have to define the active components of the specification (processors). In our case we need not define any subsystems because of the simplicity of the system to be designed. So we add four processors, one for each action that can be taken. The names of these processors will be: search, add, remove and change.

1. Click with the left mouse button on the Processor button:  .
2. Click with the left mouse button in the System Definition window at the place you want to put the processor. Now a processor with an automatically generated name will appear.
3. We repeat 1 and 2 until we have four processors in the System Definition window.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

4. By double-clicking a processor the Processor Definition window will open. In the Name box of this window we can give each processor it's appropriate name (use the names search, add, remove and change).

Connecting places/stores and processors

Before we can finish the processor definitions, we have to place the connections between our objects. The following connections have to be made:

1. The store contents_phone_book has to be connected to **all** processors.
2. A connection **from** the channel entry_to_add **to** the processor add.
3. A connection **from** the channel number_to_search **to** the processor search.
4. A connection **from** the processor search **to** the channel answer.
5. A connection **from** the channel entry_to_remove **to** the processor remove.
6. A connection **from** the channel entry_to_change **to** the processor change.
7. Click with the left mouse button on the Connector button:  .
8. Select the source of your connection (for example number_to_search). Moving the mouse pointer above the object and pressing the left mouse button does selecting.
9. Select the destination of your connection (for example search)
10. Repeat 1,2 and 3 until all connections are made.

Note: By double-clicking the connector button, several connections can be made without having to select the connector button again. The same can be done when placing channels, stores, systems and processors. Pressing the pointer tool button does deselecting a button  .

The contents of the resulting System Definition window is shown in Figure 3.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

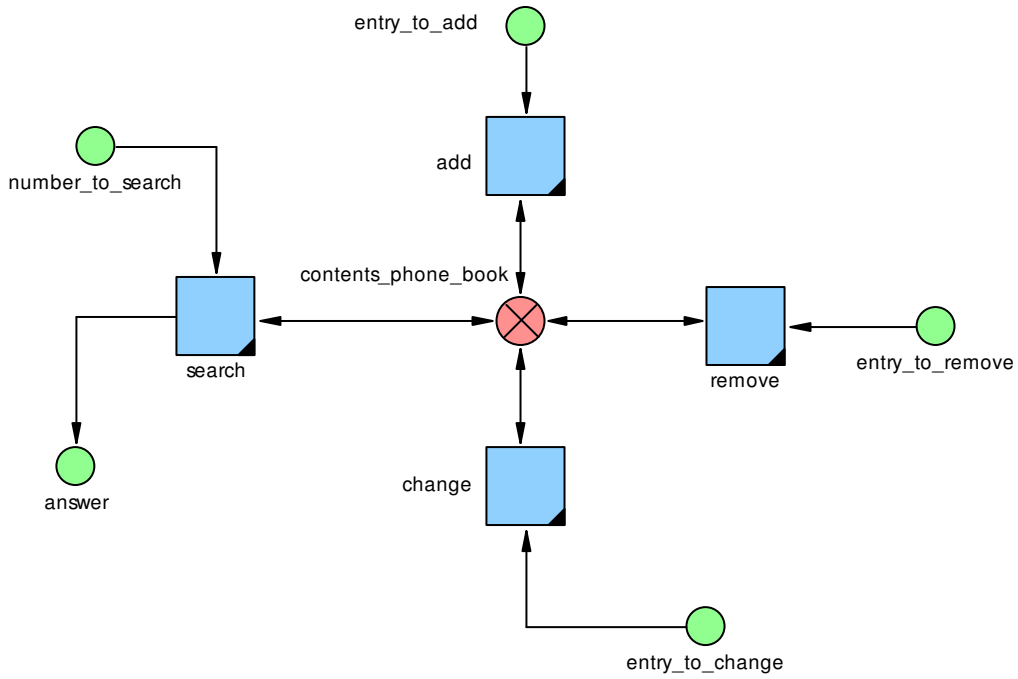

Figure 3

Should one of the connections accidentally have been placed wrong, then the connection has to be removed and the processor definition updated. The following can be done to correct such an error:

1. Double click the wrongly connected processor, the Processor Definition window will open.
2. Select the wrong connection in the Connections box, by pressing the button in front of the connection in question.
3. Select Delete from the Edit menu.
4. Press OK to save and close the Processor Definition window.

The next step is to give the processors the required functionality so they will have the appropriate input/output behaviour. In the remainder the functionality of each processor is given and described. This functionality is entered in the definition box of the Processor Definition window. The inexperienced reader probably will not understand the full functionality that

is entered. This is no problem since the purpose of this tutorial is to give the reader an idea on how to work with ExSpect. The language that is used to specify the behaviour is explained in detail in chapter 3.

Processor: search

The way we want the processor search to behave is the following:
if the entered query is in the phone book
then return the phone number
else return a message that the number was not found.

This is implemented in the following way:

1. Double-click the processor search.
2. Enter the following text in the definition box of the processor definition window:

```
if number_to_search elt rng([x:contents_phone_book|x@name])
then
    answer <- pick(set([x:contents_phone_book|x@name =
    number_to_search]))@phone_number
else
    answer <- 'not found'
fi
```

3. Press OK to save and close the processor definition window.

Processor: add

The processor add inserts a new phone number into the phone book. This is implemented in the following way:

1. Double-click the processor add.
2. Enter the following text in the definition box of the processor definition window:

```
contents_phone_book <- entry_to_add ins contents_phone_book
```

3. Press OK to save and close the processor definition window.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Processor: change

The processor change changes the number belonging to one of the names in the phone book. Inserting a new (changed) entry and removing the old one from the phone book does this. This is implemented in the following way:

1. Double-click the processor change.
2. Enter the following text in the definition box of the processor definition window:

   contents_phone_book <- entry_to_change ins
   (set([x:contents_phone_book|x@name != entry_to_change@name]))
3. Press OK to save and close the processor definition window.

Processor: remove

The processor remove removes an entry (with a given name) from the phone book. This is implemented in the following way:

1. Double-click the processor remove.
2. Enter the following text in the definition box of the processor definition window:

   contents_phone_book <- set([x:contents_phone_book|x@name !=
   entry_to_remove])
3. Press OK to save and close the processor definition window.

Now the specification is finished and ready to be checked. Clicking the Translate button does this:  . This will first save the specification and then check it for mistakes. If we do this the following translation error will appear: ‘store/ phone_book/contents_phone_book: empty init value’. By double clicking this error with the left mouse button, the Store Definition window will open. Here we see that we have forgotten to give our store (contents_phone_book) an initial value. We choose that the initial value of contents_phone_book is empty. Therefore we insert {} in the Initial Value box. Now we translate our specification again and we see that there are no error messages left.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

The use of the phone book

In the following we describe the use of our information system. But before we can do this we have to start the simulation by pressing the simulation button:  . This will initiate three consecutive actions: first the specification will be saved, then checked and finally the simulation will be started.

Adding entries to the phone book

We start by entering several entries into our phone book.

1. Select the place entry_to_add.
2. Press the right mouse button and select Add Token from the pop-up menu.
3. Enter a valid entry in the Add Token window and press the Apply button. For this example we have entered the first three rows from Table 1. After this pressing the Close button closes the Add token window.
4. Next we start the simulation by pressing the Play button  . We see that the new entries are added to contents_phone_book.

Adding a table dashboard object

Now we want to check if the right entries are added to contents_phone_book. Adding a dashboard object is the best way.

1. First activate the dashboard window by pressing the Dashboard button:  .
2. Press the Table button:  and move the mouse pointer in the Dashboard window. Now press the left mouse button and an indication of a table will appear.
3. Double-click the indication of the table with the left mouse button and the Dashboard Object Properties window appears.
4. First we give a name to the dashboard object by entering a name in the Name box.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

5. Then we enter a place or store of which we want to see the contents in the place box. In our case this is Phone_book.contents_phone_book (System name, dot, name of the channel/store).
6. Click OK to save and close the Dashboard Object Properties window.
7. In the dashboard object the contents of contents_phone_book can be seen.

Changing entries in the phone book

Now that these three entries are entered we would like to change one of them. So suppose Gary moves and gets a new number say 040-2471234.

1. Select the channel entry_to_change.
2. Press the right mouse-button and select Add Token from the pop-up menu.
3. Enter *Gary* in the Name field and *040-2471234* in the Number field.
4. Press the Play button. We see that contents_phone_book is updated.
5. Check the value of contents_phone_book by looking at the dashboard table.

Removing entries from the phone book

Now we want to remove an entry from our phone book. So suppose Mary marries Gary and moves in with him. Now we can remove Mary from our phone book.

1. Select the channel entry_to_remove.
2. Press the right mouse-button and select Add token from the pop-up menu.
3. Enter *Mary* in the Add Token window.
4. Press the Play button. We see that contents_phone_book is updated.
5. Check the value of contents_phone_book by looking at the dashboard table object.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Searching numbers in the phone book

Finally we want to search for a phone number. Since it has been a long time since we spoke Jack we are going to search for his phone number.

1. Select the place number_to_search.
2. Press the right mouse button and select Add Token from the pop-up menu.
3. Enter *Jack* in the Add Token window.
4. Press the Play button. Now we see that a number is searched in the phone book and an anwer is produced in the channel answer.

To check the contents of answer we need to add another dashboard object to the dashboard. This time we choose the Textbox dashboard object.

1. Press the Textbox button:  and move the mouse pointer in the Dashboard window. Now press the left mouse button and a Texbox will appear.
2. Press the connector button.
3. Select the channel answer in the System Definition window.
4. Select the Textbox in the Dashboard window. Now the Textbox will show the contents of answer. (Note: the name of the Texbox is automatically changed to answer).

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

2. Tutorial 2: Simulating a petrol service-station

Introduction

In this tutorial the case of a petrol service station is used to explain the basic principles of simulation with ExSpect. Simulation is an analysis technique that mimics reality by using a model. It enables experimentation with a model of the reality. This is useful when real experiments are expensive or dangerous.

Case description

A taxicab company has its own petrol service station where the taxi drivers can refuel their cars. A taxi driver that needs to refuel his car is obliged to drive to this service station. The station has a queue that can hold three cars. When there is no room in the queue the taxi driver must drive on and go to another service station. It takes between 2 and 5 minutes to refuel a car and there is one petrol pump available.

The taxicab company is growing and lately 8% of the refuelling takes place at another service station. Management suspects that taxi drivers go directly to other service stations to win time. They want to use simulation to find answers to the following questions:

- Do taxi drivers evade the rule to go to the company's service station first?
- Taxi drivers complain about waiting times of ten minutes or more. Is this true?
- What is the best investment alternative for the service station? Should the waiting area for the queue be enlarged, should a faster one replace the pump or should a second pump be added?

Getting started

In this tutorial an incomplete model of the Taxicab Company has to be finished so it can be used to run a simulation. What has been omitted from the model is the petrol service station.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Perform the following steps to get a copy of an ExSpect specification of the environment of the petrol service station.

1. Start ExSpect
2. Select Open from the File menu.
3. In the Folders box select the folder ‘Tutorial\Petrol Service Station’.
4. In the File Name box select the file ‘Template.ex’.
5. Choose OK.
6. Select Save As from the File menu.
7. In the File Name box type a name for the copy of the file, for example ‘pump.ex’
8. Choose OK.

The main system

In the System Definitions window double click the system main to view the system petrol_service_station and its environment (see Figure 4).

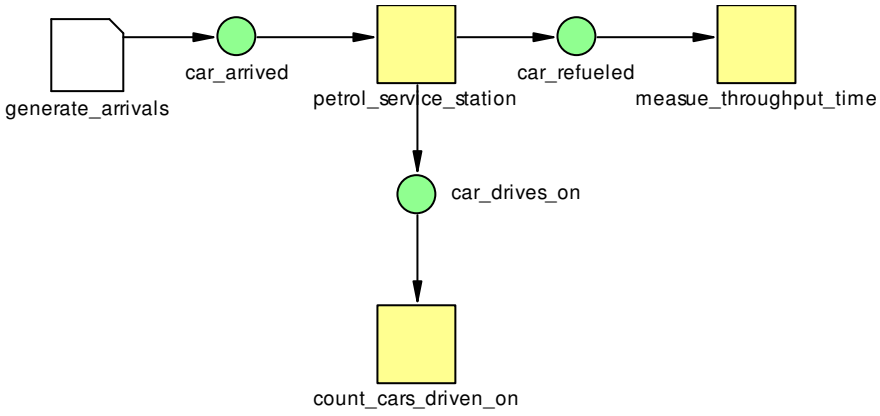

Figure 4 - System main

All interaction of the service station with its environment is the arrival or departure of cars. The environment is built with building blocks from a library. The system labelled generate_arrivals is an installation of the building block generator. This system generates tokens of type Car and places them in the channel car_arrived. This models the arrival of cars at the

service station. The system petrol_service_station consumes the cars from the channel car_arrived and depending on the situation puts them in the channel car_refueled or car_drives_on. The systems measure_throughput_time and count_cars_driven_on collect all kinds of simulation information.

Double click the system petrol_service_station. This is the system that has to be completed. It has one input connector (car_arrived) and two output connectors (car_drives_on and car_refueled). Figure 5 shows the subsystem we are going to build.

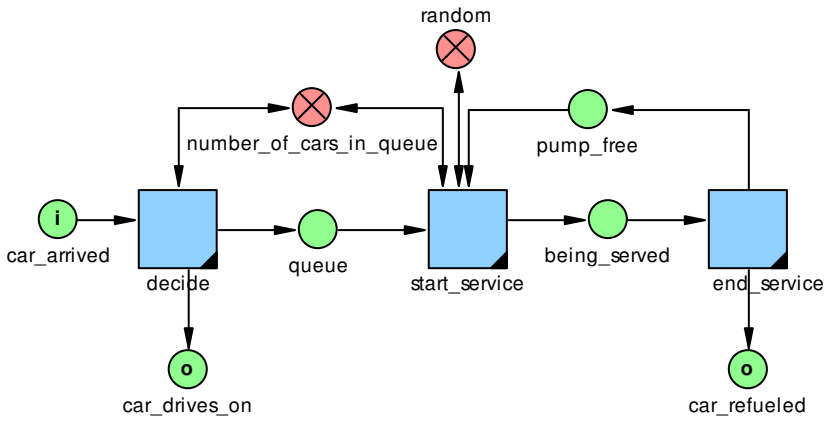

Figure 5 - System petrol_service_station

Adding channels

After arrival in the service station a car will enter the queue when there is enough room for it, otherwise it will drive on. These two possible outcomes of the decision are modelled by the channels queue and car_drives_on (see Figure 5). The decision is modelled by the processor decide. The processors start_service and end_service model the beginning and the ending of the service. When a car is waiting and the pump is free the processor start_service fires. The car is moved from the queue to the channel being_served. When the service is ready the transition moves the car to the output channel car_refueled.

To add the channels do the following:

1. In the submenu Add of the menu Object select Channel (or use the channel button ).
2. Move the mouse pointer over the gas station system window. The cursor will change into the symbol for a channel.
3. Click the left mouse button at the place you want to put the channel.
4. Double click the new channel. A property window appears.
5. In the Name box type queue.
6. Choose OK.

Repeat this for the channel pump_free and being_served.

Adding types

To make the definition of the channels complete we have to specify the appropriate type.

Make the following changes:

1. Double click the channel queue. A property window appears.
2. In the Type box select the type Car.
3. Choose OK.

Repeat this for the channel being_served. (Tip: when you place a channel you can use the right mousebutton. A pop-up menu appears from which you can select the channel type.)

The channels being_served and pump_free model the situation that a car is being served or not. In the initial state there is one token in the channel pump_free. The type of this channel is not important. The fact that the pump is free is denoted by the fact that the channel contains a token. The type of that token is not relevant. To keep the specification simple we choose the type str.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

1. Double click the channel pump_free. A property window appears.
2. In the Type box select the type str.
3. In the Initial Tokens list type 'pump' in the first row.
4. Choose OK.

Adding a store

The processor decide needs to know the number of cars in the queue. This number is kept in the store number_of_cars_in_queue. When a car is put in the queue the value of this store is increased by one. When a car is removed from the queue the value is decreased by one.

Add the store to the specification by performing the following steps:

1. In the submenu Add of the Object menu select Store (or use the store button: ).
2. Move the mouse pointer over the gas station system window. The cursor will change into the symbol for a store.
3. Click the left mouse button at the place you want to put the store.
4. Double click the new store. A property window appears.
5. In the Name box type number_of_cars_in_queue.
6. In the Type box select the type num.
7. Choose OK.

Adding processors

Now that all channels and stores are created it is time to create the processors. They have already been discussed in the previous paragraph:

1. In the submenu Add of the menu Object select Processor. (or use the processor button ).
2. Move the mouse pointer over the gas station system window. The cursor will change into the symbol for a processor.
3. Click the left mouse button at the place you want to put the processor.
4. Double click the new processor. The Processor Definitions window appears.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

5. In the Name box type decide.
6. Choose OK.

Repeat this for the processors start_service and end_service.

Adding connections

Before we can finish the definition of the processors the channels/stores/pins and the processors have to be connected. Connecting channels/stores/pins and processors is straightforward.

To connect the pin car_arrived with the processor decide do the following:

1. Click on the Connector button  .
2. Click on the source of the connection. In this case this is the input pin car_arrived.
3. Click on the destination of the connection. In this case this is the processor decide.

Repeat this for the following connections:

- The store numbers_of_cars_in_queue is connected to the processors decide and start_service.
- A connection **from** the processor decide **to** the channel queue.
- A connection **from** the processor decide **to** the output pin car_drives_on.
- A connection **from** the channel queue **to** the processor start_service.
- A connection **from** the processor start_service **to** the channel being_served.
- A connection **from** the channel being_served **to** the processor end_service.
- A connection **from** the processor end_service **to** the output pin car_refueled.
- A connection **from** the processor end_service **to** the channel pump_free.
- A connection **from** the processor pump_free **to** start_service.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Defining processors

The last step in the creation of the service station is the definition of the processors. Remember that the processor decide had to move the car from car_arrived to queue when there was enough room in the queue, otherwise it had to be moved to the channel drive_on. Also remember that in the first case the number in number_of_cars_in_queue had to be increased.

The following steps achieve this:

1. Double click the processor decide
2. In the definition box type the following:

```
if number_of_cars_in_queue < 3 then
   queue <- car_arrived,
   number_of_cars_in_queue <- number_of_cars_in_queue +1
else
   car_drives_on <- car_arrived
fi
```

3. Choose OK.

An assignment statement is used to move a token from one channel to another. In the ExSpect language an assignment statement looks as follows:

Output channel <- *Input channel*
For example the statement queue <- car_arrived in the second line of the processor means that a token is moved from the channel car_arrived to the channel queue. Another language feature that is used is the if statement that looks like this:

if *expression* then *alternative1* else *alternative2* fi

When *expression* evaluates to true *alternative1* is performed otherwise *alternative2*. In the example it is used to make the decision.

The next chapter explains the ExSpect language in more detail.

Adding a random store

From the case description we know that it takes between 2 and 5 minutes to refuel a car. We assume that this service time has a uniform distribution. To model this random behaviour the processor start_service is connected with a random store. A random store is a store that always contains a new random number.
To add the random store do the following:

1. In the submenu Add of the menu Object select Random Store. (Tip: use the random store button:  .)
2. Move the mouse pointer over the gas station system window. The cursor will change into the symbol for a random store.
3. Click the left mouse button at the place you want to put the store.
4. Connect the store with the processor start_service.

Now the definition of the processor start_service can be added. Put the following definition in the processor start_service:

```
number_of_cars_in_queue <- number_of_cars_in_queue -1,
being_served <- queue delay uniform(2.0,5.0,random)
```

The keyword delay in the last statement specifies that the token becomes available in the channel filling_tank only after a certain amount of time. This models the time it takes to refuel the car. In the next chapter this will be explained in more detail.

To finish the model add the following definition for the processor end_service:

pump_free <- 'pump',
car_refueled <- being_served

Running a simulation

The model that has been created in the previous paragraphs can now be executed. To start the simulation mode do the following:

1. In the Tools menu choose Simulate (or press the simulation button ).

An *Incompleteness Errors* window appears containing an error message about an empty initial value. We forgot to enter an initial token value for the store number_of_cars_in_queue.

This error must be resolved first:

1. Double-click the error message. The store definition window for the store number_of_cars_in_queue appears.
2. In the Initial Value box type 0.
3. Choose OK

Now we can start the simulation:

1. Push the simulation button. When the specification is free of errors a System Animation window will appear for the system main.
2. Double click the system gas station to get its System Animation window.
3. In the menu Simulation select Play (or press the Play button: ).

In the System Animation window you can see the tokens flow through the process. The simulation time is shown in the right bottom corner of the screen. If you run the simulation long enough it will halt automatically.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Dashboard objects

It is not directly observable whether the queue is full or not. We will add a traffic light to the picture to solve this.

1. Press the Halt button:  . The simulation will pause.
2. Press the Animation Dashboard object button:  .
3. Move the mouse pointer over the gas station system animation window. The cursor will change into the symbol for an animation dashboard object.
4. Click the left mouse button at the place you want to put the animation object.
5. Double click the new animation object. A Dashboard Objects Properties window appears.
6. In the Label box type ‘Traffic light’.
7. In the Place box type main.petrol_service_station.number_of_cars_in_queue.
8. In the first row of the Animation Frames list type ‘green.bmp’
9. In the second row of the Animation Frames list type ‘green.bmp’
10. In the third row of the Animation Frames list type ‘green.bmp’
11. In the fourth row of the Animation Frames list type ‘red.bmp’
12. Size the animation object to make it look like a traffic light.
13. Choose OK.

It is also possible to view the number of tokens in a channel:

1. Double click the channel queue. A property window appears.
2. Check the checkbox Counter Visible.
3. Choose OK.

Resume the simulation by pressing the Play button. Now you will see that the traffic light turns red when the queue contains three cars. Close the System Animation windows and wait until the simulation is finished (the simulation will run faster if all windows are closed).

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Simulation results

The systems measure_throughput_time and count_cars_driven_on collect all kinds of simulation information. For this case the number of cars that have driven on and the throughput time are interesting. For every car the waiting time and the service time is measured. The sum of these two numbers is the throughput time. To view the simulation results we add some dashboard objects to the dashboard. With the dashboard the simulation can be monitored.

1. In the Simulation menu select Show Dashboard (or press the dashboard button ).
2. Push the Table button.
3. Move the mouse pointer over the Dashboard window. The cursor will change into the symbol for a table dashboard object.
4. Click the left mouse button at the place you want to put the table object.
5. Double click the new table object. A Dashboard Objects Properties window appears.
6. In the Place box select the channel main. measure_throughput_time.tmeasurement.cumm_results.
7. Choose OK.
8. Resize the table if necessary.

Figure 6 shows an example of a table containing simulation results. The table contains a row for every subrun. The column *arrivals* shows the number of tokens that arrived in the measure, which is the number of cars that have refuelled. The column *average* contains the average throughput time of all tokens in the subrun. This is the average time it took to refuel the cars in the subrun. The column *variance* indicates the reliability of the average throughput time. For the interpretation of the variance we refer to [Ros90].

	subrun	xarrivals	xaverage	xvariance
	1	109	6.954936082020955	12.81203172238193
	2	110	6.195319184636666	7.466758890373583
	3	93	7.167902520024491	7.073428243566428
	4	107	6.944454749455748	11.60485782514961
	5	105	7.754224983777169	13.06791146543264
	6	109	6.577307048234441	10.96258891151067
	7	113	8.040644461882469	16.82494936074397
	8	100	6.701869069717154	12.54726194834263
	9	107	7.260009003453624	12.4713196171978
	10	105	7.265234013310086	12.35795898710189

Figure 6 - Simulation results

When the differences between the results in the subruns are high it means that the results are unreliable. Because in this case for all subruns the average is near 7 minutes we can conclude that the throughput time is indeed about 7 minutes. With this information it is however impossible to give an answer to the question whether waiting of ten minutes or more occur.

We need the results of the measure count_cars_driven_on when we want to know how many cars had to drive on in comparison to the number of cars that could be serviced.

To view these results:

1. Double click the table object. A Dashboard Objects Properties window appears.
2. In the Place box select the channel main.tel_doorrijders.tmeasurement.cumm_results.
3. Choose OK.
4. Size the table if necessary.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Figure 7 shows an example of the results. Because the cars are not serviced the throughput time and the variance are 0. When the column *arrivals* is compared to this column in Figure 6 we can see that indeed about 8% of the cars have to drive on. This means that the question whether drivers evade the rules can be answered negative.

	subrun	xarrivals	xaverage	xvariance
	1	12	0.	0.
	2	5	0.	0.
	3	5	0.	0.
	4	9	0.	0.
	5	9	0.	0.
	6	10	0.	0.
	7	17	0.	0.
	8	5	0.	0.
	9	6	0.	0.
	10	12	0.	0.

Figure 7 - Simulation results

Other scenarios

As explained in the case description the management want to know what the best investment is to reduce the number of cars that drive on. If possible they also want to reduce the average throughput time. For each investment scenario you can change the model and run a simulation. When you have done that and carefully analysed the results you can advise the management of the service station.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

3. The language

Introduction

Learning to specify in a functional language is like learning a skill: you have to know about the materials you are working with, about the tools to work with, and about how to operate the tool on the materials in order to create a result.

Materials

In a functional specification, the materials we use are *types and values*. The tools we use are type and value constructions and expressions. The result we want to obtain is a token value. In ExSpect this will be a specification of (part of) the body of a processor, in this case a specification of how to build an output token value from the input token values.

In the tutorials in the previous chapters we already saw that the body of a processor specifies the exact way to produce output token values from input token values. We will give a short rehearsal here.

In the simplest form, a processor copies the value from an input channel to an output channel, without change. This is expressed in the processor's body with an assign statement:

```
output <- input
```

This means that the value of the token produced for the channel 'output' is the same as the value of the token on channel 'input'.

We also saw that we could specify a delay by which the output token should be available. This is expressed by:

```
output <- input delay 2.0
```

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

which means that the token for channel 'output' becomes available only 2.0 time units after the moment of firing of the processor.

We saw furthermore that a processor could *select* output channels under certain restrictions. This is expressed by an 'if statement':

```
if input > 0 then output <- input fi
```

This means that a new token value for channel 'output' will only be produced if the token value of channel 'input' is positive.

The statement:

```
if input ≤ 0 then
    output1 <- input
else
    output2 <- input
fi
```

expresses that negative values and ø are copied to output1 and positive values are copied to output2.

It is even possible to impose restrictions on the input token *before* taking it into an input channel. By means of preconditions it is possible to select tokens from an input channel:

```
pre: input > 0
```

A processor with these preconditions only accepts tokens on channel input, which are positive. All other tokens on that channel will not trigger the processor.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

The value of the output token can be derived from the value of input tokens by means of a function:

```
output <- function (input)
where function [a: . . .]: = . . .
```

In this tutorial we will learn how to specify the body of a *function*, that is, how we can derive a new value from other values.

First of all we will learn how to use *types* in order to classify values. This consists of an introduction of *basic types* and *type constructors*.

Together with types, we will introduce *constant values* of these types. Together with type constructors we will show how to construct new values with *value constructors*.

Types and values will be the materials we are going to use. Type constructors and value constructors will be the first tools we will handle.

Next, we will see how to define and use functions, another set of tools.

The ExSpect language and tool can be used in two ways. One way is to combine predefined building blocks (processors and subsystems) to specify a large system. However, not all systems can be built in this way. It may be necessary to adapt building blocks or even to create them from scratch.

For this use of ExSpect - as a high-level programming language - this tutorial is destined. The part that deals with these aspects is called the functional part of ExSpect. It resembles typed functional programming languages, hence this qualification. The most striking difference of the ExSpect language compared to ordinary (third-generation) programming

© 1999 Deloitte & Touche Bakkenist, Ex*Spect* is a registered trade mark of ASPT.

languages are the absence of the sequencing (semicolon: ';') operator. In the C language, for example, introducing an auxiliary variable help and executing the following statements performs interchanging the values of variable x and y usually.

```
help = x; x = y; y = help
```

In ExSpect, this is done by the following simultaneous assignment:

```
x <- y, y <- x
```

It is assumed that x, y are stores here. In a simultaneous assignment, the variables (stores) retain their value until the complete assignment has been executed. The order in which they are given is of no effect.

Another difference is that ExSpect allows assignments of very complex datatypes in one stroke; for instance the assignment:

```
s <- [x: stretch(0, 100)| x*x]
```

fills the store or channel s with a table giving the squares for the numbers 0 up to 100:

```
{<<0, 0>>, <<1, 1>>, <<2, 4>>, ..., <<100, 1000>>}
```

Stores, channels and processors

The functional part of ExSpect is about defining stores, channels and processors. Stores and channels are defined by attaching types to them. A type represents a set of values. Processors are defined by attaching value expressions or terms to them. A term represents a value, which may depend upon some parameters. We give an example: the system FINANCE depicted in the figure below.

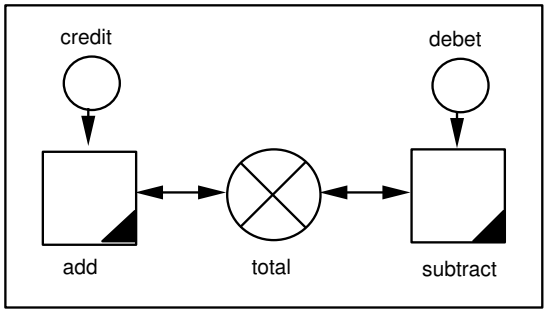

Figure 1: Finance system

The channels credit, debit and the store total are all typed with the basic type num (numeral). Non-basic types are constructed from simpler types by means of type constructors, for example num><num (numeral pair) or [name:str, salary:num] (a record or tuple).

A *type definition* consists of an identifier (the defined type) and a type (its definition). The identifier can then be used in types instead of its definition. *Type expressions* are types with variables. They are used in *signatures* to describe the behaviour of *functions*.

The processors in figure 1 are both installations of the processor book with the following definition:

Connections:

```
amount:  in:     num
total:   store: num
```

Pre:

```
amount > 0
```

Val / fun parameters:

```
kind:  val:    str
```

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Definition:

```
if kind='credit' then
    total <- total-amount
else
    total <- total+amount
fi
```

The installations are given in the following table:

name	*amount*	*total*	*kind*
add	credit	total	'credit'
subtract	debit	total	'debit'

In the example, the values 'debit', 'credit', the conditions amount>0, kind='credit' and the assigned values total-amount, total+amount all are *terms*.

Terms can be either atomic (constants), a variable or constructed by means of functions or *quantors* from simpler terms. In the example, 'debit', credit', 0, are atomic terms, kind, amount and total are variables, whereas >, =, -, + are function symbols representing functions. We will discuss quantors later. We shall first deal with types and values and then turn to terms and functions. We then explain in more detail how processors and systems are defined and installed. We conclude by explaining how to create and use modules.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

3.1 Lesson 1: Types and values

Introduction

Types represent sets of values. Basic types are bool, str, num and real. They contain atomic values called *constants*. Non-basic or constructed types contain basic types, one or more *type constructors* plus brackets ((,)) to define the order in which the type constructors are applied. Their values are represented by constructions containing constants and special function symbols called construction symbols. We first treat basic types and their values, then type constructors and constructed values. We conclude by treating type definitions.

Basic types and constants

The basic types are bool, str, num, real and void.

bool

The type bool contains two values: the boolean constants false and true.

Str

The type str contains all printable ASCII strings. These strings are represented between quotes. So '#%AC/DC' is a constant of type str. To represent a quote in a string constant, it must be doubled. So '"a"' represents the string 'a'. The empty string is '', which is also a valid constant. ASCII strings containing non-printable characters, for example new lines are not in this type.

Num

The type num contains the rational numbers, i.e. natural numbers (0, 1, 2), negative integers (-1, -2), and their quotients, like 5/17 or -191/27. The rational numbers have no upper or lower bound; ExSpect can handle 1000-digit numbers and their quotients. Of course, manipulations with large numbers will be rather time consuming.

Real

The type real is a numeric type like num. To distinguish them from nums, reals must have a single decimal dot somewhere. Note that for example 2 and 2. are different constants, although they might mean the same to a user.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

So -2. and 3.14159 are reals, whereas -2 and 314159/100000 are nums. Manipulation with reals is faster than with nums, but at a price: reals are approximated. So 2.5+2.5=5. need not necessarily hold, the result might be for example 4.9999999999761. The approximation made depends on the hardware platform used. There is also a maximum real; exceeding this maximum gives rise to errors similar to division by zero.

Void

There is a fifth basic type void, which has no constants. It is used in type constructions to denote some "degenerate" values.

Constructed types and their values

To construct new types, we use our type constructors $, *, ><, -> and the record operator. Basic types and type constructors represent sets of constructed (non-atomic) values. In the sequence, we assume that A, B, A', ... are types.

Sets

The type $A contains finite sets of values from A. Enclosing their elements in braces represents these sets. So $num contains for example the values {} (the empty set), {1}, {-2, 5/3}, {3599/333} and so on. The value constructor is here {_, ..., _}. Note that the empty set {} belongs to all set types. This fact is reflected by giving it the degenerate type $void. It is not allowed to put values from different types within the same set, so for example {1, -2.0} is an illegal value.

Note that bool and {false, true} have the same elements, but are nevertheless different: one is a type, the other a value.

Lists

The type *A contains finite lists (sequences) of values from A. The value constructors for lists is <|-, ..., -|>. These lists are represented by enclosing its elements between <| and |>. So for example <|4, 0, 0, 1, 4, 5|> is a list containing six integers; it is of type *num. There is an empty list, denoted <||>, having the degenerate type *void. Unlike sets, the order is maintained

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

and no duplicates are removed. So <|1, 2, 3|>, <|2, 3, 1, |> and <|2, 2, 3, 1|> are all different.

Tuple types

The type A><B contains value pairs, the first from A the second from B. They are represented as <<a, b>>. Here <<_, _>> is the value constructor. So for example <<5, 3.14>> is a value from num><real and <<", {", 'a'}>> is from str><$str. The $ and * operators have priority over >< and ->, so a "set of pairs" type needs brackets, for example $(num><str).

Mapping types

The type A->B contains mappings: finite sets of value pairs. The values of this type can be constructed by combining the above two constructions. A restriction is imposed, though: the first components (from A) have to be different. So {<<1, 1>>, <<2, 1>>} is a value of num->num, but {<<1, 1>>, <<1, 2>>} is not. Note that the second value does have type $(num><num). The first value also has type $(num><num), since any value of num->num is also a value of $(num><num). We say that num->num is *stronger* than $(num><num).

The mapping operator can be used to model arrays: num->A can be used to represent an A array, with elements {<<1, a_1>>, <<2, a_2>>..., }. This is an alternative for the list *A operator. When sequential access only is needed, the list approach is preferred, but for random access the type num->A is more profitable. For combinations of the >< and -> operator, it is advised always to use brackets to indicate the order.

Record types

Record types are constructed by means of attribute identifiers. If l_1, l_2, ...are identifiers, then [l_1:A, l_2:A', ...] is a record type. Its values are for example [l_1:a, l_2:a', ...]. The type constructor and value constructor thus look the same in this case. This is like the record construction in programming languages. The order of the attributes is not important, for example [a:num, b:str] and [b:str, a:num] denote the same (record) type and the values [a:4, b:'a'] and [b:'a', a:4] denote the same record value within this type.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Type definitions

Type constructions can be given a name: a type definition. Names must obey the syntax for identifiers: they must begin with an alphabetic character and may not contain spaces. After its definition, a newly defined type can be used in type constructions, thus maybe creating more complex defined types. So the order in which type definitions are given is important. We show an example, where a client file is defined. The defined types are address and client (in that order), with the following constructions.

type	*name type from*
address	[street: str, city: str, zip: str]
client	[nr: num, name: str, addr: address]

The type $client can be used to model for example a store containing client data.

Exercises

1. Which of the following examples represent correct values. What is their type?

2.56.7	'abcd"	{7/5, '4'}
{false, true}	{'!@', 4}	{{}}
<<1, {7}>>	[a:1, b:{}]	<<<<4, ">>, '1'>>

2. Try to define ExSpect types for the following kinds of data. Give type definitions by choosing an appropriate name for them.

- a single-line error message;
- the measured volume, pressure and temperature of a gas;
- a multi-line ASCII document;
- the results of the games played in a soccer league.

3. Give the values in the types $bool and boo|><bool. Give the number of values in the types $(boo|><bool) and bool->bool.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

3.2 Lesson 2: Functions and terms

Functions

Terms are atoms (constants), variables or constructions out of these by means of functions and quantors. We first concentrate on functions. Functions transform input values into a result value. Function applications are terms involving the function symbol and simpler argument terms. For instance the term sin(3.14) signifies the application of the function sin (sinus) onto the term 3.14; the *evaluation* of this term results in a term of type real close to 0.0. The term 1+2 signifies the application of add (addition) onto 1 and 2; the evaluation result is 3. The term a+2 signifies the application of add (addition) onto a and 2; the evaluation result depends upon the value that is substituted for a.

Signature

Which input values of a function are accepted and which result value is given, is determined by the function's *signature*. The signature of sin is real for input and real for result, of add it is num, num for input and num for result We represent these signatures by real→real and num, num→num respectively.

Standard functions

ExSpect has a large number of standard functions. From them, new functions can be defined. We first describe some standard functions and their applications. Then we discuss quantors. Finally we describe how new functions can be defined.

Simple functions

A simple function has a single signature containing only types (without type variables). In the following table, we give a selection of simple functions with their signatures.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

name	*signature*	*example*
not	bool→bool	not(true) = false
and	bool, bool→bool	false and true = false
or	bool, bool→bool	false or true = true
cat	str, str→str	'Oh' cat ' boy' = 'Oh boy'
head	str→str	head('Gee') = 'G'
tail	str→str	tail('Gee') = 'ee'

The fourth line in this table defines the simple function cat, with two string parameters and a string result. From the example can be deduced that the function can be used in infix mode, i.e. the function name or symbol between the parameter terms. Its effect is to concatenate its two parameter strings. In chapter 5 is the exact nature of these functions is explained.

Every function with two parameters can be used in infix mode. However, in nested function applications, like (a and b) or c one is advised to use brackets for priority.

Overloading

The same function symbol and name is sometimes used for several simple functions. An example is formed by the arithmetical functions. The arithmetical manipulation of reals and nums is different. Yet, when specifying an addition, one wants to use the symbol +, irrespective of the kind of numbers added. This is done by *overloading*: attaching more signatures to the same function name, as shown in the table below.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

name	*signature*	*examples*
add	num, num→num	4/5 + 2/3 = 22/15
	real, real→real	0.8 + 0.67 ≈ 1.47
	num, $num→$num	4/5 + {0, 2/3} = {4/5, 22/15}
sub	num, num→num	4/5 - 2/3 = 2/15
	real, real→real	0.8 - 0.67 ≈ 0.13
mult	num, num→num	4/5 * 2/3 = 8/15
	real, real→real	0.8 * 0.67 ≈ 0.536
rdiv	num, num→num	(4/5)/(2/3) = 6/5
	real, real→real	0.8 / 0.67 ≈ 1.194
gt	num, num→bool	5 > 7 = false
gt	real, real→bool	5.> 7. = false

Note that the name of the above functions differ from the symbol used. Also note the third definition of addition that manipulates sets of numerals. A more extensive list of numeric functions can be found in the appendix.

The priority in nested arithmetical terms like a+b*c is the normal arithmetical priority. Note the extra brackets that are needed in the term (4/5)/(2/3). We encountered the division symbol earlier in the values belonging to type num.

Polymorphy

Polymorphy can be seen as infinite overloading. For instance set union does not regard the type of the contained elements. However, the union of sets of different type is illegal: the result cannot be typed. The signatures of the union thus contain $num, $num→$num, $str, $str→$str, but not $num, $str→??. Hence, $T, $T→$T is a signature for all types T. This is the meaning of the signature $T, $T→$T, where T is a *type variable*. Functions with signatures containing type variables are called *polymorphic*.

In the following table, we give some standard polymorphic functions. The type variables used are T and S. Even polymorphic functions can be overloaded, as in the last example.

name	*signature*	*examples*
eq	T, T→bool	'a' = 'a' = true
		{1} = {} = false
cond	bool, T, T→T	if true then 6 else 7 fi = 6
elt	T, $T→bool	2 elt {1, 3, 5} = false
pick	$T→T	pick({0}) = 0
union	$T, $T→$T	{1, 2} union {3} = {1, 2, 3}
pi1	T><S→T	pi1(<<3, 'a'>>) = 3
pi2	T><S→S	pi2(<<3, 'a'>>) = 'a'
dom	T->S→$T	dom({<<1, 5>>, <<2, 7>>}) = {1, 2}
rng	T->S→$S	rng({<<1, 5>>, <<2, 7>>}) = (5, 7}
apply	T->S, T→S	{<<1, 5>>, <<2, 7>}.2 = 7
	T->S, $T→$S	{<<1, 5>>, <<2, 7>>}.{1, 2} = {5, 7}
ins	T, *T→*T	4 ins <|4, 2|> = <|4, 4, 2|>
ins	T, $T→$	4 ins {2, 4} = {2, 4}
		4 ins {3, 2} = {2, 3, 4}
head	*T→T	head(<|4, 4, 2|>) = 4
	str→str	head('Gee') = 'G'
name	*signature*	*examples*
tail	*T→*T	tail(<|4, 4, 2|>) = <|4, 2|>
		str→str tail('Gee') = 'ee'
cat	*T, *T→*T	<|4, 4|> cat <|2|> = <|4, 4, 2|>
	str, str→str	'Oh' cat ' boy' = 'Oh boy'

Type constructors

Irregular functions are functions that have no finitely representable signature. The set, list and record constructors are irregular functions. They can be used with terms as in the following examples.

{1+1, 3-1, 5} = {2, 2, 5}
<|1+1, 3-1, 5|> = <|2, 2, 5|>
[name:'J.'cat ' Doe', sal:10∗10∗30] = [name:'J. Doe', sal: 3000]

The pair constructor (prod) is not irregular. It has signature T, S→T><S. Like the set and record constructors, it can be used with terms as in the following example. It has been included here because of its similarity with the previous two functions.

<<1.1∗1.1, 2+{5, 7}>> ≈ <<1.21, {7, 9}>>.

Also the record projection (symbol @) and update (upd) functions are irregular. The following examples illustrate their use.

[a:5, b:'q']@a = 5,
[a:5, b:'q'] upd [b:'r', c:true] = [a:5, b:'r', c:true]

These functions are called irregular, because their signature cannot be represented finitely, although the allowed parameter types and the way they affect the result type are completely determined. The set constructor accepts one or more parameters of type T and yields a result of type $T. The record constructors [l_1:_, l_2:_, ...] accept terms of types T_1, T_2... and yield a term of type [l_1:T_1, l_2:T_2...]. The list constructor accepts one or more parameters of type T and yield a result of type *T.

The projection functions _@l accept a parameter of any type of the form [...:T...] and yield a result of type $T. The record update function accepts two parameters of type [l_1:T_1] and [m_1:S_1] respectively, such that common labels are matched to the same types, i.e. if $l_1 = m_1$ then $T_1 = S_1$. The result type is the record type that joins the parameter types.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Erroneous function applications

In writing function application terms, three kinds of errors may occur. The first ones are syntax errors due to ill-matched brackets, misspellings and the like. The second are errors against the signature, for instance in the term {1, 2} union {5.0} or 5.0 = 5. Both kinds of errors are discovered during translation and reported by a "syntax error" or "type unknown" error message. The third are runtime errors, which are detected during execution only. Examples are given in the table below.

runtime error	*explanation*
1/0	division by zero
head('')	head/tail of empty string
pick({})	pick on empty set
{<<1, 2>>, <<3, 4>>}.2	invalid apply argument
[name:'Jan']@address	invalid apply argument

In these cases the "non-value" abort is the evaluation result. Of course nobody will write the above erroneous terms, but they may occur while evaluating a term involving variables. In applications of the cond function, the branch that is not chosen is not evaluated which could have aborted otherwise. So if a=0 then 0 else b/a fi and if a elt dom(M) then M.a else 0 fi are not aborting terms.

Exercises

1. Discuss the correctness of the following function application terms. Evaluate the correct ones.

```
if not 2=1 then 6-1 else 0
```

```
<|0, 4, 1|>} >-1
```

```
tail('a') elt {tail('b')}
```

```
if 6=7 then 6/0 else 5.1 fi
```

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

{0, 4+1, 5}

[a:head('a')]@b

[a:'', b:1/(1-1)] upd [c:1/2]

5.41+0.99=6.4

pi1(<<8-4, 1/0>>)

2. Do the following boolean terms always evaluate to true ? If not, give a counterexample. The variables a, b have type num, s has type str and A has type $num.

(head(s) cat tail(s)) = s

not(a elt A) or ({a} union A) = A

not(a elt {b})

a/2+a/2=a and a*1/a=1

Implicit mapping construction and quantors

We have constructed mappings by explicit enumeration. Mappings can also be implicitly constructed. The syntax is [x:A|E_x]. Here, x must be an identifier, A a term of set type, i.e. containing sets or mappings, and E_x a term that may contain x as parameter. If A is of type $B, the parameter x is understood to have type B. Also mappings have a set type, since they contain sets of pairs. B may even contain type variables when it occurs in a term with variables. The result type is B->C, where C is the type of E_x. The explicit value is obtained by constructing the set of pairs <<a,

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

b>>, where a subsequently takes all values from A and b is obtained by replacing x by a. Some examples of this construction are given below.

mapping	*type*	*explicit value*
[y:{'a', 'b', 'd'}|tail(y)]	str->str	{<<'a', ''>>, <<'b', ''>>, <<'d', ''>>}
[x:{1, 3}|[a:x-1]]	num->[a:num]	{<<1, [a:0]>>, <<3, [a:2]>>}
[z:{{}, {0}}|0 elt z]	$num->bool	{<<{}, false>>, <<{0}, true>>}

The implicit mapping construction can be combined with the rng function to construct sets in an implicit way. The term rng[x:A|E_x] is equivalent to the mathematical notation for sets. The above examples combined with rng give the following results.

term	*type*	*value*
rng[y:{'a', 'b', 'd'}|tail(y)]	$str	{''}
rng[x:{1, 2, 3}|[a:x-1]]	$[a:num]	{[a:0], [a:1], [a:2]}
rng[z:, 0 |0 elt z]	$num->bool	{false, true}

There are other functions (quantors) that combine well with implicit mappings. We give a table.

name	*signature*	*example*
all	T->bool→bool	all[x:{1, 2, 3}|x>0] = true
any	T->bool→bool	any[x:{1, 2, 3}|x>3] = false
{}	T->bool→$T	set[x:{1, 2, 3}|x>1] = {2, 3}
sum	T->num→num	sum[x:{1, 2, 3}|x+1] = 9
union	T->$S→$S	union[x:{1, 2, 3}|{x, x+1}] = {1, 2, 3, 4}

Note that union is an overloaded polymorphic function. The term set [x:A|E_x] can also be written $[x:A|$E_x$] because of the association of "set" with the $ symbol.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

We tabulate some terms with their mathematical counterpart. Here x is a variable, E_x a term containing x, P_x a predicate containing x, N_x a numerical term containing x, S_x a set term containing x and A a (finite) set. We added the max and min quantors that have the same signature as sum.

ExSpect	*math*
[x:A|E_x]	$\{(x, E_x)\|x \in A\}$
rng[x:A|P_x]	$\{P_x\|x \in A\}$
all[x:A|P_x]	$\forall_x \in A: P_x$
any[x:A|P_x]	$\exists_x \in A: P_x$
set[x:A|P_x]	$\{x\|x \in A \wedge P_x\}$
sum[x:A|N_x]	$\Sigma_{x \in A} N_x$
max[x:A|N_x]	$MAX_{x \in A} N_x$
min[x:A|N_x]	$MIN_{x \in A} N_x$
union[x:A|S_x]	$U_{x \in A} S_x$
rng[x:set[x:A|P_x]|E_x]	$\{E_x\| x \in A \wedge P_x\}$

Note the last element where two quantors are combined.

Function definitions

Like type definitions, function definitions are made by attaching a name to a term. A function definition must be accompanied by its signature, so its parameter variables and their types and its result type must accompany the body term. A term without parameters can be defined too, giving a value definition. In the following examples simple functions are defined.

```
name:         pi
parameters:
result type:  real
body:         3.14159

name:         triangle
parameters:   x:num
result type:  num
body:         x*(x-1)/2
```

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

name:	headstogether
parameters:	x:str, y:str
result type:	str
body:	head(x) cat head(y)

name:	mult
parameters:	x:$num, y:num
result type:	$num
body:	rng[t:x|t*y]

The last function in fact is an extra overloading of the mult function that we saw earlier. The choice of the name mult means that we can use the infix symbol *; for instance in the term {5, 4, 3} * 7 meaning mult({5, 4, 3}, 7) and yielding {35, 28, 21}.

Polymorphic functions

Polymorphic functions are defined by using type variables, like in the following examples: the set intersection and delete functions, followed by mapping update.

name:	isect
parameters:	x:$T, y:$T
result type:	$T
body:	rng[t:x|t elt y]

name:	del
parameters:	x:T, y:$T
result type:	$T
body:	set[t:y|not(t=x)]

name:	upd
parameters:	x:T->S, y:T->S
result type:	T->S
body:	[t:dom(x) union dom(y)| if t elt dom(y) then y.t else x.t fi]

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

The body of the mapping update function could also have been:

```
y union set[t:x| not(pi1(t) elt dom(y))].
```

This example needs some study, since many functions are combined. An example of the mapping update:

```
{<<1, 2>>, <<3, 4>>} upd {<<3, 5>>} yields

{<<1, 2>>, <<3, 5>>}.
```

The update of a mapping at a single point as in the above example is used frequently; an extra (overloaded) definition has been added as follows.

name:	upd
parameters:	x:T->S, y:T, z:S
result type:	T->S
body:	x upd{<<y, z>>}

Type casting

The above definition is not accepted by the translator, since it cannot perceive that the term{<<y, z>>} is a mapping. Instead, it is typed with $(T><S) and a function upd with signature T->S→$(T><S) is not found. So, an error message "type of upd unknown" is issued. This problem can be solved by a *type cast*: the type is reinforced to T->S by applying the "dummy" function tomap with the following definition.

name:	tomap
parameters:	x:$(T><S)
result type:	T->S
body:	x

The refused body of upd is replaced by x upd tomap({<<y, z>>}) and this is accepted.

Recursive definitions

Unlike type definitions, function definitions need not be defined before their use. This allows for recursive definitions that refer to themselves. As an example, we define the size of a set.

```
name:          size
parameters:    x: $T
result type:   num
body:          if x= {} then 0 else 1+size(rest(x)) fi
```

Recursive definitions must be used with care, since they may lead to non-termination. We give an example of a "dangerous" recursive definition.

```
name:          div
parameters:    x:num, y:num
result type:   num
body:          if x<y then x else 1+div(x-y, y) fi
```

Evaluating the function `2 div -1` will not terminate. As an exercise, try to improve upon this definition.

In the majority of cases, recursion can be replaced by quantor definitions. The body in `size` could have been `sum[t:x|1]`. Consider the function that extracts the addresses from a set of client records.

```
name:          addresses
parameters:    x:$[name:str, addr:str]
result type:   $str
body:          if x= {} then
                   {}
               else
                   {pick(x)@addr}union addresses(rest(x))
               fi
```

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Much more comprehensible, concise and efficient is the body

```
rng[t:x|t@addr].
```

Exercises

1. Define a function avg that computes the average of two numbers and also of a set of numbers. Make 4 overloaded definitions: for 2 nums, for $num, for 2 reals and for $real.

2. Define a function dommax of signature T->num→$T that selects from a mapping the set of points where the maximum is reached. So for example dommax[x:{1, 2, 3}|x*(x-4)] = {1, 3}.

3. Define a function torel of signature T->$S→$(T><S) that converts a set-valued mapping to the corresponding relation. If <<a, B>> elt f holds and b elt B, then also <<a, b>> elt torel(f) must hold and vice versa. Hint: combine the union and rng quantors.

Types and sets

ExSpect types and sets can mean the same, but are used in different context. The same holds for functions and mappings. A type construction means a certain set of values, that may be infinite. A set is a term that has a type (a set type, usually starting with a $ symbol) and is always finite. When a type is needed, a set is not accepted and vice versa.

A function has a signature and means a set of pairs that may be infinite. A function is always implicitly given by its parameters and a defining term. A mapping has a type (with a -> symbol in it) and is always finite. It may be explicitly or implicitly given. In the last case, a parameter, a set (term) and a defining term must be given.

Functions and mappings

The following expressions are incorrect terms for this reason. What is meant by the last non-term can be correctly formulated by the stretch function, giving the integers in a certain range.

expression	*term to use*
[x:bool|not(x)]	[x:{false, true}|not(x)]
all[x:num|not(x∗x < 0)]	true
set[x:num|5<x<13]	stretch(6, 12)

The following expressions are incorrect types.

expression	*type to use*
{false, true}	bool
{5, 6, 7}	num (cannot be strengthened)
set[x:num|5<x<13]	num (cannot be strengthened)

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

3.3 Lesson 3: The dynamic part

Introduction

The dynamic part of the language is about modelling processes and their interaction. The theory behind the dynamic part of ExSpect is Petri net theory. We will give a brief introduction into the aspects that are relevant for modelling and specification.

Petri nets

A Petri net is a network of active objects called *processors* and passive objects called *channels*. The channels may contain any number of *tokens*. Depending on the kind of channels involved, the tokens in it may be interpreted as units of information, control or even physical objects. A special channel called *store* contains a single token at all times.

Processors

Processors can be connected to channels in three ways: for input, output and both. This is represented graphically by arrowheads. If the tokens in the input channels of a processor satisfy certain conditions, the processor may become activated. It then consumes certain tokens from its input channels and produces tokens for its output channels. The production of tokens may be subject to a *delay*. Delayed tokens become available only when the simulation clock has advanced the same amount of time. Under which conditions a processor becomes activated and which tokens it may consume and produce is defined by the specification of the processor. How a processor is specified is defined in the next subsection.

Systems

A certain set of processors and channels can be grouped together in a subnet or *system*. Such a system can be used to build larger systems, by connecting some special channels or *pins* within the subsystem to the channels of the larger system. Connecting a processor or subnet into a larger net is called *installing*. The final model is a system without pins. This way of hierarchical modelling corresponds to the well-known and intuitively clear DFD (data flow diagram) modelling technique. In standard Petri net theory, processors are called *transitions* and channels are called *places*.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Processor definition

We now can explain how processors are defined and installed. Like a function, a processor has a name, parameters and a body. It also has a precondition. The parameters are divided into *input pin, output pin, store pin, value,* and *function parameters*. The parameters consist of a name and a type (for function parameters a signature). The precondition contains a predicate: a term of type bool. The body consists of a statement list. A statement is a conditional statement, the skip statement or an assignment. Commas separate the statements (,) instead of semicolons, indicating that they can be executed concurrently: their order has no significance. A conditional statement consists of an if-predicate, a then-part and an optional else-part.

The then- and else-parts are statement lists. An assignment consists of an output channel or store name, the assignment symbol (<-) and a term of the same type as the assigned channel or store. It may be followed by a delay term (of type real) preceded by the keyword delay. The terms and predicates may contain all variables amongst the parameters except the output channel parameters.

Processor installation

Processors are *installed* in systems. Upon installing, pins are linked to channels and pins of the system; terms are attached to value parameters and functions to function parameters. Their types must match of course. Terms attached to a value parameter may only contain value parameter names of the system as variables; these must be filled in when the system itself is installed in a higher-order system. The topmost system has no value or function parameters, nor pins. Value and function parameters are static: their values/definitions are bound to the variables when they are installed. In the topmost system they are bound to fixed values/functions. Pins are dynamic: the values they contain may vary during the execution of the topmost system.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Processor activation

A processor is *activated* when the channels linked to its input pins contain a token combination that satisfies the precondition. Upon activation, the activating tokens are consumed. Their values are bound to the corresponding parameter variables as are the store values. According to the specified statements, tokens are produced for the channels linked to the output pins, and the store values are updated. In a conditional statement, the then-part is executed if the condition predicate evaluates to true; the else-part is executed if it evaluates to false. A missing else-part means that the statement is skipped in this last case. The skip statement performs no actions. An assignment statement causes the creation of an output token (for output pins) or an update of the store value (for store pins).

The created token becomes available after d time units, where d is the corresponding value of the delay term. If the delay term is absent, d = 0. More assignments for the same output pin are allowed; in this case several tokens are produced for the same channel. More assignments for the same store pin are not allowed. Delayed assignments for store pins are not allowed either. As an example, we use the finance system defined earlier. The processor book has the following definition.

Connections:

```
input:   amount:  num
store:   total:   num
```

Pre:

```
amount > 0
```

Val / fun parameters:

```
kind:  val:          str
```

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Definition:

```
if kind = 'credit' then
    total <- total - amount
else
    total <- total + amount
fi
```

When it is installed with name cr, value parameter kind bound to 'credit', input pin amount bound to channel a containing 3 tokens with value 5, -3 and 7 respectively, store pin total bound to t containing the value 1000, the following can happen.

The tokens with values 5, 7 both satisfy the precondition, so any of them may activate cr. Suppose 5 is selected. The body is evaluated; because of the value of kind, the then-part is executed. So total is updated with 1000-5. In the new situation, a contains tokens with values -3, 7, and total contains 995. If the token with value 7 has not disappeared in the meantime, it can again activate cr and result in total containing 988. The token with value -3 does not satisfy the precondition; it will not be consumed by cr. For more activations of cr, other processors (or the end user) must insert positive-valued tokens in a.

Preconditions and if-statements

Precondition predicates determine whether a token combination is selected; the if-predicate determines what to do with the tokens once they are selected. So a processor without precondition (i.e. true as precondition) and body if P then S fi differs from the processor with precondition P and body S. The first is activated by any input token combination; if this combination does not satisfy S, it is consumed without causing any output or store update. The second does not consume token combinations that do not satisfy S, so they may be left for another round or for other processors. For token combinations that do satisfy S, both act the same.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Polymorphy

Like functions, processors can be polymorphic, i.e. their signature types may contain type variables. When a polymorphic processor is installed, the installed types must match the definition types. More explicitly, the same type variable T must correspond to the same installed type. Consider the following polymorphic processor copy.

```
input:    inpt: T
output:   outpt: T
val:      t: real
body:     outpt <- inpt delay t
```

This processor cannot be installed by connecting the input pin to a channel of type num and the output pin to a str channel. Nor can it be installed by producing for example a num term for the value parameter t.

Systems

When defining systems, parameters like in a processor definition can be specified. Systems can be polymorphic, like processors. Pins can be drawn in the graphical editor; value and function parameters must be added after opening the 'signature' window. For systems, even processor and (sub)system parameters are possible. This advanced feature is seldom used and outside the scope of this tutorial. Furthermore, channels and stores are defined and processors and subsystems installed.

Defining channels and stores is done by drawing them and then giving a name and type construction and (for channels optionally) an initialisation. A store initialisation is a term of the store type. It may contain variables from the value and function parameters. A channel initialisation may contain one or more terms of the channel type. It may also be absent. When installing subsystems, their parameters must be bound, like for processors.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Exercises

1. Convert the system FINANCE to a system with 2 different processors, one for addition and the other for subtraction.

2. Define a polymorphic processor with an input and output pin of type T and a store pin of type bool. Whenever the store has value true the input is copied to the output with a delay of 1 time unit. When the store has value false the input tokens are left untouched. This processor can be used to model a transistor or a lazy bureaucrat.

3. Define a subsystem with input pin of type $T and output pin of type T. If the set A arrives as input, the output must consist of all the individual elements of A.

Modules and scoping

ExSpect has a number of libraries containing general-purpose types, functions, processors and systems. It is possible to add one's own libraries. A library is called a "module" in the language; modules can be imported by "including" them. The order in which modules are included is important.

Exporting definitions

When creating a module, definitions that are to be used outside the module must be *exported*. Types can be exported with or without their defining type construction. In the last case, all possible access functions of the type must be defined in the module; i.e. the type is abstract. In this case, it is possible to implement the type and its access functions differently without the users of the type noticing it. In the other case this is impossible. Also definitions that are accessible by users must be exported.

Local definitions

Modules are one way of limiting the scope of definitions. Another way is by *local definitions* that may accompany a single definition. The local definitions are not known outside the definition that they accompany. Local definitions may be used for creating more efficient definitions, when recursion is involved.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

Consider the following definition of the Fibonacci function.

name:	fib
parameters:	x:num
result type:	num
body:	if x<2 then 1 else fib(x-1) + fib(x-2) fi

Evaluating for example fib(5) means going through the following steps.

fib(5)→
fib(4)+fib(3)→
fib(3)+fib(2)+fib(2)+fib(1)→
fib(2)+fib(1)+fib(1)+fib(0)+fib(1)+fib(0)+1→
fib(1)+fib(0)+1+1+1+1+1+1→
1+1+6→
8

We see that in the above process, fib(3) is evaluated twice and fib(2) three times. When evaluating fib for higher values, this gets worse and worse, thus causing bad performance.

It would be less inefficient to evaluate fib(5) as follows:

fib(5)→
1∗fib(5)+0∗fib(4)→
1∗fib(4)+1∗fib(3)→
2∗fib(3)+1∗fib(2)→
3∗fib(2)+2∗fib(1)→
5∗fib(1)+3∗fib(0)→
8.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

By creating a subordinate function with more parameters, this can be achieved. The body of fib becomes fib2(x, 1, 0), where fib2 with num parameters x, y, z is defined with the body:

```
if x<2 then y+z else fib2(x-1, y+z, y) fi
```

The function fib2 has no use except in the context of the fib function, so it is natural to define it as local function of fib. Local functions can be nested, but it is not advised to use this feature. It is not pleasant to hunt for definitions that are too deeply nested. It is better to put definitions on the top level, unless they are specifically linked to one definition.

Scope

We conclude this lesson by treating the scope of variables. The smallest and strongest scope is the variable in an implicit mapping term. In the term [x:A|E], any variable x within term E is bound by the mapping, and thus is interpreted as an element of the set A. An occurrence of x in A is not bound by it, so it must occur in some wider scope. An example is [x: stretch(0, x)| x+1] in a function body where x is a parameter. Here, the x in x+1 is the mapping variable, whereas the one in stretch(0, x) is the function parameter.

Parameters and local definitions in a definition form the second scope. The variables there take priority over those in the third scope, the global and imported definition names.

Exercises

Define a function freqcount of signature str, str→num, such that freqcount(a, b) gives the number of times that the string b occurs as substring of a. For example, freqcount('abracradrabra', 'ra') = 4. Compute how many recursion steps are needed if the length of a is n and of b is m. Improve the efficiency of your definition (the most efficient definition requires n steps).

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

4. Syntax

Release note

In ExSpect version 6.x the following constructions are not supported anymore by the designer and animator:

- Bundle types;
- Processor and system parameters.

The type checker has however not been changed since version 5, so the type checker still supports these constructions. That is why the keyword ctype still needs to be listed as reserved word.

Reserved words

as	bounds	channel	ctype
delay	default	end	export
form	from	fun	in
include	inhibit	init	module
out	pre	prio	proc
skip	sys	store	type
val	void	where	with
if	then	else	fi
elif			

Notation

The syntax is described in BNF (Beckes Nauer form). The following special notations are used:

< X >	A comma separated list of X's
^X	Anything except an X

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

```
module    ::=  ( line ';'  )+
line      ::=  [ 'export' ] dec 'in' id                                  |
               [ 'export' ] def [ 'where' line ( ';' line )* 'end' ]     |
               'include' string
dec       ::=  typedec | fundec | procdec | sysdec
typedec   ::=  'type' id [ ( ':=' | 'from' ) type ]
fundec    ::=  id [ [ par ] ] : type
procdec   ::=  'proc' id '[' < afpar > ']'
sysdec    ::=  'sys' id '[' < afpar > ']'
par       ::=  < id ':' type >
actpar    ::=  ( 'in' | 'out' | 'store' | 'val' ) par
funpar    ::=  'fun' < fundec [ 'default' id ] >
afpar     ::=  actpar | funpar
def       ::=  typedef | fundef | procdef | sysdef
typedef   ::=  [ 'export' ] 'type' id ( ':=' | 'from' ) type
fundef    ::=  id [ '[' par ']' ] ':=' expr ':' type
procdef   ::=  'proc' id '[' < afpar > [ '|' preprio ] ']' ':=' < stat >
sysdef    ::=  'sys' id [ '[' < afpar > ']' ] ':=' < obj >
preprio   ::=  'pre' expr                    |
               'prio' expr                   |
               'pre' expr ''' 'prio' expr    |
               'prio' expr ''' 'pre' expr
stat      ::=  id '<-' expr [ 'delay' expr ] [ 'bounds' real [ real ] ]  |
               'if' expr 'then'
                    < stat >
```

```
                ( ‘elif’ expr ‘then’ < stat > )*
                [ ‘else’ < stat > ]
                ‘fi’                                                    |
                ‘skip’
obj     ::=     ( ‘channel’ | ‘store’ ) id ‘:’ type [ ‘in’ id ]
                        [ ‘form’ string ] ( ‘init’ expr )*              |
                id [ ‘:’ id ] ‘(‘ < avarg > ‘)’
actarg  ::=     ( ‘in’ | ‘out’ | ‘store’ | ‘inhibit’ | ‘fun’ ) < id >
valarg  ::=     ‘val’ < expr >
avarg   ::=     actarg | valarg
expr    ::=     id                                                      |
                const                                                   |
                ‘(‘ expr ‘)’                                            |
                [ id ] ‘[‘ id ‘:’ expr ‘|’ expr ‘]’                     |
                ‘[‘ < id ‘:’ expr > ‘]’                                 |
                expr ‘@’ id                                             |
                id ‘(‘ arg ‘)’                                          |
                ‘if’ expr ‘then’ expr ( ‘elif’ expr ‘then’ expr )* ‘else’ expr ‘fi’ |
                ‘<<‘ < expr > ‘>>‘                                      |
                ‘{‘ < expr > ‘}’                                        |
                ‘<|’ < expr > ‘|>‘                                      |
                expr ( id | ‘=‘ | ‘-’ | ‘/’ | ‘!=‘ | ‘->‘ | ‘+’ | ‘*’
                        | ‘>‘ | ‘<‘ | ‘>=‘ | ‘<=‘ | ‘\’ | ‘.’ ) expr  |
                ( ‘-’ | ‘$’ | ‘#’ ) expr
type    ::=     stype | type ‘->‘ stype | type ‘><‘ stype
```

```
stype     ::=  id | '(' type ')'          |
               ( '$' | '*' ) stype        |
               '[' < id ':' type > ']'
const     ::=  num          |
               real         |
               string       |
               '{}'         |
               '<||>'
num       ::=  ( '0' - '9' )+
real      ::=  ( '0' - '9' )+ '.' ( '0' - '9' )* [ 'e' [ '+' | '-' ] ( '0' - '9' )+ ]
string    ::=  "'" ( "'" | ^"'" )* "'"
id        ::=  ( 'a' - 'z' | 'A' - 'Z' ) ( 'a' - 'z' | 'A' - 'Z' | '0' - '9' | '_' )*
```

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

5. Function list

Here all functions defined in the modules basic utils, stat and adt are listed alphabetically. For every function the module is given in which the function is defined, and a short hint as to what the result will be of an invocation of that particular function.

abort: void;	(basic) value to denote abortion
abs[x:num]: num;	(utils) | x | (absolute value)
add[x:num, y:num]: num;	(basic) x + y
add[x:$num, y:num]: $num;	(utils) { z + y | z ∈ x }
add[x:real, y:real]: real;	(basic) x + y
add[x:$real, y:real]: $real;	(utils) { z + y | z ∈ x }
add[n:num, x:real]: real;	(utils) real(n) + x
add[x:real, n:num]: real;	(utils) x + n
add[x:*num, y:num]: *num;	(utils) x with each member incremented by y
add[x:*real, y:real]: *real;	(utils) x with each member incremented by y
adel [x:num->T, y:num]: num->T;	(adt) array x without the y'th member
aempty [x:num->T]: bool;	(adt) is x an empty array?
aelement [x:num->T, y:num]: T;	(adt) x(y), value of the y'th member of array x
aindex [x:num->T, y:T]: $num;	(adt) {i:dom(x) | x(i)=y}
ains [x:num->T, y:num, z:T]: num->T;	(adt) array x with (y,z) inserted
ains [x:num->T, y:num, z:num->T]: num->T;	(adt) array x with array z inserted at position y
all[x:T->bool]: bool;	(utils) forall {y ∈ dom(x)} x(y)
and[x:bool, y:bool]: bool;	(utils) x and y (logical and)
anull: num->void;	(adt) the empty array
any[x:T->bool]: bool;	(utils) exists {y ∈ dom(x)} x(y)
apply[x:T->S, y:T]: S;	(basic) x(y) (mapping application)
apply[x:T->S, y:$T]: $S;	(utils) { x(z) | z ∈ y }
asplit [x:num->T, y:num]: (num->T)><(num->T);	(adt) split array x at index y
aswap [x:num->T, y:num, z:num]: num->T;	(adt) array x with values at y and z interchanged
atobag [x:num->T]: T->num;	(adt) array x converted to a bag

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

badd [x:T->num, y:$T]: T->num;	(adt) bag x with all elements of y added
badd [x:$$T]: T->num;	(adt) bag of all elements occurring in the sets in x
bdel [x:T->num, y:T]: T->num;	(adt) bag x with one element y deleted
bdiff [x:T->num, y:T->num]: T->num;	(adt) bag x minus all elements of bag y
bempty [x:T->num]: bool;	(adt) is x an empty bag?
bernouilli[p:real, seed:real]: real;	(stat) a random number (bernouilli)
binomial[n:real, p:real, seed:real]: real;	(stat) a random number (binomial)
bins [x:T->num, y:T]: T->num;	(adt) bag x with element y inserted
bisect [x:T->num, y:T->num]: T->num;	(adt) biggest bag both contained in x and y (bag intersection)
bjoin [x:T->num, y:T->num]: T->num;	(adt) bag of all elements of x and y (join)
bjoin [x:$(T->num)]: T->num;	(adt) bag of all elements of all bags in x (join)
bmax [x:num->num]: num;	(adt) maximum (greatest element) of bag x
bmin [x:num->num]: num;	(adt) minimum (smallest element) of bag x
bnull: void->num;	(adt) the empty bag
bocc [x:T->num, y:T]: num;	(adt) number of times y occurs in bag x
botint[x:num]: num;	(utils) floor of x , largest integer <= x
botint[x:real]: num;	(basic) floor of x , largest integer <= x
bpick [x:T->num]: T;	(adt) an arbitrary (though deterministic) element from bag x
bproj [x:T->num, y:$T]: T->num;	(adt) projection of bag x onto y
brest [x:T->num]: T->num;	(adt) bag x without the element bpick(x)
bsize [x:T->num]: num;	(adt) number of elements in bag x
bsum [x:num->num]: num;	(adt) sum of all elements of bag x
btoset [x:T->num]: $T;	(adt) bag x converted to a set
bunion [x:T->num, y:T->num]: T->num;	(adt) smallest bag containing x and y (bag union)
bunion [x:$(T->num)]: T->num;	(adt) smallest bag containing all bags of x (bag union)
cat[x:str, y:str]: str;	(basic) x concatenated with y
cat[x:*T, y:*T]: *T;	(basic) the concatenation of x and y
chisq[n:real, seed:real]: real;	(stat) a random number (X-squared)
chop[x:str, n:num]: str;	(utils) string x reduced to its first n characters

cond[x:bool, y:T, z:T]: T;	(basic) if x then y else z (condition)
cos[x:real]: real;	(basic) (x)
del[x:T, y:$T]: $T;	(basic) y \ { x }, x is deleted from the set y
denominator[x:num]: num;	(basic) the denominator of a rational number
div[x:num, y:num]: num;	(basic) x div y (truncated)
dom[x:T->S]: $T;	(utils) domain of x
elt[x:T, y:$T]: bool;	(basic) x y
elt[x:T, y:*T]: bool;	(utils) x occurs in y
eq[x:T, y:T]: bool;	(basic) x = y
erlang[m:real, k:real, seed:real]: real;	(stat) a random number (erlang)
even[n:num]: bool;	(utils) is n even?
exp[x:real]: real;	(basic) e^x
false: bool;	(basic) falsehood
fcomp[x:T->S, y:S->R]: T->R;	(utils) x o y, 'function' composition
frc[x:num]: num;	(utils) fractional part of x
frc[x:real]: real;	(utils) fractional part of x
frest[x:S->T]: S->T;	(utils) x \ {pick(x)}, x without the element pick(x)
front[x:*T]: *T;	(utils) x with the last member removed
gamma[l:real, k: real, seed:real]	(stat) draw from a gamma distribution with mean l/k and variance l/k^2.
gcd[x:num, y:num]: num;	(basic) x y, greatest common divisor
ge[x:num, y:num]: bool;	(utils) x >= y
ge[x:real, y:real]: bool;	(utils) x >= y
gt[x:num, y:num]: bool;	(utils) x > y
gt[x:real, y:real]: bool;	(utils) x > y
head[x:str]: str;	(basic) first char in string x
head[x:*T]: T;	(basic) the first member of x
iff[x:bool, y:bool]: bool;	(utils) x == y (logical equivalence)
impl[x:bool, y:bool]: bool;	(utils) x -> y (logical implication)
ins[x:T, y:$T]: $T;	(basic) y union { x } (insert x into y)
ins[x:T, y:*T]: *T;	(basic) y with x added as head
inv[x:T->S, y: S]: $T;	(basic) { z | z ∈ dom(x) and x(z) = y } (inverse)

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

inv[x:T->S, y: $S]: $T;	(utils) { z | z ∈ dom(x) and x(z) y } (inverse)
isect[x: $T, y: $T]: $T;	(basic) x intersect y (intersection)
isint[x:num]: bool;	(utils) is x an integer?
isint[x:str]: bool;	(utils) is x an integer?
isnum[x:str]: bool;	(basic) is x a num?
isreal[x:str]: bool;	(basic) is x a real?
last [x:*T]: *T;	(utils) the last member of x
lcat [x:num->T, y:num->T]: num->T;	(adt) concatenation of lists x and y
lcons [x:num->T, y:T]: num->T;	(adt) append y in front of list x
le[x:num, y:num]: bool;	(utils) x <= y
le[x:real, y:real]: bool;	(utils) x <= y
lempty [x:num->T]: bool;	(adt) is x an empty list?
lfront [x:num->T]: num->T;	(adt) list x without the last member
lhead [x:num->T]: T;	(adt) first member of list x
list [x:T]: *T;	(basic) a list containing all members of x
llast [x:num->T]: T;	(adt) last member of list x
ln[x:real]: real;	(basic) (x), natural logarithm
lnull: num->void;	(adt) the empty list
lreverse [x:num->T]: num->T;	(adt) reverse of list x
lsnoc [x:num->T, y:T]: num->T;	(adt) append y at the back of list x
lt[x:num, y:num]: bool;	(utils) x < y
lt[x:real, y:real]: bool;	(utils) x < y
ltail [x:num->T]: num->T;	(adt) list x without the first member
ltobag [x:num->T]: T->num;	(adt) list x converted to a bag
match[x:str, y:str]: bool;	(utils) is x head of y?
max[x: $num]: num;	(utils) z : z ∈ x and forall {y ∈ x} z >= y, maximum of set x
max[x:num, y:num]: num;	(utils) x max y
max[x:T->num]: num;	(utils) max(rng(x))
max[x: $real]: real;	(utils) z : z ∈ x and forall {y ∈ x} z >= y, maximum of set x
max[x:real, y:real]: real;	(utils) x max y

max[x:T->real]: real;	(utils) max(rng(x))
max[x:*num]: num;	(utils) the maximum in x
max[x:*real]: real;	(utils) the maximum in x
minus[x:num]: num;	(utils) -x
minus[x:real]: real;	(utils) -x
min[$x:num]: num;	(utils) z : z ∈ x and forall {y ∈ x} z <= y, minimum of set x
min[x:num, y:num]: num;	(utils) x min y
min[x:T->num]: num;	(utils) min(rng(x))
min[x: $real]: real;	(utils) z : z ∈ x and forall {y ∈ x} z <= y, minimum of set x
min[x:real, y:real]: real;	(utils) x min y
min[x:T->real]: real;	(utils) min(rng(x))
min[x:*num]: num;	(utils) the minimum in x
min[x:*real]: real;	(utils) the minimum in x
mod[x:num, y:num]: num;	(basic) x mod y
mod[x: $num, y:num]: num;	(utils) { z mod y | z ∈ x }
mod[x:*num, y:num]: *num;	(utils) x with each member replaced by its value module y
mult[x:num, y:num]: num;	(basic) x * y
mult[x: $num, y:num]: $num;	(utils) { z * y | z ∈ x }
mult[x:real, y:real]: real;	(basic) x * y
mult[x:real, y:real]: real;	(utils) { z * y | z ∈ x }
mult[n:num, x:real]: real;	(utils) real(n) * x
mult[x:real, n:num]: real;	(utils) n * x
mult[x:*num, y:num]: *num;	(utils) each member of x multiplied by y
mult[x:*real, y:real]: *real;	(utils) each member of x multiplied by y
ne[x:T, y:T]: bool;	(utils) x /= y, x not equal to y
nexp[m:real, seed:real]: real;	(stat) a random number (negative exponential)
normal[m:real, v:real, seed:real]: real;	(stat) a random number (normal)
not[x:bool]: bool;	(utils) not x (logical not)

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

num[x:real]: num;	(utils) real x converted to num with a certain precision
num[x:real, y:num]: num;	(basic) real x converted to num with precision y
num[x:str]: num;	(basic) str x converted to num
numerator[x:num]: num;	(basic) the numerator of a rational number
odd[n:num]: bool;	(utils) is n odd?
or[x:bool, y:bool]: bool;	(utils) x or y (logical or)
parse[y:str, symbols:str]: num -> str;	(utils) divides the string y in a list of substrings e.g. words that are separated by a delimiter in symbols
parse[y:str]: num -> str;	(utils) divides the string y in a list of substrings e.g. words that are separated by a space or by the characters `, : ; " ! "
PERT_beta [l:real, u:real, e: real, seed:real]	(stat) draw from a beta distribution with minimum l, maximum u and expectation e.
pi1[x:T><S]: T;	(basic) first element of x
pi1[x: $(T><S)]: $T;	(utils) { y | y ∈ T and exists {z ∈ S} (y >< z) ∈ x }
pi2[x:T><S]: S;	(basic) second element of x
pi2[x: $(T><S)]: $S;	(utils) { z | z ∈ S and exists {y ∈ T} (y >< z) ∈ x }
pick[x: $T]: T;	(basic) arbitrary (though deterministic) element from x
poisson[m:real, seed:real]: real;	(stat) a random number (poisson)
pos[x:num]: bool;	(basic) x >= 0
pos[x:real]: bool;	(utils) x >= 0
pow[x:num, y:num]: num;	(utils) x^y
pow[x:real, y:real]: real;	(utils) x^y
prod[x:T, y:S]: T><S;	(basic) << x , y >> (pair of x and y)
random[seed:real]: real;	(stat) a random number (between 0 and 1)
rdiv[x:num, y:num]: num;	(basic) x / y
rdiv[x: $num, y:num]: num;	(utils) { z / y | z ∈ x }
rdiv[x:real, y:real]: real;	(basic) x / y
rdiv[x: $real, y:real]: real;	(utils) { z / y | z ∈ x }
rdiv[n:num, x:real]: real;	(utils) real(n) / x

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

rdiv[x:real, n:num]: real;	(utils) x / real(n)
rdiv[x:*num, y:num]: *num;	(utils) each member of x divided by y
rdiv[x:*real, y:real]: *real;	(utils) each member of x divided by y
real[x:num]: real;	(basic) num x converted to real
real[x:str]: real;	(basic) str x converted to real
rest[x: $T]: $T;	(basic) x \ {pick(x) }
restrict[x:T->S, y: $T]: T->S;	(utils) x |` y (restricted to)
rev[x:str]: str;	(utils) string x in reverse order
reverse[x:*T]: *T;	(utils) list x in reverse order
rint[a:real, b:real, seed:real]: real;	(stat) a random number (between a and b)
rng[x:T->S]: $S;	(utils) range of x
scan[x:str, y:str]: bool;	(utils) does x occur in y?
scanrest[x:str, y:str]: str;	(utils) tail of y, from first occurrence of x in y
sdiff[x: $T, y: $T]: $T;	(basic) x \ y
set[x:T->bool]: $T;	(basic) { y | y ∈ T and x(y) }
set[x:*T]: T;	(basic) the set of members of x
sin[x:real]: real;	(basic) (x)
size[x: $T]: num;	(basic) sum {y ∈ x} 1, number of elements of x
size[x:*T]: num;	(utils) number of members of x
sqrt[x:real]: real;	(basic) square root of {x}
stobag [x: $T]: T->num;	(adt) set x converted to a bag
str[x:num]: str;	(basic) x converted to string
str[x:real]: str;	(basic) x converted to string
strncmp[x:str, y:str, n:num]: bool;	(utils) are the first n characters of strings x and y the same?
stretch[x:num, y:num]: *num;	(utils) list containing all members of { z | x <= z <= y }
stretch[x:num, y:num]: $num;	(utils) { z | x <= z <= y }
strlen[x:str]: num;	(utils) length of string x
student[n:real, seed:real]: real;	(stat) a random number (student)
sub[x:num, y:num]: num;	(basic) x - y
sub[x: $num, y:num]: $num;	(utils) { z - y | z ∈ x }

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

sub[x:real, y:real]: real;	(basic) x - y
sub[x: $real, y:real]: $real;	(utils) { z - y | z ∈ x }
sub[x:*num, y:num]: *num;	(utils) each member of x subtracted by y
sub[x:*real, y:real]: *real;	(utils) each member of x subtracted by y
sub[n:num, x:real]: real;	(utils) real(n) - x
sub[x:real, n:num]: real;	(utils) x - real(n)
subbag [x:T->num, y:T->num]: bool;	(adt) is bag x contained in y? (bag subset)
subset[x: $T, y: $T]: bool;	(utils) x <= y
sum[x: $num]: num;	(utils) sum {y ∈ x} y
sum[x:T->num]: num;	(utils) sum {y ∈ dom(x)} x(y)
sum[x: $real]: real;	(utils) sum {y ∈ x} y
sum[x:T->real]: real;	(utils) sum {y ∈ dom(x)} x(y)
sum[x:*num]: num;	(utils) the sum of all numbers in x
sum[x:*real]: real;	(utils) the sum of all numbers in x
tail[x:str]: str;	(basic) string x without first character
tail[x:*T]: *T;	(basic) the tail of x
that[x:T->bool]: T;	(utils) z : z ∈ dom(x) and x(z) and forall {y ∈ dom(x) and x(y)} y = z the only element in T that is true
tomap[x:(T><S)]: T->S;	(utils) type casting of set of pairs to mapping
topint[x:num]: num;	(utils) smallest integer >= x
tovoid[x:T]: void;	(utils) x converted to void
true: bool;	(basic) truth
uniform[a:real, b:real, seed:real]: real;	(stat) a random number (uniform)
union[x: $$T]: $T;	(utils) Union_{y ∈ x} y, union of all elements of x
union[x: $T, y: $T]: $T;	(basic) x union y
upd[x:T->S, y:T, z:S]: T->S;	(basic) update x with x(y) = z
upd[x:[], y:[]]: [];	(basic) update x according to y

5.1 Boolean functions

This section shows all boolean functions of the modules basic and utils.

all[x:T->bool]: bool;	(utils) forall {y ∈ dom(x)} x(y)
and[x:bool, y:bool]: bool;	(utils) x and y (logical and)
any[x:T->bool]: bool;	(utils) exists {y ∈ dom(x)} x(y)
cond[x:bool, y:T, z:T]: T;	(basic) if x then y else z (condition)
elt[x:T, y: $T]: bool;	(basic) x ∈ y
elt[x:T, y:*T]: bool;	(basic) x occurs in y
eq[x:T, y:T]: bool;	(basic) x = y
even[n:num]: bool;	(utils) is n even?
false: bool;	(basic) falsehood
ge[x:num, y:num]: bool;	(utils) x >= y
ge[x:real, y:real]: bool;	(utils) x >= y
gt[x:num, y:num]: bool;	(utils) x > y
gt[x:real, y:real]: bool;	(utils) x > y
iff[x:bool, y:bool]: bool;	(utils) x == y (logical equivalence)
impl[x:bool, y:bool]: bool;	(utils) x -> y (logical implication)
isint[x:num]: bool;	(utils) is x an integer?
le[x:num, y:num]: bool;	(utils) x <= y
le[x:real, y:real]: bool;	(utils) x <= y
lt[x:num, y:num]: bool;	(utils) x < y
lt[x:real, y:real]: bool;	(utils) x < y
match[x:str, y:str]: bool;	(utils) is x head of y?
ne[x:T, y:T]: bool;	(utils) x /= y, x not equal to y
not[x:bool]: bool;	(utils) not x (logical not)
odd[n:num]: bool;	(utils) is n odd?
or[x:bool, y:bool]: bool;	(utils) x or y (logical or)
pos[x:num]: bool;	(basic) x >= 0
pos[x:real]: bool;	(utils) x >= 0
scan[x:str, y:str]: bool;	(utils) does x occur in y?
subset[x: $T, y: $T]: bool;	(utils) x <= y
true: bool;	(basic) truth

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

5.2 Numerical functions

This section shows all functions with numericals (integers) as a result

type or as an argument in the modules basic and utils.

abs[x:num]: num;	(utils) | x | (absolute value)
add[x:num, y:num]: num;	(basic) x + y
add[x: $num, y:num]: num;	(utils) { z + y | z x }
add[n:num, x:real]: real;	(utils) real(n) + x
add[x:real, n:num]: real;	(utils) x + n
add[x:*num, y:num]: *num;	(utils) y added to each element of x
botint[x:num]: num;	(utils) floor of x , largest integer less than/ equal to x
denominator[x:num]: num;	(basic) the denominator of a rational number
div[x:num, y:num]: num;	(basic) x div y (truncated)
even[n:num]: bool;	(utils) is n even?
frc[x:num]: num;	(utils) fractional part of x
gcd[x:num, y:num]: num;	(basic) x y, greatest common divisor
ge[x:num, y:num]: bool;	(utils) x >= y
gt[x:num, y:num]: bool;	(utils) x > y
isint[x:num]: bool;	(utils) is x an integer?
le[x:num, y:num]: bool;	(utils) x <= y
lt[x:num, y:num]: bool;	(utils) x < y
max[x: $num]: num;	(utils) z : z ∈ x and forall {y ∈ x} z >= y, maximum of set x
max[x:num, y:num]: num;	(utils) x max y
max[x:T->num]: num;	(utils) max(rng(x))
max[x:*num]: num;	(utils) the maximum of x
min[x: $num]: num;	(utils) z : z ∈ x and forall {y ∈ x} z <= y, minimum of set x
min[x:num, y:num]: num;	(utils) x min y
min[x:T->num]: num;	(utils) min(rng(x))

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

min[x:*num]: num;	(utils) the minimum of x
minus[x:num]: num;	(utils) -x
mod[x:num, y:num]: num;	(basic) x mod y
mod[x: $num, y:num]: num;	(utils) { z mod y | z ∈ x }
mod[x:*num, y:num]: *num;	(utils) each element of x replaced by its value modulo y
mult[x:num, y:num]: num;	(basic) x * y
mult[x: $num, y:num]: num;	(utils) { z * y | z ∈ x }
mult[n:num, x:real]: real;	(utils) real(n) * x
mult[x:real, n:num]: real;	(utils) n * x
mult[x:*num, n:num]: *num;	(utils) each element of x multiplied by y
num[x:real]: num;	(utils) real x converted to num with a certain precision
num[x:real, y:num]: num;	(basic) real x converted to num with precision y
num[x:str]: num;	(basic) str x converted to num
numerator[x:num]: num;	(basic) the numerator of a rational number
odd[n:num]: bool;	(utils) is n odd?
pos[x:num]: bool;	(basic) x >= 0
pow[x:num, y:num]: num;	(utils) x^y
rdiv[x:num, y:num]: num;	(basic) x / y
rdiv[x: $num, y:num]: num;	(utils) { z / y | z ∈ x }
rdiv[n:num, x:real]: real;	(utils) real(n) / x
rdiv[x:real, n:num]: real;	(utils) x / real(n)
rdiv[x:*num, n:num]: *num;	(utils) each element of x divided by y
real[x:num]: real;	(basic) num x converted to real
size[x:*T]: num;	(utils) number of elements in x
stretch[x:num, y:num]: *num;	(utils) list containing all elements of { x ... y }
stretch[x:num, y:num]: $num;	(utils) { z | x <= z <= y }
sub[x:num, y:num]: num;	(basic) x - y
sub[x: $num, y:num]: $num;	(utils) { z - y | z ∈ x }
sub[x:*num, y:num]: *num;	(utils) each element of x subtracted with y
sum[x: $num]: num;	(utils) sum {y ∈ x} y

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

sum[x:T->num]: num;	(utils) sum {y ∈ dom(x)} x(y)
sum[x:*num]: num;	(utils) the sum of all members of x

5.3 Real functions

This section shows all functions with reals as a result type or as an argument in the modules basic and utils.

add[x:real, y:real]: real;	(basic) x + y
add[x: $real, y:real]: $real;	(utils) { z + y | z ∈ x }
add[n:num, x:real]: real;	(utils) real(n) + x
add[x:real, n:num]: real;	(utils) x + n
add[x:*real, y:real]: *real;	(utils) each element of x incremented by y
botint[x:real]: num;	(basic) floor of x , largest integer <= x
cos[x:real]: real;	(basic) (x)
exp[x:real]: real;	(basic) e^x
frc[x:real]: real;	(utils) fractional part of x
ge[x:real, y:real]: bool;	(utils) x >= y
gt[x:real, y:real]: bool;	(utils) x > y
le[x:real, y:real]: bool;	(utils) x <= y
ln[x:real]: real;	(basic) (x)
lt[x:real, y:real]: bool;	(utils) x < y
max[x: $real]: real;	(utils) z : z ∈ x and forall {y ∈ x} z >= y, maximum of set x
max[x:real, y:real]: real;	(utils) x max y
max[x:T->real]: real;	(utils) max(rng(x))
max[x:*real]: real;	(utils) the maximum of x
minus[x:real]: real;	(utils) -x
min[x: $real]: real;	(utils) z : z ∈ x and forall {y ∈ x} z <= y, minimum of set x
min[x:real, y:real]: real;	(utils) x min y

min[x:T->real]: real;	(utils) min(rng(x))
min[x:*real]: real;	(utils) the minimum of x
mult[x:real, y:real]: real;	(basic) x * y
mult[x: $real, y:real]: real;	(utils) { z * y | z ∈ x }
mult[n:num, x:real]: real;	(utils) real(n) * x
mult[x:real, n:num]: real;	(utils) n * x
mult[x:*real, y:real]: *real;	(utils) each element of x multiplied by y
num[x:real]: num;	(utils) real x converted to num
pos[x:real]: bool;	(utils) x >= 0
pow[x:real, y:real]: real;	(utils) x^y
rdiv[x:real, y:real]: real;	(basic) x / y
rdiv[x: $real, y:real]: real;	(utils) { z / y | z ∈ x }
rdiv[n:num, x:real]: real;	(utils) real(n) / x
rdiv[x:real, n:num]: real;	(utils) x / real(n)
rdiv[x:*real, y:real]: *real;	(utils) each member of x divided by y
real[x:num]: real;	(basic) num x converted to real
real[x:str]: real;	(basic) str x converted to real
sin[x:real]: real;	(basic) (x)
sqrt[x:real]: real;	(basic) square root of {x}
str[x:real]: str;	(basic) x converted to string
sub[x:real, y:real]: real;	(basic) x - y
sub[x: $real, y:real]: real;	(utils) { z - y | z ∈ x }
sub[n:num, x:real]: real;	(utils) real(n) - x
sub[x:real, n:num]: real;	(utils) x - real(n)
sub[x:*real, y:real]: *real;	(utils) each member of x subtracted by y
sum[x: $real]: real;	(utils) sum {y ∈ x} y
sum[x:T->real]: real;	(utils) sum {y ∈ dom(x)} x(y)
sum[x:*real]: real;	(utils) the sum of all members of x

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

5.4 String functions

This section shows all string functions of the modules basic and utils.

cat[x:str, y:str]: str;	(basic) x concatenated with y
chop[x:str, n:num]: str;	(utils) string x reduced to its first n characters
head[x:str]: str;	(basic) first char in string x
isint[x:str]: bool;	(utils) is x an integer?
isnum[x:str]: bool;	(basic) is x a num?
isreal[x:str]: bool;	(basic) is x a real?
match[x:str, y:str]: bool;	(utils) is x head of y?
num[x:str]: num;	(basic) str x converted to num
parse[y:str, symbols:str]: num -> str;	(utils) divides the string y in a list of substrings e.g. words that are separated by a delimiter in symbols
parse[y:str]: num -> str;	(utils) divides the string y in a list of substrings e.g. words that are separated by a space or by the characters , : ; " ! '
real[x:str]: real;	(basic) str x converted to real
rev[x:str]: str;	(utils) string x in reverse order
scan[x:str, y:str]: bool;	(utils) does x occur in y?
scanrest[x:str, y:str]: str;	(utils) tail of y, from first occurrence of x in y
str[x:num]: str;	(basic) x converted to string
strncmp[x:str, y:str, n:num]: bool;	(utils) are the first n characters of strings x and y the same?
strlen[x:str]: num;	(utils) length of string x
tail[x:str]: str;	(basic) string x without first character

5.5 Tuple functions

This section shows all tuple functions of the modules basic and utils.

upd[x:[ ], y:[ ]]: [ ];	(basic) update x according to y

5.6 Product functions

This section shows all product functions of the modules basic and utils.

pi1[x:T><S]: T;	(basic) first element of x
pi1[x: $(T><S)]: $T;	(utils) { y | y ∈ T and exists {z ∈ S} (y >< z) ∈ x }
pi2[x:T><S]: S;	(basic) second element of x
pi2[x: $(T><S)]: $S;	(utils) { z | z ∈ S and exists {y ∈ T} (y >< z) ∈ x }
prod[x:T, y:S]: T><S;	(basic) << x , y >> (pair of x and y)

5.7 Set functions

This section shows all set functions of the modules basic and utils

add[x: $num, y:num]: $num;	(utils) { z + y | z ∈ x }
add[x: $real, y:real]: $real;	(utils) { z + y | z ∈ x }
del[x:T, y: $T]: $T;	(basic) y \ { x }, x is deleted from the set y
elt[x:T, y: $T]: bool;	(basic) x ∈ y
ins[x:T, y: $T]: $T;	(basic) y union { x } (insert x into y)
isect[x: $T, y: $T]: $T;	(basic) x intersect y (intersection)
max[x: $num]: num;	(utils) z : z ∈ x and forall {y ∈ x} z >= y, maximum of set x

max[x: $real]: real;	(utils) z : z ∈ x and forall {y ∈ x} z >= y, maximum of set x
min[x: $num]: num;	(utils) z : z ∈ x and forall {y ∈ x} z <= y, minimum of set x
min[x: $real]: real;	(utils) z : z ∈ x and forall {y ∈ x} z <= y, minimum of set x
mod[x: $num, y:num]: num;	(utils) { z mod y | z ∈ x }
mult[x: $num, y:num]: num;	(utils) { z * y | z ∈ x }
mult[x: $real, y:real]: real;	(utils) { z * y | z ∈ x }
pi1[x: $(T><S)]: T;	(utils) { y | y ∈ T and exists {z ∈ S} (y >< z) ∈ x }
pi2[x: $(T><S)]: S;	(utils) { z | z ∈ S and exists {y T} (y >< z) ∈ x }
pick[x: $T]: T;	(basic) arbitrary (though deterministic) element from x
rdiv[x: $num, y:num]: num;	(utils) { z / y | z ∈ x }
rdiv[x: $real, y:real]: real;	(utils) { z / y | z ∈ x }
rest[x: $T]: $T;	(basic) x \ {pick(x)}
sdiff[x: $T, y: $T]: $T;	(basic) x \ y
size[x: $T]: num;	(basic) sum {y x} 1, number of elements of x
sub[x: $num, y:num]: num;	(utils) { z - y | z ∈ x }
sub[x: $real, y:real]: real;	(utils) { z - y | z ∈ x }
subset[x: $T, y: $T]: bool;	(utils) x <= y
sum[x: $num]: num;	(utils) sum {y ∈ x} y
sum[x: $real]: real;	(utils) sum {y ∈ x} y
union[x: $$T]: T;	(utils) Union_{y ∈ x} y, union of all elements of x
union[x: $T, y: $T]: T;	(basic) x union y

5.8 Statistical functions

In these functions, seed should be in the open interval (0,1), i.e., 0 < seed < 1. It can be obtained by using random.

bernouilli [p:real, seed:real]: real	(stat) draw from a bernouilli distribution with expectation p; 0 <= p <= 1
binomial [n:real, p:real, seed:real]: real	(stat) draw from a binomial distribution with expectation p*n; 0 <= p <= 1 and n should be a non-negative integer
chisq [n:real, seed:real]: real	(stat) draw from a X^2 distribution with n degrees of freedom; n should be a non-negative integer
erlang [m:real, k:real, seed:real]: real	(stat) draw from an erlang distribution with expectation k/m and variance k/m^2; k should be a positive integer and m > 0
nexp [m:real, seed:real]: real	(stat) draw from a negative exponential distribution with expectation 1/m; m > 0
gamma[l:real, k: real, seed:real]	(stat) draw from a gamma distribution with mean l/k and variance l/k^2.
normal [m:real, v:real, seed:real]: real	(stat) draw from a normal distribution with expectation m and variance v; v >= 0
PERT_beta [l:real, u:real, e: real, seed:real]	(stat) draw from a beta distribution with minimum l, maximum u and expectation e.
poisson [m:real, seed:real]: real	(stat) draw from a distribution of a poisson process with intensity m; m should be a non-negative integer
random [seed:real]: real	(stat) draw from a uniform distribution on the open interval (0,1)
rint [a:real, b:real, seed:real]: real	(stat) closed interval [a,b]. The values of a and b should be integer and a <= b.
student [n:real, seed:real]: real	(stat) draw from a t-distribution with n degrees of freedom. The value of n should be a positive integer.
uniform [a:real, b:real, seed:real]: real	(stat) draw from a uniform distribution on the open interval (a,b), a < b.

5.9 List functions

This section shows all list functions of the modules basic and utils.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

add[x:*num, y:num]: *num;	(utils) x with each member incremented by y
add[x:*real, y:real]: *real;	(utils) x with each member incremented by y
cat[x:*T, y:*T]: *T;	(basic) the concatenation of x and y
elt[x:T, y:*T]: bool;	(utils) x occurs in y
front[x:*T]: *T;	(utils) x with the last member removed
head[x:*T]: T;	(basic) the first member of x
ins[x:T, y:*T]: *T;	(basic) y with x added as head
last [x:*T]: *T;	(utils) the last member of x
list [x:T]: *T;	(basic) a list containing all members of x
max[x:*num]: num;	(utils) the maximum in x
max[x:*real]: real;	(utils) the maximum in x
min[x:*num]: num;	(utils) the minimum in x
min[x:*real]: real;	(utils) the minimum in x
mod[x:*num, y:num]: *num;	(utils) x with each member replaced by its value module y
mult[x:*num, y:num]: *num;	(utils) each member of x multiplied by y
mult[x:*real, y:real]: *real;	(utils) each member of x multiplied by y
rdiv[x:*num, y:num]: *num;	(utils) each member of x divided by y
rdiv[x:*real, y:real]: *real;	(utils) each member of x divided by y
reverse[x:*T]: *T;	(utils) list x in reverse order
set[x:*T]: T;	(basic) the set of members of x
size[x:*T]: num;	(utils) number of members of x
stretch[x:num, y:num]: *num;	(utils) list containing all members of { z | x <= z <= y }
sub[x:*num, y:num]: *num;	(utils) each member of x subtracted by y
sub[x:*real, y:real]: *real;	(utils) each member of x subtracted by y
sum[x:*num]: num;	(utils) the sum of all numbers in x
sum[x:*real]: real;	(utils) the sum of all numbers in x
tail[x:*T]: *T;	(basic) the tail of x

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

5.10 Mapping functions

This section shows all mapping functions.

all[x:T->bool]: bool;	(utils) forall {y ∈ dom(x)} x(y)
any[x:T->bool]: bool;	(utils) exists {y ∈ dom(x)} x(y)
apply[x:T->S, y:T]: S;	(basic) x(y) (mapping application)
apply[x:T->S, y: $T]: $S;	(utils) { x(z) | z ∈ y }
dom[x:T->S]: $T;	(utils) domain of x
fcomp[x:T->S, y:S->R]: T->R;	(utils) x o y, function composition
frest[x:S->T]: S->T;	(utils) x \ {pick(x)}, x without the element pick(x)
inv[x:T->S, y:S]: $T;	(basic) { z | z ∈ T and x(z) = y } (inverse)
inv[x:T->S, y: $S]: $T;	(utils) { z | z ∈ T and x(z) ∈ y } (inverse)
max[x:T->num]: num;	(utils) max(rng(x))
max[x:T->real]: real;	(utils) max(rng(x))
min[x:T->num]: num;	(utils) min(rng(x))
min[x:T->real]: real;	(utils) min(rng(x))
restrict[x:T->S, y:T]: T->S;	(utils) x |` y (restricted to)
rng[x:T->S]: $S;	(utils) range of x
set[x:T->bool]: $T;	(basic) { y | y ∈ T and x(y) }
sum[x:T->num]: num;	(utils) sum {y ∈ dom(x)} x(y)
sum[x:T->real]: real;	(utils) sum {y ∈ dom(x)} x(y)
that[x:T->bool]: T;	(utils) z : z ∈ dom(x) and x(z) and forall {y ∈ dom(x) and x(y)} y = z the only element in T that is true
tomap[x:(T><S)]: T->S;	(utils) type casting of set of pairs to mapping
upd[x:T->S, y:T, z:S]: T->S;	(basic) update x with x(y) = z

5.11 Void functions

This section shows all void functions.

abort: void;	(basic) value to denote abortion

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

tovoid[x:T]: void;	(utils) x converted to void

5.12 Array functions

This section shows the array functions of the module adt.

adel [x:num->T, y:num]: num->T;	(adt) array x without the y'th member
aelement [x:num->T, y:num]: T;	(adt) x(y), value of the y'th member of array x
aempty [x:num->T]: bool;	(adt) is x an empty array?
aindex [x:num->T, y:T]: $num;	(adt) {i:dom(x) | x(i)=y}
ains [x:num->T, y:num, z:T]: num->T;	(adt) array x with (y,z) inserted
ains [x:num->T, y:num, z:num->T]: num->T;	(adt) array x with array z inserted at position y
anull num -> Void;	(adt) the empty array.
asplit [x:num->T, y:num]: (num->T)><(num->T);	(adt) split array x at index y
aswap [x:num->T, y:num, z:num]: num->T;	(adt) array x with values at y and z interchanged

5.13 List functions

This section shows the list functions of the module adt.

lcat [x:num->T, y:num->T]: num->T;	(adt) concatenation of lists x and y
lcons [x:num->T, y:T]: num->T;	(adt) append y in front of list x
lempty [x:num->T]: bool;	(adt) is x an empty list?
lfront [x:num->T]: num->T;	(adt) list x without the last member
lhead [x:num->T]: T;	(adt) first member of list x
lnull: num->void;	(adt) the empty list
lreverse [x:num->T]: num->T;	(adt) reverse of list x
lsnoc [x:num->T, y:T]: num->T;	(adt) append y at the back of list x
ltail [x:num->T]: num->T;	(adt) list x without the first member
ltobag [x:num->T]: T->num;	(adt) list x converted to a bag

5.14 Bag functions

This section shows the bag functions of the module adt.

atobag [x:num->T]: T->num;	(adt) array x converted to a bag
badd [x:T->num, y: $T]: T->num;	(adt) bag x with all elements of y added
badd [x: $$T]: T->num;	(adt) bag of all elements occurring in the sets in x
bdel [x:T->num, y:T]: T->num;	(adt) bag x with one element y deleted
bdiff [x:T->num, y:T->num]: T->num;	(adt) bag x minus all elements of bag y
bempty [x:T->num]: bool;	(adt) is x an empty bag?
bins [x:T->num, y:T]: T->num;	(adt) bag x with element y inserted
bisect [x:T->num, y:T->num]: T->num;	(adt) biggest bag both contained in x and y (bag intersection)
bjoin [x:T->num, y:T->num]: T->num;	(adt) bag of all elements of x and y (join)
bjoin [x: $(T->num)]: T->num;	(adt) bag of all elements of all bags in x (join)
bmax [x:num->num]: num;	(adt) maximum (greatest element) of bag x
bmin [x:num->num]: num;	(adt) minimum (smallest element) of bag x
bnull: void->num;	(adt) the empty bag
bocc [x:T->num, y:T]: num;	(adt) number of times y occurs in bag x
bpick [x:T->num]: T;	(adt) an arbitrary (though deterministic) element from bag x
bproj [x:T->num, y: $T]: T->num;	(adt) projection of bag x onto y
brest [x:T->num]: T->num;	(adt) bag x without the element bpick(x)
bsize [x:T->num]: num;	(adt) number of elements in bag x
bsum [x:num->num]: num;	(adt) sum of all elements of bag x
btoset [x:T->num]: $T;	(adt) bag x converted to a set
bunion [x:T->num, y:T->num]: T->num;	(adt) smallest bag containing x and y (bag union)
bunion [x: $(T->num)]: T->num;	(adt) smallest bag containing all bags of x (bag union)
ltobag [x:num->T]: T->num;	(adt) list x converted to a bag
stobag [x: $T]: T->num;	(adt) set x converted to a bag
subbag [x:T->num, y:T->num]: bool;	(adt) is bag x contained in y? (bag subset)

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

6. Bibliography

References

For a good introduction to the modelling of systems and processes in terms of Petri nets, the user is referred to [AAKea96] and [AH97]. Both books are in Dutch. Other goods introductions to Petri-net modelling are given in: [Pet81], [Jen92], and [Rei85]. The tools described in [Pal97] and [SL96] export script files, which can be imported by ExSpect. For more background information on modelling and simulation with ExSpect, the user is referred to [Aal94a], [Aal94b], [Aal96a], [Aal96b], [DE95], [Hee94], [Ros90], and [JR91].

[AAKea96] W.M.P. van der Aalst, A. Aarts, H. Koppelman, and R.V. Schuwer et al. Informatiesystemen: Modelleren en Specificeren. Open Universiteit, Heerlen, 1996 (in Dutch).

[Aal94a] W.M.P. van der Aalst. Procesmodelleren met behulp van Petri-netten. Informatie, 36(4):244--252, 1994 (in Dutch).

[Aal94b] W.M.P. van der Aalst. Putting Petri nets to work in industry. Computers in Industry, 25(1):45--54, 1994.

[Aal96a] W.M.P. van der Aalst. Petri-net-based Workflow Management Software. In A. Sheth, editor, Proceedings of the NFS Workshop on Workflow and Process Automation in Information Systems, pages 114--118, Athens, Georgia, May 1996.

[Aal96b] W.M.P. van der Aalst. Three Good reasons for Using a Petri-net-based Workflow Management System. In S. Navathe and T. Wakayama, editors, Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96), pages 179--201, Camebridge, Massachusetts, Nov 1996.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.

[AH97] W.M.P. van der Aalst and K.M. van Hee. Workflow Management: Modellen, Methoden en Systemen (in Dutch). Academic Service, Schoonhoven, 1997 (in Dutch).

[DE95] J. Desel and J. Esparza. Free choice Petri nets, volume 40 of Cambridge tracts in theoretical computer science. Cambridge University Press, Cambridge, 1995.

[Hee94] K.M. van Hee. Information System Engineering: a Formal Approach. Cambridge University Press, 1994.

[Jen92] K. Jensen. Coloured Petri Nets. Basic concepts, analysis methods and practical use. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1992.

[JR91] K. Jensen and G. Rozenberg, editors. High-level Petri Nets: Theory and Application. Springer-Verlag, Berlin, 1991.

[Pal97] Pallas Athena. Protos User Manual. Pallas Athena BV, Plasmolen, The Netherlands, 1997.

[Pet81] J.L. Peterson. Petri net theory and the modeling of systems. Prentice-Hall, Englewood Cliffs, 1981.

[Rei85] W. Reisig. Petri nets: an introduction, volume 4 of Monographs in theoretical computer science: an EATCS series. Springer-Verlag, Berlin, 1985.

[Ros90] S.M. Ross. A course in simulation. Macmillan, New York, 1990.

[SL96] Software-Ley. COSA User Manual. Software-Ley GmbH, Pullheim, Germany, 1996.

© 1999 Deloitte & Touche Bakkenist, ExSpect is a registered trade mark of ASPT.