Chi 3 tutorial

I.J.B.F. Adan, A.T. Hofkamp, and J.E. Rooda

November 17, 2017

Contents

Chapter 1

Introduction

This text is about the modeling of the operation of systems, e.g. semiconductor factories, assembly and packaging lines, car manufacturing plants, steel foundries, metal processing shops, beer breweries, health care systems, warehouses, order-picking systems. For a proper functioning of these systems, these systems are controlled by operators and electronic devices, e.g. computers.

During the design process, engineers make use of (analytical) mathematical models, e.g. algebra and probability theory, to get answers about the operation of the system. For complex systems, (numerical) mathematical models are used, and computers perform 'simulation' experiments, to analyze the operation of the system. Simulation studies give answers to questions like: What is the throughput of the system?; What is the effect of set-up time in a machine?; How will the batch size of an order influence the flow time of the product-items?; What is the effect of more surgeons in a hospital?

The operation of a system can be described, e.g. in terms of concurrent or parallel operating processes. An example of a system with parallel operating processes is a manufacturing line, with a number of manufacturing machines, where product-items go from machine to machine. A surgery room in a hospital is a system where patients are treated by teams using

medical equipment and sterile materials. A biological system can be described by a number of parallel processes, where, e.g. processes transform sugars into water and carbon-dioxide producing energy. In all these examples, processes operate in parallel to complete a task, and to achieve a goal. Concurrency is the dominant aspect in these type of systems, and as a consequence this holds too for their models.

The operating behavior of parallel processes can be described by different formalisms, e.g. automata, Petri-nets or parallel processes. This text uses the programming language Chi 3, based on an algebra of concurrent processes defined in terms of structural operational semantics (SOS).

A system is abstracted into a model, with cooperating processes, where processes are connected to each other via channels. The channels are used for exchanging material and information. Models of the above mentioned examples consist of a number of concurrent processes connected by channels, denoting the flow of products, patients or personnel.

In Chi 3, communication takes place in a synchronous manner. This means that communication between a sending process, and a receiving process takes place only when both processes are able to communicate. Processes and channels can dynamically be altered. To model times, e.g. inter arrival times and server processing times, the language has a notation of time.

The rationale behind the language is that models for the analysis of a system should be formal (exactly one interpretation, every reader attaches the same meaning to the model), easily writable (write the essence of the system in a compact way), easily readable (non-experts should be able to understand the model), and easily extendible (adding more details in one part should not affect other parts). Verification of the models to investigate the properties of the model should be effortless. (A model has to preserve some properties of the real system otherwise results from the simulation study have no relation with the system being modeled. The language must allow this verification to take place in a simple manner.) Experiments should be performed in an straightforward manner. (Minimizing the effort in doing simulation studies, in particular for large systems, makes the language useful.) Finally, the used models should be usable for the supervisory

(logic) control of the systems (simulation studies often provide answers on how to control a system in a better way, these answers should also work for the modeled system).

1.1 Chi 3 in a nutshell

During the past decades, Chi 3 and his ancestors have been used with success, for the analysis of a variety of (industrial) systems. Based on this experience, the language Chi 3 has been completely redesigned, keeping the strong points of the previous versions, while making it more powerful for advanced users, and easier to access for non-experts.

Its features are:

- A system (and its control) is modeled as a collection of parallel running processes, communicating with each other using channels.
- Processes do not share data with other processes and channels are synchronous (sending and receiving is always done together at the same time), making reasoning about process behaviour easier.
- Processes and channels are dynamic, new processes can be created as needed, and communication channels can be created or rerouted.
- Variables can have elementary values such as *boolean*, *integer* or *real* numbers, to high level structured collections of data like *lists*, *sets* and *dictionaries* to model the data of the system. If desired, processes and channels can also be part of that data.
- A small generic set of statements to describe algorithms, assignment, *if*, *while*, and *for* statements. This set is relatively easy to explain to non-experts, allowing them to understand the model, and participate in the discussions.
- Tutorials and manuals demonstrate use of the language for effective modeling of system processes. More detailed modeling of the pro-

cesses, or custom tailoring them to the real situation, has no inherent limits.

- Time and (quasi-) random number generation distributions are available for modeling behavior of the system in time.
- Likewise, measurements to derive performance indicators of the modeled system are integrated in the model. Tutorials and manuals show basic use. The integration allows for custom solutions to obtain the needed data in the wanted form.
- Input and output facilities from and to the file system exists to support large simulation experiments.

1.2 Exercises

1. Install the Chi 3 programming environment. Follow the instructions given at `http://chi.se.wtb.tue.nl/`.
2. Test your first program.
   (a) Copy the following program in the workspace of your computer:

   ```
   model M():
      writeln("It works!")
   end
   ```

   (b) Compile, and simulate the model as explained in the tool manual.
   (c) Try to explain the result.

   (a) Copy the following program in the same manner:

   ```
   model M(string s):
      write("%s\n", s)
   end
   ```

(b) Simulate the model, where you have to set the *Model instance* text to `M("OOPS")`.

(c) Try to explain the result.

Chapter 2

Data types

The language is a statically typed language, which means that all variables and values in a model have a single fixed type. All variables must be declared in the program. The declaration of a variable consists of the type, and the name, of the variable. The following fragment shows the declaration of two elementary data types, integer variable `i` and real variable `r`:

```
...
int i;
real r;
...
```

The ellipsis (`...`) denotes that non-relevant information is left out from the fragment. The syntax for the declaration of variables is similar to the language *C*. All declared variables are initialized, variables `i` and `r` are both initialized to zero.

An expression, consisting of operators, e.g. plus (`+`), times (`*`), and operands, e.g. `i` and `r`, is used to calculate a new value. The new value can be assigned to a variable by using an *assignment* statement. An example with four variables, two expressions and assignment statements is:

```
...
int  i = 2, j;
```

```
real r = 1.50, s;

j = 2 * i + 1;
s = r / 2;
...
```

The value of variable `j` becomes `5`, and the value of `s` becomes `0.75`. Statements are described in Chapter 3.

Data types are categorized in five different groups: *elementary* types, *tuple* types, *container* types, *custom* types, and *distribution* types. Elementary types are types such as Boolean, integer, real or string. Tuple types contain at least one element, where each element can be of different type. In other languages tuple types are called records (Pascal) or structures (C). Variables with a container type (a list, set, or dictionary) contain many elements, where each element is of the same type. Custom types are created by the user to enhance the readability of the model. Distributions types are types used for the generation of distributions from (pseudo-) random numbers. They are covered in Chapter 6.

2.1 Elementary types

The elementary data types are Booleans, numbers and strings. The language provides the elementary data types:

- `bool` for booleans, with values `false` and `true`.
- `int` for integers, e.g. `-7`, `20`, `0`.
- `real` for reals, e.g. `3.14`, `7.0e9`.
- `string` for text strings, e.g. `"Hello"`, `"world"`.

Booleans

A boolean value has two possible values, the truth values. These truth values are `false` and `true`. The value `false` means that a property is not

fulfilled. A value `true` means the presence of a property. Boolean variables are initialized with the value `false`.

In mathematics, various symbols are used for unary and binary boolean operators. These operators are also present in Chi 3. The most commonly used boolean operators are `not`, `and`, and `or`. The names of the operators, the symbols in mathematics and the symbols in the language are presented in Table 2.1.

Operator	Math	Chi 3
boolean not	$\neg$	`not`
boolean and	$\wedge$	`and`
boolean or	$\vee$	`or`

Table 2.1: Table with boolean symbols.

Examples of boolean expressions are the following. If `z` equals `true`, then the value of `(not z)` equals `false`. If `s` equals `false`, and `t` equals `true`, then the value of the expression `(s or t)` becomes `true`.

The result of the unary `not`, the binary `and` and `or` operators, for two variables `p` and `q` is given in Table 2.2.

p	q	not p	p and q	p or q
false	false	true	false	false
false	true		false	true
true	false	false	false	true
true	true		true	true

Table 2.2: Truth table for `not`, `and` and `or` operators.

If `p = true` and `q = false`, we find for `p or q` the value `true` (third line in Table 2.2).

Numbers

In the language, two types of numbers are available: integer numbers and real numbers. Integer numbers are whole numbers, denoted by type `int` e.g. `3`, `-10`, `0`. Real numbers are used to present numbers with a fraction, denoted by type `real`. E.g. `3.14`, `2.7e6` (the scientific notation for 2.7 million). Note that real numbers *must* either have a fraction or use the scientific notation, to let the computer know you mean a real number (instead of an integer number). Integer variables are initialized with `0`. Real variables are initialized with `0.0`.

For numbers, the normal arithmetic operators are defined. Expressions can be constructed with these operators. The arithmetic operators are in Table 2.3.

Operator name	Notation	Comment
unary plus	`+ x`	
unary minus	`- x`	
raising to the power	`x ^ y`	always a `real` result
multiplication	`x * y`	
real division	`x / y`	always a `real` result
division	`x div y`	for `int` only
modulo	`x mod y`	for `int` only
addition	`x + y`	
subtraction	`x - y`	

Table 2.3: The arithmetic operators

The priority of the operators is given from high to low. The unary operators have the strongest binding, and the `+` and `-` the weakest binding. So, `-3^2` is read as `(-3)^2` and not `-(3^2)`, because the priority rules say that the unary operator binds stronger than the raising to the power operator. Binding in expressions can be changed by the use of parentheses.

The integer division, denoted by `div`, gives the biggest integral number smaller or equal to `x / y`. The integer remainder, denoted by `mod`, gives

the remainder after division `x - y * (x div y)`. So, `7 div 3` gives `2` and `-7 div 3` gives `-3`, `7 mod 3` gives `1` and `-7 mod 3` gives `2`.

The rule for the result of an operation is as follows. The real division and raising to the power operations always produce a value of type `real`. Otherwise, if both operands (thus `x` and `y`) are of type `int`, the result of the operation is of type `int`. If one of the operands is of type `real`, the result of the operation is of type `real`.

Conversion functions exist to convert a real into an integer. The function `ceil` converts a real to the smallest integer value not less than the real, the function `floor` gives the biggest integer value smaller than or equal to the real, and the function `round` rounds the real to the nearest integer value (or up, if it ends on `.5`). Between two numbers a relational operation can be defined. If e.g. variable `x` is smaller than variable `y`, the expression `x < y` equals `true`. The relational operators, with well known semantics, are listed in Table 2.4.

Name	Operator
less than	`x < y`
at most	`x <= y`
equals	`x == y`
differs from	`x != y`
at least	`x >= y`
greater than	`x > y`

Table 2.4: The relational operators

Strings

Variables of type string contains a sequence of characters. A string is enclosed by double quotes. An example is `"Manufacturing networks"`. Strings can be composed from different strings. The concatenation operator (`+`) adds one string to another, e.g. `"Systems" + " " + "engineering"` gives `"Systems engineering"`. Moreover the relational operators (`<`, `<=`,

==, != >=, and >) can be used to compare strings alphabetically, e.g. `"a" < "aa" < "ab" < "b"`. String variables are initialized with the empty string `""`.

2.2 Tuple types

Tuple types are used for keeping several (related) kinds of data together in one variable, e.g. the name and the age of a person. A tuple variable consists of a number of fields inside the tuple, where the types of these fields may be different. The number of fields is fixed. One operator, the projection operator denoted by a dot (`.`), is defined for tuples. It selects a field in the tuple for reading or assigning.

Notation

A type `person` is a tuple with two fields, a 'name' field of type `string`, and an 'age' field of type `int`, is denoted by:

```
type person = tuple(string name; int age)
```

Operator

A projection operator fetches a field from a tuple. We define two persons:

```
person eva  = ("eva" , 29),
       adam = ("adam", 27);
```

And we can speak of `eva.name` and `adam.age`, denoting the name of `eva` (`"eva"`) and the age of `adam` (27). We can assign a field in a tuple to another variable:

```
ae = eva.age;
eva.age = eva.age + 1;
```

This means that the age of `eva` is assigned tot variable `ae`, and the new age of `eva` becomes `eva.age + 1`.

By using a multi assignment statement all values of a tuple can be copied into separate variables:

```
string name;
int age;

name, age = eva
```

This assignment copies the name of `eva` into variable `name` of type `string` and her age into `age` of type `int`.

2.3 Container types

Lists, sets and dictionaries are container types. A variable of this type contains zero or more identical elements. Elements can be added or removed in variables of these types. Variables of a container type are initialized with zero elements.

Sets are unordered collections of elements. Each element value either exists in a set, or it does not exist in a set. Each element value is unique, duplicate elements are silently discarded. A list is an ordered collection of elements, i.e. there is a first and a last element (in a non-empty list). A list also allows duplicate element values. Dictionaries are unordered and have no duplicate value, just like sets, but you can associate a value (of a different type) with each element value.

Lists are denoted by a pair of (square) brackets. For example, `[7, 8, 3]` is a list with three integer elements. Since a list is ordered, `[8, 7, 3]` is a different list. With empty lists, the computer has to know the type of the elements, e.g. `<int>[]` is an empty list with integer elements. The prefix `<int>` is required in this case.

Sets are denoted by a pair of (curly) braces, e.g. `{7, 8, 3}` is a set with three integer elements. As with lists, for an empty set a prefix is required, for example `<string>{}` is an empty set with strings. A set is an unordered collection of elements. The set `{7, 8, 3}` is a set with three integer numbers. Since order of the elements does not matter, the same

set can also be written as `{8, 3, 7}` (or in one of the four other orders). In addition, each element in a set is unique, e.g. `{8, 7, 8, 3}` is equal to `{7, 8, 3}`. For readability, elements in a set are normally written in increasing order, i.e. as `{3, 7, 8}`.

Dictionaries are denoted by a pair of (curly) braces, whereby an element value consists of two parts, a 'key' and a 'value' part. The two parts separated by a colon (`:`). For example `{"jim" : 32, "john" : 34}` is a dictionary with two elements. The first element has `"jim"` as key part and `32` as value part, the second element has `"john"` as key part and `34` as value part. The key parts of the elements work like a set, they are unordered and duplicates are silently discarded. A value part is associated with its key part. In this example, the key part is the name of a person, while the value part keeps the age of that person. Empty dictionaries are written with a type prefix just like lists and sets, e.g. `<string:int>{}`.

Container types have some built-in functions (Functions are described in Chapter 4) in common:

- The function `size` gives the number of elements in a variable. E.g. `size([7, 8, 3])` yields 3; `size({7, 8})` results in 2; `size({"jim":32})` gives 1 (an element consists of two parts).

- The function `empty` yields `true` if there are no elements in variable. E.g. `empty(<string>{})` with an empty set of type `string` is true. (Here the type `string` is needed to determine the type of the elements of the empty set.)

- The function `pop` extracts a value from the provided collection and returns a tuple with that value, and the collection minus the value.

  For `lists` the first element of the list becomes the first field of the tuple. The second field of the tuple becomes the list minus the first list element. E.g.

  ```
  pop([7, 8, 3]) -> (7, [8, 3])
  ```

The '`->`' above denotes 'yields'. The value of the list is split into a 'head' (the first element) and a 'tail' (the remaining elements).

For sets the first field of the tuple becomes the value of an arbitrary element from the set. The second field of the tuple becomes the original set minus the arbitrary element. For example, a `pop` on the set `{8, 7, 3}` thus has three possible answers:

```
pop({8, 7, 3}) -> (7, {3, 8}) or
pop({8, 7, 3}) -> (3, {7, 8}) or
pop({8, 7, 3}) -> (8, {3, 7})
```

Performing a `pop` on a dictionary follows the same pattern as above, except 'a value from the collection' are actually a key item and a value item. In this case, the `pop` function gives a three-tuple as result. The first field of the tuple becomes the key of the extracted element, the second field of the tuple becomes the value of the element, and the third field of the tuple contains the dictionary except for the extracted element.

```
pop({"a" : 32, "b" : 34}) -> ("a", 32, {"b" : 34}) or
pop({"a" : 32, "b" : 34}) -> ("b", 34, {"a" : 32})
```

Lists

A list is an ordered collection of elements of the same type. They are useful to model anything where duplicate values may occur or where order of the values is significant. E.g. waiting customers in a shop, process steps in a recipe, or products stored in a warehouse. Various operations are defined for lists.

An element can be fetched by *indexing*. This indexing operation does not change the content of the variable. The first element of a list has index `0`. The last element of a list has index `size(xs) - 1`. A negative index, say `m`, starts from the back of the list, or equivalently, at offset `size(xs) + m` from the front. You cannot index non-existing elements. Some examples, with `xs = [7, 8, 3, 5, 9]` are:

```
xs[0]  -> 7
xs[3]  -> 5
xs[5]  -> ERROR (there is no element at position 5)
xs[-1] -> xs[5 - 1] -> xs[4] -> 9
xs[-2] -> xs[5 - 2] -> xs[3] -> 5
```

In Figure 2.1 the list with indices is visualized.

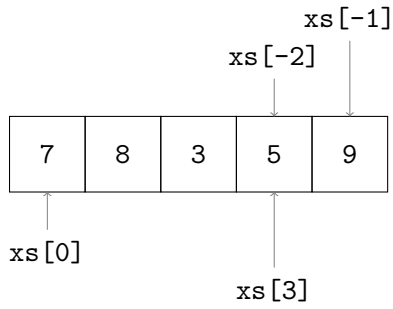


Figure 2.1: A list with indices.

A part of a list can be fetched by *slicing*. The slicing operation does not change the content of the list, it copies a contiguous sequence of a list. The result of a slice operation is again a list, even if the slice contains just one element.

Slicing is denoted by `xs[i:j]`. The slice of `xs[i:j]` is defined as the sequence of elements with index `k` such that `i <= k < j`. Note the upper bound `j` is noninclusive. If `i` is omitted use `0`. If `j` is omitted use `size(xs)`. If `i` is greater than or equal to `j`, the slice is empty. If `i` or `j` is negative, the index is relative to the end of the list: `size(xs) + i` or `size(xs) + j` is substituted. Some examples with `xs = [7, 8, 3, 5, 9]`:

```
xs[1:3] -> [8, 3]
xs[:2]  -> [7, 8]
xs[1:]  -> [8, 3, 5, 9]
```

```
xs[:-1] -> [7, 8, 3, 5]
xs[:-3] -> [7, 8]
```

A common name for the first element of a list (i.e., `x[0]`) is the *head* of a list. The list of all but the first elements (`xs[1:]`) is often called *tail*. Similarly, the last element of a list (`xs[-1]`) is also known as *head right*, and `xs[:-1]` is also known as *tail right*. In Figure 2.2 the slicing operator is visualized.

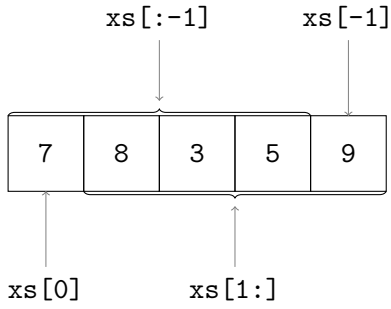


Figure 2.2: A list with indices and slices.

Two lists can be 'glued' into a new list. The glueing or concatenation of a list with elements `7`, `8`, `3` and a list with elements `5`, and `9` is denoted by:

```
[7, 8, 3] + [5, 9] -> [7, 8, 3, 5, 9]
```

An element can be added to a list at the rear or at the front. The action is performed by transforming the element into a list and then concatenate these two lists. In the next example the value `5` is added to the rear, respectively the front, of a list:

```
[7, 8, 3] + [5] -> [7, 8, 3, 5]
[5] + [7, 8, 3] -> [5, 7, 8, 3]
```

Elements also can be removed from a list. The `del` function removes by position, e.g. `del(xs, 2)` returns the list `xs` without its third element (since positions start at index 0). Removing a value by value can be performed by the subtraction operator `-`. For instance, consider the following subtractions:

```
[1, 4, 2, 4, 5] - [2] -> [1, 4, 4, 5]
[1, 4, 2, 4, 5] - [4] -> [1, 2, 4, 5]
[1, 4, 2, 4, 5] - [8] -> [1, 4, 2, 4, 5]
```

Every element in the list at the right is searched in the list at the left, and if found, the *first* occurrence is removed. In the first example, element `2` is removed. In the second example, only the first value `4` is removed and the second value (at position 3) is kept. In the third example, nothing is removed, since value `8` is not in the list at the left.

When the list at the right is longer than one element, the operation is repeated. For example, consider `xs - ys`, whereby `xs = [1, 2, 3, 4, 5]` and `ys = [6, 4, 2, 3]`. The result is computed as follows.

```
    [1, 2, 3, 4, 5] - [6, 4, 2, 3]
-> ([1, 2, 3, 4, 5] - [6]) - [4, 2, 3]
->  [1, 2, 3, 4, 5] - [4, 2, 3]
-> ([1, 2, 3, 4, 5] - [4]) - [2, 3]
->  [1, 2, 3,    5] - [2, 3]
-> ([1, 2, 3,    5] - [2]) - [3]
->  [1,    3,    5] - [3]
-> [1,5]
```

Lists have two relational operators, the equal operator and the not-equal operator. The equal operator (`==`) compares two lists. If the lists have the same number of elements and all the elements are pair-wise the same, the result of the operation is `true`, otherwise `false`. The not-equal operator (`!=`) does the same check, but with an opposite result. Some examples, with `xs = [7, 8, 3]`:

```
xs == [7, 8, 3] -> true
xs == [7, 7, 7] -> false
```

The membership operator (`in`) checks if an element is in a list. Some examples, with `xs = [7, 8, 3]`:

```
6 in xs -> false
7 in xs -> true
8 in xs -> true
```

Initialization

A list variable is initialized with a list with zero elements. E.g.

```
list int xs
```

The initial value of `xs` equals `<int>[]`.

A list can be initialized with a number, denoting the number of elements in the list:

```
list(2) int ys
```

This declaration creates a list with `2` elements, whereby each element of type `int` is initialized. The initial value of `ys` equals `[0, 0]`. Another example with a list of lists:

```
list(4) list(2) int zm
```

This declaration initializes variable `zm` with the value `[ [0, 0], [0, 0], [0, 0], [0, 0] ]`.

Sets

Set operators for union, intersection and difference are present. Table 2.5 gives the name, the mathematical notation and the notation in the language.

The union of two sets merges the values of both sets into one, that is, the result is the collection of values that appear in at least one of the arguments of the union operation. Some examples:

```
{3, 7, 8} + {5, 9} -> {3, 5, 7, 8, 9}
```

Operator	Math	Chi 3
set union	$\cup$	`+`
set intersection	$\cap$	`*`
set difference	$\setminus$	`-`

Table 2.5: Table with set operations.

All permutations with the elements `3`, `5`, `7`, `8` and `9` are correct (sets have no order, all permutations are equivalent). To keep sets readable the elements are sorted in increasing order in this tutorial.

Values that occur in both arguments, appear only one time in the result (sets silently discard duplicate elements).

```
{3, 7, 8} + {7, 9} -> {3, 7, 8, 9}
```

The intersection of two sets gives a set with the common elements, that is, all values that occur in *both* arguments. Some examples:

```
{3, 7, 8} * {5, 9} -> <int>{}   # no common element
{3, 7, 8} * {7, 9} -> {7}       # only 7 in common
```

Set difference works much like subtraction on lists, except elements occur at most one time (and have no order). The operation computes 'remaining elements'. The result is a new set containing all values from the first set which are not in the second set. Some examples:

```
{3, 7, 8} - {5, 9} -> {3, 7, 8}
{3, 7, 8} - {7, 9} -> {3, 9}
```

The membership operator `in` works on sets too:

```
3 in {3, 7, 8} -> true
9 in {3, 7, 8} -> false
```

Dictionaries

Elements of dictionaries are stored according to a key, while lists elements are ordered by a (relative) position, and set elements are not ordered at all. A dictionary can grow and shrink by adding or removing elements respectively, like a list or a set. An element of a dictionary is accessed by the key of the element.

The dictionary variable `d` of type `dict(string : int)` is given by:

```
dict (string : int) d =
   {"jim"  : 32,
    "john" : 34,
    "adam" : 25}
```

Retrieving values of the dictionary by using the key:

```
d["john"] -> 34
d["adam"] -> 25
```

Using a non-existing key to retrieve a value results in a error message.

A new value can be assigned to the variable by selecting the key of the element:

```
d["john"] = 35
```

This assignment changes the value of the `"john"` item to `35`. The assignment can also be used to add new items:

```
d["lisa"] = 19
```

Membership testing of keys in dictionaries can be done with the `in` operator:

```
"jim" in d -> true
"peter" in d -> false
```

Merging two dictionaries is done by adding them together. The value of the second dictionary is used when a key exists in both dictionaries:

```
{1 : 1, 2 : 2} + {1 : 5, 3 : 3} -> {1 : 5, 2 : 2, 3 : 3}
```

The left dictionary is copied, and updated with each item of the right dictionary.

Removing elements can be done with subtraction, based on key values. Lists and sets can also be used to denote which keys should be removed. A few examples for `p` is `{1 : 1, 2 : 2}`:

```
p - {1 : 3, 5 : 5} -> {2 : 2}
p - {1, 7} -> {2 : 2}
p - [2, 8] -> {1 : 1}
```

Subtracting keys that do not exist in the left dictionary is allowed and has no effect.

2.4 Custom types

To structure data the language allows the creation of new types. Types can be used as alias for elementary data types to increase readability, e.g. a variable of type item:

```
type item = real;
```

Variables of type `item` are, e.g.:

```
item box, product;
box = 4.0; product = 120.5;
```

This definition creates the possibility to speak about an item.

Types also can be used to make combinations of other data types, e.g. a recipe:

```
type step = tuple(string name; real process_time),
     recipe = tuple(int id; list step steps);
```

A type `step` is defined by a `tuple` with two fields, a field with `name` of type `string`, denoting the name of the step, and a field with `process_time` of type `real`, denoting the duration of the (processing) step. The `step` definition is used in the type `recipe`. Type `recipe` is defined by a `tuple` with two fields, an `id` of type `int`, denoting the identification number, and a field `steps` of type `list step`, denoting a list of single steps. Variables of type recipe are, e.g.:

```
recipe plate, bread;
plate = (34, [("s", 10.8), ("w", 13.7), ("s", 25.6)]);
bread = (90, [("flour", 16.3), ("yeast", 6.9)]);
```

2.5 Exercises

1. Exercises for integer numbers.

   (a) What is the result of the following expressions:

   ```
   -5 ^ 3
   -5 * 3
   -5 mod 3
   ```

2. Exercises for tuples. Given are tuple type `box` and variable `x` of type `box`:

   ```
   type box = tuple(string name; weight: real);
   box x = ("White", 12.5);
   ```

   (a) What is the result of the following expressions:

   ```
   x.name
   x.real
   x
   ```

3. Exercises for lists. Given is the list `xs = [0,1,2,3,4,5,6]`. Determine the outcome of:

```
xs[0]
xs[1:]
size(xs)
xs + [3]
[4,5] + xs
xs - [2,2,3]
xs - xs[2:]
xs[0] + (xs[1:])[0]
```

Chapter 3

Statements

There are several kinds of statements, such as assignment statements, choice statements (select and if statements), and loop statements (while and for statements).

Semicolons are required after statements, except at the end of a sequence (that is, just before an `end` keyword and after the last statement) or after the keyword `end`. In this text semicolons are omitted before `end`.

3.1 The assignment statement

An *assignment* statement is used to assign values to variables. An example:

```
y = x + 10
```

This assignment consists of a name of the variable (`y`), an assignment symbol (`=`), and an expression (`x + 10`) yielding a value. For example, when `x` is `2`, the value of the expression is `12`. Execution of this statement copies the value to the `y` variable, immediately after executing the assignment, the value of the `y` variable is `10` larger than the value of the `x` variable at this point of the program. The value of the `y` variable will not change until the next assignment to `y`, for example, performing the assignment `x = 7` has no effect on the value of the `y` variable.

An example with two assignment statements:

```
i = 2;
j = j + 1
```

The values of `i` becomes 2, and the value of `j` is incremented. Independent assignments can also be combined in a multi-assignment, e.g.

```
i, j = 2, j + 1
```

The result is the same as the above described example, the first value goes into the first variable, the second value into the second variable, etc.

In an assignment statement, first all expression values are computed before any assignment is actually done. In the following example the values of `x` and `y` are swapped:

```
x, y = y, x;
```

3.2 The `if` statement

The *if* statement is used to express decisions. An example:

```
if x < 0:
   y = -x
end
```

If the value of `x` is negative, assign its negated value to `y`. Otherwise, do nothing (skip the `y = -x` assignment statement).

To perform a different statement when the decision fails, an `if`-statement with an `else` alternative can be used. It has the following form. An example:

```
if a > 0:
   c = a
else:
   c = b
end
```

If a is positive, variable c gets the value of a, otherwise it gets the value of b.

In some cases more alternatives must be tested. One way of writing it is by nesting an if-statement in the else alternative of the previous if-statement.

```
if i < 0:
   writeln("i < 0")
else:
   if i == 0:
      writeln("i = 0")
   else:
      if i > 0 and i < 10:
         writeln("0 < i < 10")
      else:
         # i must be greater or equal 10
         writeln("i >= 10")
      end
   end
end
```

This tests i < 0. If it fails, the else is chosen, which contains a second if-statement with the i == 0 test. If that test also fails, the third condition i > 0 and i < 10 is tested, and one of the writeln statements is chosen.

The above can be written more compactly by combining an else-part and the if-statement that follows, into an elif part. Each elif part consists of a boolean expression, and a statement list. Using elif parts results in:

```
if i < 0:
   writeln("i < 0")
elif i == 0:
   writeln("i = 0")
elif i > 0 and i < 10:
   writeln("0 < i < 10")
```

```
else:
   # i must be greater or equal 10
   writeln("i >= 10")
end
```

Each alternative starts at the same column, instead of having increasing indentation. The execution of this combined statement is still the same, an alternative is only tested when the conditions of all previous alternatives fail.

Note that the line '`# i must be greater or equal 10`' is a comment to clarify when the alternative is chosen. It is not executed by the simulator. You can write comments either at a line by itself like above, or behind program code. It is often useful to clarify the meaning of variables, give a more detailed explanation of parameters, or add a line of text describing what the purpose of a block of code is from a birds-eye view.

3.3 The `while` statement

The *while* statement is used for repetitive execution of the same statements, a so-called *loop*. A fragment that calculates the sum of `10` integers, `10, 9, 8, ..., 3, 2, 1`, is:

```
int i = 10, sum;
while i > 0:
   sum = sum + i; i = i - 1
end
```

Each iteration of a `while` statement starts with evaluating its condition (`i > 0` above). When it holds, the statements inside the while (the `sum = sum + i; i = i - 1` assignments) are executed (which adds `i` to the sum and decrements `i`). At the end of the statements, the `while` is executed again by evaluating the condition again. If it still holds, the next iteration of the loop starts by executing the assignment statements again, etc. When the condition fails (`i` is equal to `0`), the `while` statement ends, and execution continues with the statement following `end`.

A fragment with an infinite loop is:

```
while true:
   i = i + 1;
   ...
end
```

The condition in this fragments always holds, resulting in `i` getting incremented 'forever'. Such loops are very useful to model things you switch on but never off, e.g. processes in a factory.

A fragment to calculate `z = x^y`, where `z` and `x` are of type `real`, and `y` is of type `integer` with a non-negative value, showing the use of two `while` loops, is:

```
real x; int y; real z = 1;
while y > 0:
   while y mod 2 == 0:
      y = y div 2; x = x * x
   end;
   y = y - 1; z = x * z
end
```

A fragment to calculate the greatest common divisor (GCD) of two integer numbers `j` and `k`, showing the use of `if` and `while` statements, is:

```
while j != k:
   if j > k:
      j = j - k
   else:
      k = k - j
   end
end
```

The symbol `!=` stands for 'differs from' ('not equal').

3.4 The `for` statement

The while statement is useful for looping until a condition fails. The *for* statement is used for iterating over a collection of values. A fragment with the calculation of the sum of `10` integers:

```
int sum;
for i in range(1,11):
   sum = sum + i
end
```

The result of the expression `range(1, 11)` is a list whose items are consecutive integers from `1` (included) up to `11` (excluded): `[1, 2, 3, ..., 9, 10]`.

The following example illustrates the use of the for statement in relation with container-type variables. Another way of calculating the sum of a list of integer numbers:

```
list int xs = [1,2,3,5,7,11,13];
int sum;
for x in xs:
   sum = sum + x
end
```

This statement iterates over the elements of list `xs`. This is particularly useful when the value of `xs` may change before the `for` statement.

3.5 Notes

In this chapter the most used statements are described. The language offers the following extensions:

1. Inside loop statements *break* and *continue* statements are allowed. The `break` statements allows 'breaking out of a loop', that is, abort a while or a for statement. The `continue` statement aborts execution of the statements in a loop. It 'jumps' to the start of the next iteration.

2. A rarely used statement is the *pass* statement. It's like an `x = x` assignment statement, but more clearly expresses 'nothing is done here'.

3.6 Exercises

1. Study the Chi 3 specification below and explain why, though it works, it is not an elegant way of modelling the selection. Make a suggestion for a shorter, more elegant version.

```
model M():
   int i = 3;

   if (i <  0) == true:
      write("%d is a negative number\n");
   elif (i <= 0) == false:
      write("%d is a positive number\n");
   end
end
```

2. Construct a list with the squares of the first 10 integers
   (a) using a `for` statement, and
   (b) using a `while` statement.
3. Write a program that
   (a) makes a list with the first 50 prime numbers.
   (b) Extend the program with computing the sum of the first 7 prime numbers.
   (c) Extend the program with computing the sum of the last 11 prime numbers.

Chapter 4

Functions

In a model, computations must be performed to process the information that is sent around. Short and simple calculations are written as assignments between the other statements, but for longer computations or computations that are needed at several places in the model, a more encapsulated environment is useful, a *function*. In addition, the language comes with a number of built-in functions, such as `size` or `empty` on container types. An example:

```
func real mean(list int xs):
   int sum;
   for x in xs:
      sum = sum + x
   end;
   return sum / size(xs)
end
```

The `func` keyword indicates it is a function. The name of the function is just before the opening parenthesis, in this example '`mean`'. Between the parentheses, the input values (the *formal parameters*) are listed. In this example, there is one input value, namely `list int` which is a list of integers. Parameter name `xs` is used to refer to the input value in the body of the function. Between `func` and the name of the function is the type

of the computation result, in this case, a `real` value. In other words, this `mean` function takes a list of integers as input, and produces a `real` value as result.

The colon at the end of the first line indicates the start of the computation. Below it are new variable declarations (`int sum`), and statements to compute the value, the *function algorithm*. The `return` statement denotes the end of the function algorithm. The value of the expression behind it is the result of the calculation. This example computes and returns the mean value of the integers of the list.

Use of a function (*application* of a function) is done by using its name, followed by the values to be used as input (the *actual parameters*). The above function can be used like

```
m = mean([1, 3, 5, 7, 9])
```

The actual parameter of this function application is `[1, 3, 5, 7, 9]`. The function result is `(1 + 3 + 5 + 7 + 9)/5` (which is `5.0`), and variable `m` becomes `5.0`.

A function is a mathematical function: the result of a function is the same for the same values of input parameters. A function has no *side-effect*, and it cannot access variables outside the body. For example, it cannot access `time` (explained in Chapter 10) directly, it has to be passed in through the parameter list.

A function that calculates the sign of a real number, is:

```
func int sign(real r):
   if r < 0:
      return -1
   elif r = 0:
      return  0
   end;
   return  1
end
```

The sign function returns: if `r` is smaller than zero, the value minus one; if `r` equals zero, the value zero; and if `r` is greater than zero, the value one.

The computation in a function ends when it encounters a `return` statement. The `return 1` at the end is therefore only executed when both `if` conditions are false.

4.1 Sorted lists

The language allows *recursive* functions as well as *higher-order* functions. Explaining them in detail is beyond the scope of this tutorial, but these functions are useful for making and maintaining sorted lists. Such a sorted list is useful for easily getting the smallest (or largest) item from a collection, for example the order with the nearest deadline.

To sort a list, the first notion that has to be defined is the desired order, by making a function of the following form:

```
func bool decreasing(int x, y):
   return x >= y
end
```

The function is called *predicate function*. It takes two values from the list (two integers in this case), and produces a boolean value, indicating whether the parameters are in the right order. In this case, the function returns `true` when the first parameter is larger or equal than the second parameter, that is, larger values must be before smaller values (for equal values, the order does not matter). This results in a list with decreasing values.

The requirements on any predicate function `f` are:

1. If `x` $\neq$ `y`, either `f(x, y)` must hold, or `f(y, x)` must hold, but not both. (Unequal values must have a unique order.)

2. If `x` $=$ `y`, both `f(x, y)` and `f(y, x)` must hold. (Equal values can be placed in arbitrary order.)

3. For values `x`, `y`, and `z`, if `f(x, y)` holds and `f(y, z)` holds (that is `x` $\geq$ `y` and `y` $\geq$ `z`), then `f(x, z)` must also hold (that is, `x` $\geq$ `z` should

also be true). The order between `x` and `z` must be stable, even when you compare with an intermediate value `y` between `x` and `z`.

These requirements hold for functions that test on `<=` or `>=` like above.

Sort

The first use of such a predicate function is for sorting a list. For example list `[3, 8, 7]` is sorted decreasingly (larger numbers before smaller numbers).

```
ys = sort([3, 8, 7], decreasing)
```

Sorting is done with the *sort* function, it takes two parameters, the list to sort, and the predicate *function*. (There are no parentheses '`()`' behind `decreasing`!) The value of list `ys` becomes `[8, 7, 3]`.

Another sorting example is a list of type `tuple(int number, real slack)`, where field `number` denotes the number of an item, and field `slack` denotes the slack time of the item. The list should be sorted in ascending order of the slack time. The type of the item is:

```
type item = tuple(int number, real slack);
```

The predicate function `spred` is defined by:

```
func bool spred(item x, y):
   return x.slack <= y.slack
end
```

Function `spred` delivers `true` if the two elements are in increasing order in the list, otherwise `false`. Note, the parameters of the function are of type `item`. Given a variable `ps` equal to `[(7, 21.6), (5, 10.3), (3, 35.8)]`. The statement denoting the sorting is:

```
qs = sort(ps, spred)
```

variable `qs` becomes `[(5, 10.3), (7, 21.6), (3, 35.8)]`.

Insert

Adding a new value to a sorted list is the second use of higher-order functions. The simplest approach would be to add the new value to the head or rear of the list, and sort the list again, but sorting an almost sorted list is very expensive. It is much faster to find the right position in the already sorted list, and insert the new value at that point. This function also exists, and is named `insert`. An example is (assume `xs` initially contains `[3,8]`):

```
xs = insert(xs, 7, increasing)
```

where `increasing` is

```
func bool increasing(int x, y): return x <= y end
```

assign the result `[3,7,8]` as new value to `xs`, `7` is inserted in the list.

Chapter 5

Input and output

A model communicates with the outside world, e.g. screen and files, by the use of read statements for input of data, and write statements for output of data.

5.1 `read` functions

Data can be read from the command line or from a file by *read* functions. A read function requires a type value for each parameter to be read. An example:

```
int i; string s;
i = read(int); s = read(string);
```

Two values, an integer value and a string value are read from the command line. On the command line the two values are typed:

```
1 "This is a string"
```

Variable `i` becomes `1`, and string `s` becomes `"This is a string"`. The double quotes are required! Parameter values are separated by a space or a tabular stop. Putting each value on a separate line also works.

Data also can be read from files. An example fragment:

```
file f;
int i; real r;
f = open("data_file", "r");
i = read(f, int); r = read(f, real);
close(f)
```

Before a file can be used, the file has to be declared, *and* the file has to be opened by statement `open`. Statement `open` has two parameters, the first parameter denotes the file name (as a string), and the second parameter describes the way the file is used. In this case, the file is opened in a read-only mode, denoted by string "r". If the file is no longer needed, the file is closed by the statement `close`, with one parameter, the variable of the file. If a file is still open after an experiment, the file is closed automatically before the program quits.

5.2 `write` statements

The *write* statement is used for for output of data to the screen of the computer. Data can also be written to a file. An example:

```
int i = 5;
write("i = %s", i)
```

In this example the text `i = 5` is written to the screen by the `write` statement. The `"i = %s"` string is called the *format string*. It defines what output is written. All 'normal' characters are copied as-is. The `%s` is not copied, it acts as a place holder for a value. In this case, it gets replaced by the value of `i`, the first parameter after the format string.

The `s` in the format string is a *format specifier*. It means 'print as string'. This works nicely in general, but for numeric values a little more control over the output is often useful. To this end, there are also format specifiers `d` (for integer numbers, and `f` for real numbers. An example:

```
int i = 5; real r = 3.14;
write("%4d/%f8.2", i, r)
```

This fragment has the effect that the values of `i` and `r` are written to the screen as follows:

```
5/    3.14
```

The value of `i` is written in `d` format, as `int` value, and the value of `r` is written in `f` format, as `real` value. The symbols `d` and `f` originate respectively from 'decimal', and 'floating point' numbers. The numbers `4` respectively `8.2` denote that the integer value is written `4` positions wide (that is, 3 spaces and a '5' character), and that the real value is written `8` positions wide, with `2` characters after the decimal point (that is, 4 spaces and the text '3.14').

A list of format specifiers is given in Table 5.1. The '`%s`' is a general

`%b`	boolean value (outputs `false` or `true`)
`%d`	integer
`%10d`	integer, at least 10 characters wide
`%f`	real
`%10f`	real, at least 10 characters wide
`%.4f`	real, 4 characters after the decimal point
`%10.4f`	real, at least 10 wide and 4 characters after the decimal point
`%s`	character string `s`, can also write other types of data
`%%`	the character `%`

Table 5.1: Format specifiers.

purpose specifier, you can write almost every type of data with it. For example

```
list dict(int:real) xs = [{1 : 5.3}];
write("%s", xs)
```

will output the contents of `xs`.

Finally, there are also a few special character sequences called *escape sequence* which allow to write characters like horizontal tab (which means

'jump to next tab position in the output'), or newline (which means 'go to the next line in the output') in a format string. An escape sequence consists of two characters. First a backslash character \, followed by a second character. The escape sequence are presented in Table 5.2. An

`\n`	new line
`\t`	horizontal tab
`\"`	the character "
`\\`	the character \

Table 5.2: Escape sequences.

example is:

```
int i = 5, j = 10; real r = 3.14;
write("%6d\t%d\n\t%.2f\n", i, j, r)
```

The result looks like

```
     5  10
        3.14
```

The value of `j` is written at the tab position, the output goes to the next line again at the first tab position, and outputs the value of `r`.

Data can be written to a file, analog to the read function. A file has to be defined first, and opened for writing before the file can be used. An example:

```
file f;
int i;
f = open("output_file", "w");
write(f, "%s", i); write(f, "%8.2f", r);
close(f)
```

A file, in this case `"output_file"` is used in write-only mode, denoted by the character `"w"`. Opening a file for writing destroys its old contents (if

the file already exists). In the write statement, the first parameter must be the file, and the second parameter must be the format string. After all data has been written, the file is closed by statement `close`. If the file is still open after execution of the program, the file is closed automatically.

Chapter 6

Modeling stochastic behavior

Many processes in the world vary a little bit each time they are performed. Setup of machines goes a bit faster or slower, patients taking their medicine takes longer this morning, more products are delivered today, or the quality of the manufactured product degrades due to a tired operator. Modeling such variations is often done with stochastic distributions. A distribution has a mean value and a known shape of variation. By matching the means and the variation shape with data from the system being modeled, an accurate model of the system can be obtained. The language has many stochastic distributions available, this chapter explains how to use them to model a system, and lists a few commonly used distributions. More information can be found in the reference manual.

The following fragment illustrates the use of the random distribution to model a dice. Each value of the six-sided dice is equally likely to appear. Every value having the same probability of appearing is a property of the integer uniform distribution, in this case using interval `[1, 7)` (inclusive on the left side, exclusive on the right side). The model is:

```
dist int dice = uniform(1,7);
int x, y;
```

```
x = sample dice;
y = sample dice;
writeln("x=%d, y=%d", x, y);
```

The variable `dice` is an integer distribution, meaning that values drawn from the distribution are integer numbers. It is assigned an uniform distribution. A throw of a dice is simulated with the *operator* `sample`. Each time `sample` is used, a new sample value is obtained from the distribution. In the fragment the dice is thrown twice, and the values are assigned to the variables `x`, and `y`.

6.1 Distributions

The language provides *constant*, *discrete* and *continuous* distributions. A discrete distribution is a distribution where only specific values can be drawn, for example throwing a dice gives an integer number. A continuous distribution is a distribution where a value from a continuous range can be drawn, for example assembling a product takes a positive amount of time. The constant distributions are discrete distributions that always return the same value. They are useful during the development of the model (see below).

Constant distributions

When developing a model with stochastic behavior, it is hard to verify whether the model behaves correctly, since the stochastic results make it difficult to predict the outcome of experiments. As a result, errors in the model may not be noticed, they hide in the noise of the stochastic results. One solution is to first write a model without stochastic behavior, verify that model, and then extend the model with stochastic sampling. Extending the model with stochastic behavior is however an invasive change that may introduce new errors. These errors are again hard to find due to the difficulties to predict the outcome of an experiment. The constant distributions aim to narrow the gap by reducing the amount of changes that need

to be done after verification.

With constant distributions, a stochastic model with sampling of distributions is developed, but the stochastic behavior is eliminated by temporarily using constant distributions. The model performs stochastic sampling of values, but with predictable outcome, and thus with predictable experimental results, making verification easier. After verifying the model, the constant distributions are replaced with the distributions that fit the mean value and variation pattern of the modeled system, giving a model with stochastic behavior. Changing the used distributions is however much less invasive, making it less likely to introduce new errors at this stage in the development of the model.

Constant distributions produce the same value `v` with every call of `sample`. There is one constant distribution for each type of sample value:

- `constant(bool v)`, a `bool` distribution.
- `constant(int v)`, an `int` distribution.
- `constant(real v)`, a `real` distribution.

An example with a constant distribution is

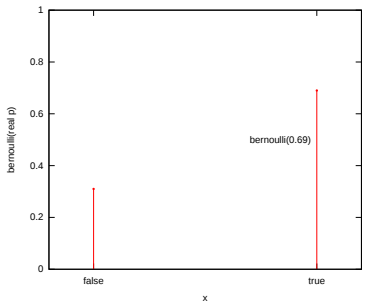
```
dist int u = constant(7);
```

This distribution returns the integer value `7` with each `sample u` operation.

Discrete distributions

Discrete distributions return values from a finite fixed set of possible values as answer. In Chi 3, there is one distribution that returns a boolean when sampled, and there are several discrete distributions that return an integer number.

Bernoulli



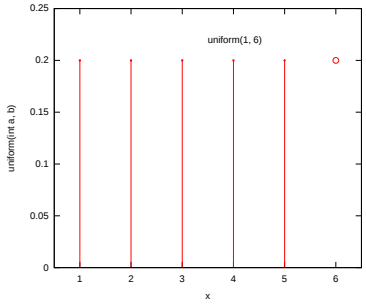
Discrete distribution that has two possible outcomes: `false` and `true`.

Function
`dist bool bernoulli(real p)`

Parameters
`p`: Chance on sampling `true`

mean p
variance $p(1-p)$

Discrete uniform



Discrete distribution that has several equally likely outcomes, the numbers $\{a, a+1, a+2, \ldots, b-2, b-1\}$. Note that b is **not included**.

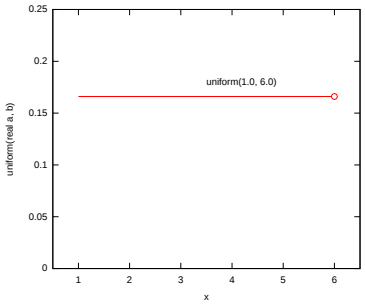
Function
`dist int uniform(int a, b)`

Parameters
`a`: Lower bound
`b`: Upper bound (exclusive!)
mean $(a+b-1)/2$
variance $((b-a)^2-1)/12$

Continuous distributions

Continuous distributions return a value from a continuous range.

Continuous uniform

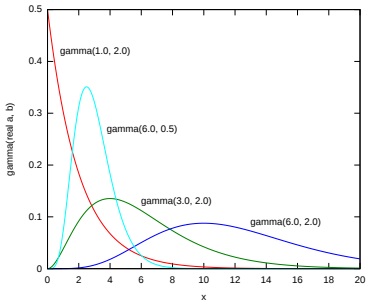


Continuous distribution with equal chance of sampling each value in the range $[a, b)$. Note that b is **not included**.

Function
`dist real uniform(real a, b)`

Parameters
`a`: Lower bound
`b`: Upper bound (exclusive!)
mean $(a + b)/2$
variance $(b - a)^2/12$

Gamma



Distribution which has either a decreasing probability function, or a peak.

Function
`dist real gamma(real a, b)`

Parameters
`a`: Shape parameter
`b`: Scale parameter
mean ab
variance ab^2

6.2 Simulating stochastic behavior

In this chapter, the mathematical notion of stochastic distribution is used to describe how to model stochastic behavior. Simulating a model with stochastic behavior at a computer is however not stochastic at all. Computer systems are deterministic machines, and have no notion of varying results.

A (pseudo-)random number generator is used to create stochastic results instead. It starts with an initial *seed*, an integer number (you can give one

at the start of the simulation). From this seed, a function creates a stream of 'random' values. When looking at the values there does not seem to be any pattern. It is not truly random however. Using the same seed again gives exactly the same stream of numbers. This is the reason to call the function a *pseudo*-random number generator (a true random number generator would never produce the exact same stream of numbers). A sample of a distribution uses one or more numbers from the stream to compute its value. The value of the initial seed thus decides the value of all samples drawn in the simulation.

While doing a simulation study, performing several experiments with the same initial seed invalidates the results, as it is equivalent to copying the outcome of a single experiment a number of times. On the other hand, when looking for the cause of a bug in the model, performing the exact same experiment is useful as outcomes of previous experiments should match exactly.

6.3 Exercises

1. According to the Chi 3 reference manual, for a gamma distribution with parameters (a, b), the mean equals ab.

   (a) Use a Chi 3 specification to verify whether this is true for at least 3 different pairs of a and b.

   (b) How many samples from the distribution are approximately required to determine the mean up to three decimals accurate?

2. Estimate the mean μ and variance σ^2 of a triangular distribution triangle$(1, 2, 5)$ by simulating 1000 samples. (Recall that the variance σ^2 of n samples x_i can be calculated by: $\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$.)

3. We would like to build a small game, called Higher or Lower. The computer picks a random integer number between 1 and 14. The player then has to predict whether the next number will be higher or lower. The computer picks the next random number and compares

the new number with the previous one. If the player guesses right his score is doubled. If the player guesses wrong, he looses all and the game is over. Try the following specification.

```
model HoL():
   dist int u = uniform(1, 15);
   int sc = 1;
   bool c = true;
   int new, oldval;
   string s;

   new = sample u;
   write("Your score is %d\n", sc);
   write("The computer drew %d\n", new);

   while c:
      writeln("(h)igher or (l)ower:\n");
      s = read(string);
      oldval = new;
      new = sample u;
      write("The computer drew %d\n", new);
      if new == oldval:
         c = false;
      else:
         c = (new > oldval) == (s == "h");
      end;

      if c:
         sc = 2 * sc;
      else:
         sc = 0;
      end;

      write("Your score is %d\n", sc)
```

```
    end;
    write("GAME OVER...\n")
end
```

(a) What is the begin score?

(b) What is the maximum end score?

(c) What happens, when the drawn sample is equal to the previous drawn sample?

(d) Extend this game specification with the possibility to stop.

Chapter 7

Processes

The language has been designed for modeling and analyzing systems with many components, all working together to obtain the total system behavior. Each component exhibits behavior over time. Sometimes they are busy making internal decisions, sometimes they interact with other components. The language uses a *process* to model the behavior of a component (the primary interest are the actions of the component rather than its physical representation). This leads to models with many processes working in *parallel* (also known as *concurrent* processes), interacting with each other.

Another characteristic of these systems is that the parallelism happens at different scales at the same time, and each scale can be considered to be a collection of co-operating parallel working processes. For example, a factory can be seen as a single component, it accepts supplies and delivers products. However, within a factory, you can have several parallel operating production lines, and a line consists of several parallel operating machines. A machine again consists of parallel operating parts. In the other direction, a factory is a small element in a supply chain. Each supply chain is an element in a (distribution) network. Depending on the area that needs to be analyzed, and the level of detail, some scales are precisely modeled, while others either fall outside the scope of the system or are modeled in

an abstract way.

In all these systems, the interaction between processes is not random, they understand each other and exchange information. In other words, they *communicate* with each other. The Chi 3 language uses *channels* to model the communication. A channel connects a sending process to a receiving process, allowing the sender to pass messages to the receiver. This chapter discusses parallel operating processes only, communication between processes using channels is discussed in Chapter 8.

As discussed above, a process can be seen as a single component with behavior over time, or as a wrapper around many processes that work at a smaller scale. The Chi 3 language supports both kinds of processes. The former is modeled with the statements explained in previous chapters and communication that will be explained in the next chapter. The latter (a process as a wrapper around many smaller-scale processes) is supported with the `run` statement.

7.1 A single process

The simplest form of processes is a model with one process:

```
proc P():
   write("Hello. I am a process.")
end

model M():
   run P()
end
```

Similar to a model, a process definition is denoted by the keyword `proc` (`proc` means process and does not mean procedure!), followed by the name of the process, here `P`, followed by an empty pair of parentheses '`()`', meaning that the process has no parameters. Process `P` contains one statement, a `write` statement to output text to the screen. Model `M` contains one

statement, a `run` statement to run a process. When simulating this model, the output is:

```
Hello. I am a process.
```

A `run` statement constructs a process from the process definition (it *instantiates* a process definition) for each of its arguments, and they start running. This means that the statements inside each process are executed. The `run` statement waits until the statements in its created processes are finished, before it ends itself.

To demonstrate, below is an example of a model with two processes:

```
proc P(int i):
   write("I am process. %d.\n", i)
end

model M():
   run P(1), P(2)
end
```

This model instantiates and runs two processes, `P(1)` and `P(2)`. The processes are running at the same time. Both processes can perform a `write` statement. One of them goes first, but there is no way to decide beforehand which one. (It may always be the same choice, it may be different on Wednesday, etc, you just don't know.) The output of the model is therefore either

```
I am process 1.
I am process 2.
```

or

```
I am process 2.
I am process 1.
```

After the two processes have finished their activities, the `run` statement in the model finishes, and the simulation ends.

An important property of statements is that they are executed *atomically*. It means that execution of the statement of one process cannot be interrupted by the execution of a statement of another process.

7.2 A process in a process

The view of a process being a wrapper around many other processes is supported by allowing to use the `run` statement inside a process as well. An example:

```
proc P():
   while true:
      write("Hello. I am a process.\n")
   end
end

proc DoubleP():
   run
      P(), P()
end

model M():
   run
      DoubleP()
end
```

The model instantiates and runs one process `DoubleP`. Process `DoubleP` instantiates and runs two processes `P`. The relevance becomes clear in models with a lot of processes. The concept of 'a process in a process' is very useful in keeping the model structured.

7.3 Many processes

Some models consist of many identical processes at a single level. The language has an `unwind` statement to reduce the amount of program text. A model with e.g. ten identical processes, and a different parameter value, is:

```
model MRun():
   run
      P(0), P(1), P(2), P(3), P(4),
      P(5), P(6), P(7), P(8), P(9)
end
```

An easier way to write this model is by applying the `unwind` statement inside `run` with the same effect:

```
model MP():
   run
      unwind j in range(10):
         P(j)
      end
end
```

Chapter 8

Channels

In Chapter 7 processes have been introduced. This chapter describes channels, denoted by the type `chan`. A channel connects two processes and is used for the transfer of data or just signals. One process is the sending process, the other process is the receiving process. Communication between the processes takes place instantly when both processes are willing to communicate, this is called *synchronous* communication.

8.1 A channel

The following example shows the sending of an integer value between two processes via a channel. Figure 8.1 shows the two processes `P` and `C`, connected by channel variable `a`. Processes are denoted by circles, and channels

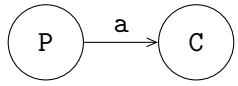


Figure 8.1: A producer and a consumer.

are denoted by directed arrows in the figure. The arrow denotes the direction of communication. Process `P` is the sender or producer, process `C` is

the receiver or consumer.

In this case, the producer sends a finite stream of integer values (5 numbers) to the consumer. The consumer receives these values and writes the values to the screen. The model is:

```
proc P(chan! int a):
   for i in range(5):
      a!i
   end
end

proc C(chan? int b):
   int x;
   while true:
      b?x;
      write("%d\n",x)
   end
end

model M():
   chan int a;
   run
      P(a), C(a)
end
```

The model instantiates processes `P` and `C`. The two processes are connected to each other via channel variable `a` which is given as actual parameter in the `run` statement. This value is copied into the local formal parameter `a` in process `P` and in formal parameter `b` inside process `C`.

Process `P` can send a value of type `int` via the actual channel parameter `a` to process `C`. In this case `P` first tries to send the value `0`. Process `C` tries to receive a value of type `int` via the actual channel parameter `a`. Both processes can communicate, so the communication occurs and the value `0` is sent to process `C`. The received value is assigned in process `C` to variable

x. The value of x is printed and the cycle starts again. This model writes the sequence 0, 1, 2, 3, 4 to the screen.

Above, process P constructs the numbers and sends them to process C. However, since it is known that the number sequence starts at 0 and increments by one each time, there is no actual need to transfer a number. Process C could also construct the number by itself after getting a signal (a 'go ahead') from process P. Such signals are called synchronization signals. They do not carry any data, they just synchronize actions between different processes.

The following example shows the use of synchronization signals between processes P and C. The connecting channel 'transfers' values of type void. The type void means that 'non-values' are sent and received; the type void is only allowed in combination with channels. The iconic model is given in the previous figure, Figure 8.1. The model is:

```
proc P(chan! void a):
   for i in range(5):
      a!    # No data is being sent
   end
end

proc C(chan? void b):
   int i;
   while true:
      b?;   # Nothing is being received
      write("%d\n", i);
      i = i + 1
   end
end

model M():
   chan void a;
   run
      P(a), C(a)
```

```
end
```

Process `P` sends a signal (and no value is sent), and process `C` receives a signal (without a value). The signal is used by process `C` to write the value of `i` and to increment variable `i`. The effect of the model is identical to the previous example: the numbers `0, 1, 2, 3, 4` appear on the screen.

8.2 Two channels

A process can have more than one channel, allowing interaction with several other processes.

The next example shows two channel variables, `a` and `b`, and three processes, generator `G`, server `S` and exit `E`. The iconic model is given in Figure 8.2. Process `G` is connected via channel variable `a` to process `S` and

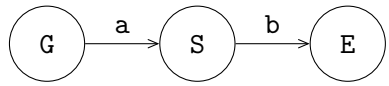


Figure 8.2: A generator, a server and an exit.

process `S` is connected via channel variable b to process `E`. The model is:

```
proc G(chan! int a):
   for x in range(5):
      a!x
   end
end

proc S(chan? int a; chan! int b):
   int x;
   while true:
      a?x; x = 2 * x; b!x
   end
end
```

```
proc E(chan int a):
   int x;
   while true:
      a?x;
      write("E %f\n", x)
   end
end

model M():
   chan int a,b;
   run
      G(a), S(a,b), E(b)
end
```

The model contains two channel variables `a` and `b`. The processes are connected to each other in model `M`. The processes are instantiated and run where the formal parameters are replaced by the actual parameters. Process `G` sends a stream of integer values `0, 1, 2, 3, 4` to another process via channel `a`. Process `S` receives a value via channel `a`, assigns this value to variable `x`, doubles the value of the variable, and sends the value of the variable via `b` to another process. Process `E` receives a value via channel `b`, assigns this value to the variable `x`, and prints this value. The result of the model is given by:

```
E   0
E   2
E   4
E   6
E   8
```

After printing this five lines, process `G` stops, process `S` is blocked, as well as process `E`, the model gets blocked, and the model ends.

8.3 More senders or receivers

Channels send a message (or a signal in case of synchronization channels) from one sender to one receiver. It is however allowed to give the same channel to several sender or receiver processes. The channel selects a sender and a receiver before each communication.

The following example gives an illustration, see Figure 8.3. Suppose

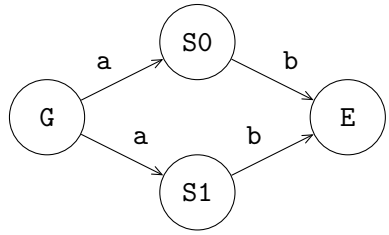


Figure 8.3: A generator, two servers and an exit.

that only `G` and `S0` want to communicate. The channel can select a sender (namely `G`) and a receiver (process `S0`), and let both processes communicate with each other. When sender `G`, and both receivers (`S0` and `S1`), want to communicate, the channel selects a sender (`G` as it is the only sender available to the channel), and a receiver (either process `S0` or process `S1`), and it lets the selected processes communicate with each other. This selection process is non-deterministic; a choice is made, but it is unknown how the selection takes place and it cannot be influenced. Note that a non-deterministic choice is different from a random choice. In the latter case, there are known probabilities of selecting a process.

Sharing a channel in this way allows to send data to receiving processes where the receiving party is not relevant (either server process will do). This way of communication is different from *broadcasting*, where both servers receive the same data value. Broadcasting is not supported by the Chi 3 language.

In case of two senders, `S0` and `S1`, and one receiver `E` the selection

process is the same. If one of the two servers `S` can communicate with exit `E`, communication between that server and the exit takes place. If both servers can communicate, a non-deterministic choice is made.

Having several senders and several receivers for a single channel is also handled in the same manner. A non-deterministic choice is made for the sending process and a non-deterministic choice is made for the receiving process before each communication.

To communicate with several other processes but without non-determinism, unique channels must be used.

8.4 Notes

- The direction in channels, denoted by `?` or `!`, may be omitted. By leaving it out, the semantics of the parameters becomes less clear (the direction of communication has to be derived from the process code).

- There are a several ways to name channels:

    1. Start naming formal channel parameters in each new process with `a`, `b`, etc. The actual names follow from the figure. This convention is followed in this chapter. For small models this convention is easy and works well, for complicated models this convention can be error-prone.
    2. Use the actual names of the channel parameters in the figures as formal names in the processes. Start naming in figures with `a`, `b`, etc. This convention works well, if both figure and code are at hand during the design process. If many processes have sub-processes, this convention does not really work.
    3. Use unique names for the channel parameters for the whole model, and for all sub-systems, for example a channel between processes `A` and `B` is named `a2b` (the lower-case name of the sending process, followed by `2`, denoting 'to', and the lower-case name of the receiving process).

In this case the formal and actual parameters can be in most cases the same. If many identical processes are used, this convention does not really work.

In the text all three conventions are used, depending on the structure of the model.

8.5 Exercises

1. Given is the specification of process `P` and model `PP`.

```
proc P(chan int a, b):
   int x;

   while true:
      a?x;
      x = x + 1;
      write("%d\n", x);
      b!x
   end
end

model PP():
   chan int a, b;

   run P(a,b), P(b,a)
end
```

   (a) Study this specification.

   (b) Why does the model terminate immediately?

2. Six children have been given the assignment to perform a series of calculations on the numbers $0, 1, 2, 3, \ldots, 9$, namely add 2, multiply by 3, multiply by 2, and add 6 subsequently. They decide to split up

the calculations and to operate in parallel. They sit down at a table next to each other. The first child, the reader R, reads the numbers $0, 1, 2, 3, \ldots, 9$ one by one to the first calculating child C_1. Child C_1 adds 2 and tells the result to its right neighbour, child C_2. After telling the result to child C_2, child C_1 is able to start calculating on the next number the reader R tells him. Children C_2, C_3, and C_4 are analogous to child C_1; they each perform a different calculation on a number they hear and tell the result to their right neighbour. At the end of the table the writer W writes every result he hears down on paper. Figure 8.4 shows a schematic drawing of the children at the table.

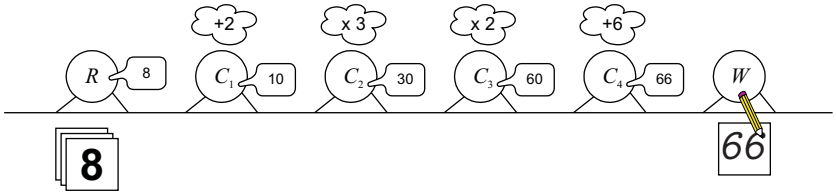


Figure 8.4: Six children working in parallel

(a) Finish the specification for the reading child R, that reads the numbers 0 till 9 one by one.

```
proc R(...):
   int i;

   while i < 10:
      ...;
      ...
   end
end
```

(b) Specify the parameterized process C_{add} that represents the children C_1 and C_4, who perform an addition.

(c) Specify the parameterized process C_{mul} that represents the children C_2 and C_3, who perform a multiplication.

(d) Specify the process W representing the writing child. Write each result to the screen separated by a new line.

(e) Make a graphical representation of the model `SixChildren` that is composed of the six children.

(f) Specify the model `SixChildren`. Simulate the model.

Chapter 9

Buffers

In the previous chapter, a production system was discussed that passes values from one process to the next using channels, in a synchronous manner. (Sender and receiver perform the communication at exactly the same moment in time, and the communication is instantaneous.) In many systems however, processes do not use synchronous communication, they use *asynchronous* communication instead. Values (products, packets, messages, simple tokens, and so on) are sent, temporarily stored in a buffer, and then received.

In fact, the decoupling of sending and receiving is very important, it allows compensating temporarily differences between the number of items that are sent and received. (Under the assumption that the receiver is fast enough to keep up with the sender in general, otherwise the buffer will grow forever or overflow.)

For example, consider the exchange of items from a producer process `P` to a consumer process `C` as shown in Figure 9.1. In the unbuffered situation, both processes communicate at the same time. This means that when one process is (temporarily) faster than the other, it has to wait for the other process before communication can take place. With a buffer in-between, the producer can give its item to the buffer, and continue with its work. Likewise, the consumer can pick up a new item from the buffer at any later

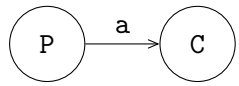


Figure 9.1: A producer and a consumer.

time (if the buffer has items).

In Chi 3, buffers are not modeled as channels, they are modeled as additional processes instead. The result is shown in Figure 9.2. The producer

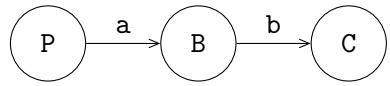


Figure 9.2: A producer and a consumer, with an additional buffer process.

sends its items synchronously (using channel `a`) to the buffer process. The buffer process keeps the item until it is needed. The consumer gets an item synchronously (using channel `b`) from the buffer when it needs a new item (and one is available).

In manufacturing networks, buffers, in combination with servers, play a prominent role, for buffering items in the network. Various buffer types exist in these networks: buffers can have a finite or infinite capacity, they have a input/output discipline, for example a first-out queuing discipline or a priority-based discipline. Buffers can store different kinds of items, for example, product-items, information-items, or a combination of both. Buffers may also have sorting facilities, etc.

In this chapter some buffer types are described, and with the presented concepts numerous types of buffer can be designed by the engineer. First a simple buffer process with one buffer position is presented, followed by more advanced buffer models. The producer and consumer processes are not discussed in this chapter.

9.1 A one-place buffer

A buffer usually has a receiving channel and a sending channel, for receiving and sending items. A buffer, buffer `B1`, is presented in Figure 9.3.

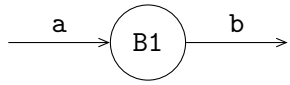


Figure 9.3: A 1-place buffer.

The simplest buffer is a one-place buffer, for buffering precisely one item. A one-place buffer can be defined by:

```
proc B1(chan? item a; chan! item b):
   item x;
   while true:
      a?x; b!x
   end
end
```

where `a` and `b` are the receiving and sending channels. Item `x` is buffered in the process. A buffer receives an item, stores the item, and sends the item to the next process, if the next process is willing to receive the item. The buffer is not willing to receive a second item, as long as the first item is still in the buffer.

A two-place buffer can be created, by using the one-place buffer process twice. A two-place buffer is depicted in Figure 9.4: A two-place buffer is

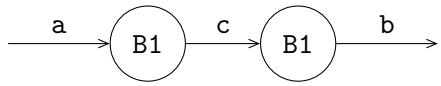


Figure 9.4: A 2-place buffer.

defined by:

```
proc B2(chan? item a; chan! item b):
   chan item c;
   run
      B1(a, c), B1(c, b)
end
```

where two processes `B1` buffer maximal two items. If each process `B1` contains an item, a third item has to wait in front of process `B2`. This procedure can be extended to create even larger buffers. Another, more preferable manner however, is to describe a buffer in a single process by using a *select* statement and a list for storage of the items. Such a buffer is discussed in the next section.

9.2 A single process buffer

An informal description of the process of a buffer, with an arbitrary number of stored items, is the following:

- If the buffer has space for an item, *and* can receive an item from another process via channel `a`, the buffer process receives that item, and stores the item in the buffer.
- If the buffer contains at least one item, *and* the buffer can send that item to another process via channel `b`, the buffer process sends that item, and removes that item from the buffer.
- If the buffer can both send and receive a value, the buffer process selects one of the two possibilities (in a non-deterministic manner).
- If the buffer can not receive an item, and can not send an item, the buffer process waits.

Next to the sending and receiving of items (to and from the buffer process) is the question of how to order the stored items. A common form is the *first-in first-out* (fifo) queuing discipline. Items that enter the buffer first

(first-in) also leave first (first-out), the order of items is preserved by the buffer process.

In the model of the buffer, an (ordered) list of type `item` is used for storing the received items. New item `x` is added at the rear of list `xs` by the statement:

```
xs = xs + [x]
```

The first item of the list is sent, and then deleted with:

```
xs = xs[1:]
```

An alternative solution is to swap the function of the rear and the front, which can be useful some times.

The statement to monitor several channels at the same time is the `select` statement. The syntax of the `select` statement, with two alternatives, is:

```
select
   boolean_expression_1, communication statement_1:
      statement_list_1
alt
   boolean_expression_2, communication statement_2:
      statement_list_2
...
end
```

There has to be at least one alternative in a select statement. The statement waits, until for one of the alternatives the `boolean_expression` holds *and* communication using the `communication statement` is possible. (When there are several such alternatives, one of them is non-deterministically chosen.) For the selected alternative, the communication statement is executed, followed by the statements in the `statement_list` of the alternative.

The above syntax is the most generic form, the `boolean_expression` may be omitted when it always holds, or the `communication statement` may be omitted when there is no need to communicate. The '`,`' also disappears then. (Omitting both the boolean expression and the communication

statement is not allowed.) Similarly, when the `statement_list` is empty or just `pass`, it may be omitted (together with the '`:`' in front of it).

The description (in words) of the core of the buffer, from page 72, is translated in code, by using a `select` statement:

```
select
   size(xs) < N, a?x:
      xs = xs + [x]
alt
   size(xs) > 0, b!xs[0]:
      xs = xs[1:]
end
```

In the first alternative, it is stated that, if the buffer is not full, and the buffer can receive an item, an item is received, and that item is added to the rear of the list. In the second alternative, it is stated that, if the buffer contains at least one item, and the buffer can send an item, the first item in the list is sent, and the list is updated. Please keep in mind that both the condition must hold and the communication must be possible *at the same moment*.

The complete description of the buffer is:

```
proc B(chan? item a; chan! item b):
   list item xs; item x;
   while true:
      select
         size(xs) < N, a?x:
            xs = xs + [x]
      alt
         size(xs) > 0, b!xs[0]:
            xs = xs[1:]
      end
   end
end
```

Instead of boolean expression `size(xs) > 0`, expression `not empty(xs)` can be used, where `empty` is a function yielding `true` if the list is empty, otherwise `false`. In case the capacity of the buffer is infinite, expression `size(xs) < N` can be replaced by `true`, or even omitted (including the comma). A buffer with infinite capacity can be written as:

```
proc B(chan? item a; chan! item b):
   list item xs; item x;
   while true:
      select
         a?x:
            xs = xs + [x]
      alt
         not empty(xs), b!xs[0]:
            xs = xs[1:]
      end
   end
end
```

A first-in first-out buffer is also called a *queue*, while a first-in last-out buffer (*lifo* buffer), is called a *stack*. A description of a lifo buffer is:

```
proc B(chan? item a; chan! item b):
   list item xs; item x;
   while true:
      select
         a?x:
            xs = [x] + xs
      alt
         not empty(xs), b!xs[0]:
            xs = xs[1:]
      end
   end
end
```

The buffer puts the last received item at the head of the list, and gets the first item from the list. An alternative is to put the last item at the rear of the list, and to get the last item from the list.

9.3 A token buffer

In the next example, signals are buffered instead of items. The buffer receives and sends 'empty' items or *tokens*. Counter variable `w` of type `int` denotes the difference of the number of tokens received and the number of tokens sent. If the buffer receives a token, counter `w` is incremented; if the buffer sends a token, counter `w` is decremented. If the number of tokens sent is less than the number of tokens received, there are tokens in the buffer, and `w > 0`. A receiving channel variable `a` of type `void` is defined for receiving tokens. A sending channel variable `b` of type `void` is defined for sending tokens. The buffer becomes:

```
proc B(chan? void a; chan! void b):
   int w;
   while true:
      select
         a?:
            w = w + 1
      alt
         w > 0, b!:
            w = w - 1
      end
   end
end
```

Note: Variables of type `void` do not exist. Type `void` only can be used in combination with channels.

9.4 A priority buffer

A buffer for items with different priority is described in this section. An item has a high priority or a normal priority. Items with a high priority should leave the buffer first.

An item is a tuple with a field `prio`, denoting the priority, `0` for high priority, and `1` for normal priority:

```
type item = tuple(...; int prio);
```

For the storage of items, two lists are used: a list for high priority items and a list for normal priority items. The two lists are described by a list with size two:

```
list(2) list item xs;
```

Variable `xs[0]` contains the high priority items, `xs[1]` the normal priority items. The first item in the high priority list is denoted by `xs[0][0]`, etc.

In the model the received items are, on the basis of the value of the `prio`-field in the item, stored in one of the two lists: one list for 'high' items and one list for 'normal' items. The discipline of the buffer is that items with a high priority leave the buffer first. The model is:

```
proc BPrio(chan? item a; chan! item b):
   list(2) list item xs; item x;
   while true:
      select
         a?x:
            xs[x.prio] = xs[x.prio] + [x]
      alt
         not empty(xs[0]), b!xs[0][0]:
            xs[0] = xs[0][1:]
      alt
         empty(xs[0]) and not empty(xs[1]), b!xs[1][0]:
            xs[1] = xs[1][1:]
      end
```

```
        end
    end
```

The buffer has two lists `xs[0]` and `xs[1]`. Received items `x` are stored in `xs[x.prio]` by the statement `xs[x.prio] = xs[x.prio] + [x]`.

If the list high priority items (`xs[0]`) is not empty, items with high priority are sent. The first element in list `xs[0]` is element `xs[0][0]`. If there are no high priority items (list `xs[0]` is empty), and there are normal priority items (list `xs[1]` is not empty), the first element of list `xs[1]`, element `xs[1][0]`, is sent.

Note that the order of the alternatives in the select statement does not matter, every alternative is treated in the same way.

9.5 Exercises

1. To study product flow to and from a factory, a setup as shown in Figure 9.5 is created. `F` is the factory being studied, generator `G`

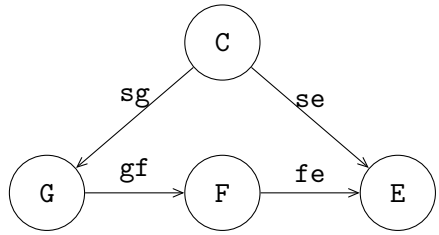


Figure 9.5: A controlled factory.

sends products into the factory, and exit process `E` retrieves finished products. The factory is tightly controlled by controller `C` that sends a signal to `G` or `E` before a product may be moved. The model is as follows:

```
proc G(chan! int a; chan? void sg):
   for i in range(10):
```

```
      sg?;
      a!i;
   end
end

proc F(chan? int a; chan! int b):
   ...
end

proc E(chan? int a; chan? void se):
   int x;
   while true:
      se?;
      a?x;
      write("E received %d\n", x);
   end
end

proc C(chan! void sg, se; int low, high):
   int count;
   while true:
      while count < high:
         sg!;
         count = count + 1;
      end
      while count > low:
         se!;
         count = count - 1;
      end
   end
end

model M():
   chan void sg, se;
```

```
   chan int gf, fe;
   run
      C(sg, se, 0, 1),
      G(gf, sg), F(gf, fe), E(fe, se);
end
```

The number of products inserted by the generator has been limited to allow for manual inspection of results.

(a) As a model of the factory, use a FIFO buffer process. Run the simulation, and check whether all products are received by the exit process.

(b) Change the control policy to `low = 1` and `high = 4`. Predict the outcome, and verify with simulation.

(c) The employees of the factory propose to stack the products in the factory to reduce the amount of space needed for buffering. Replace the factory process with a LIFO buffer process, run the experiments again, first with `low = 0` and `high = 1` and then with `low = 1` and `high = 4`.

(d) You will notice that some products stay in the factory forever. Why does that happen? How should the policy be changed to ensure all products eventually leave the factory?

Chapter 10

Servers with time

A manufacturing line contains machines and/or persons that perform a sequence of tasks, where each machine or person is responsible for a single task. The term *server* is used for a machine or a person that performs a task. Usually the execution of a task takes time, e.g. a drilling process, a welding process, the set-up of a machine. In this chapter we introduce the concept of *time*, together with the *delay* statement.

Note that here 'time' means the simulated time inside the model. For example, assume there are two tasks that have to be performed in sequence in the modeled system. The first task takes three hours to complete, the second task takes five hours to complete. These amounts of time are specified in the model (using the delay statement, as will be explained below). A simulation of the system should report 'It takes eight hours from start of the first task to finish of the second task'. However, it generally does not take eight hours to compute that result, a computer can calculate the answer much faster. When an engineer says "I had to run the system for a year to reach steady-state", he means that time inside the model has progressed a year.

10.1 The clock

The variable `time` denotes the current time in a model. It is a *global* variable, it can be used in every `model` and `proc`. The time is a variable of type `real`. Its initial value is `0.0`. The variable is updated automatically by the model, it cannot be changed by the user. The unit of the time is however determined by the user, that is, you define how long 1 time unit of simulated time is in the model.

The value of variable `time` can be retrieved by reading from the `time` variable:

```
t = time
```

The meaning of this statement is that the current time is copied to variable `t` of type real.

A process delays itself to simulate the processing time of an operation with a *delay* statement. The process postpones or suspends its own actions until the delay ends.

For example, suppose a system has to perform three actions, each action takes 45 seconds. The unit of time in the model is one minute (that is, progress of the modeled time by one time unit means a minute of simulated time has passed). The model looks like

```
proc P():
   for i in range(3):
      write("i = %d, time = %f\n", i, time);
      delay 0.75
   end
end

model M():
   run P()
end
```

An action takes 45 seconds, which is `0.75` time units. The `delay 0.75` statement represents performing the action, the process is suspended until 0.75 units of time has passed.

The simulation reports:

```
i = 0, time = 0.000000
i = 1, time = 0.750000
i = 2, time = 1.500000
All processes finished at time 2.25
```

The three actions are done in 2.25 time units (2.25 minutes).

10.2 Servers with time

Adding time to the model allows answering questions about time, often performance questions ('how many products can I make in this situation?'). Two things are needed:

- Servers must model use of time to perform their task.
- The model must perform measurements of how much time passes.

By extending models of the servers with time, time passes while tasks are being performed. Time measurements then give non-zero numbers (servers that can perform actions instantly result in all tasks being done in one moment of time, that is 0 time units have passed between start and finish). Careful analysis of the measurements should yields answers to questions about time.

In this chapter, adding of passing time in a server and how to embed time measurements in the model is explained. The first case is a small production line with a deterministic server (its task takes a fixed amount of time), while the second case uses stochastic arrivals (the moment of arrival of new items varies), and a stochastic server instead (the duration of the task varies each time). In both cases, the question is what the flow time of an item is (the amount of time that a single item is in the system), and what the throughput of the entire system is (the number of items the production line can manufacture per time unit).

A deterministic system

The model of a deterministic system consists of a deterministic generator, a deterministic server, and an exit process. The line is depicted in Figure 10.1. Generator process `G` sends items, with constant inter-arrival time `ta`, via

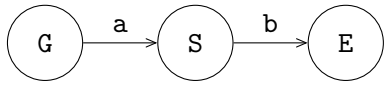


Figure 10.1: Generator `G`, server `S`, and exit `E`.

channel `a`, to server process `S`. The server processes items with constant processing time `ts`, and sends items, via channel `b`, to exit process `E`.

An item contains a real value, denoting the creation time of the item, for calculating the throughput of the system and flow time (or sojourn time) of an item in the system. The generator process creates an item (and sets its creation time), the exit process `E` writes the measurements (the moment in time when the item arrives in the exit process, and its creation time) to the output. From these measurements, throughput and flow time can be calculated.

Model `M` describes the system:

```
type item = real;

model M(real ta, ts; int N):
   chan item a, b;
   run
      G(a, ta),
      S(a, b, ts),
      E(b, N)
end
```

The `item` is a real number for storing the creation time. Parameter `ta` denotes the inter-arrival time, and is used in generator `G`. Parameter `ts`

denotes the server processing time, and is used in server `S`. Parameter `N` denotes the number of items that must flow through the system to get a good measurement.

Generator `G` has two parameters, channel `a`, and inter-arrival time `ta`. The description of process `G` is given by:

```
proc G(chan! item a; real ta):
   while true:
      a!time; delay ta
   end
end
```

Process `G` sends an item, with the current time, and delays for `ta`, before sending the next item to server process `S`.

Server `S` has three parameters, receiving channel `a`, sending channel `b`, and server processing time `ts`:

```
proc S(chan? item a; chan! item b; real ts):
   item x;
   while true:
      a?x; delay ts; b!x
   end
end
```

The process receives an item from process `G`, processes the item during `ts` time units, and sends the item to exit process `E`.

Exit `E` has two parameters, receiving channel `a` and the length of the experiment `N`:

```
proc E(chan item a; int N):
   item x;
   for i in range(N):
      a?x; write("%f, %f\n", time, time - x)
   end
end
```

The process writes current time `time` and item flow time `time - x` to the screen for each received item. Analysis of the measurements will show that the system throughput equals $1/$`ta`, and that the item flow time equals `ts` (if `ta` $\geq$ `ts`).

A stochastic system

In the next model, the generator produces items with an exponential inter-arrival time, and the server processes items with an exponential server processing time. To compensate for the variations in time of the generator and the server, a buffer process has been added. The model is depicted in Figure 10.2.

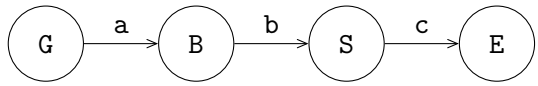


Figure 10.2: Generator `G`, buffer `B`, server `S`, and exit `E`.

Type `item` is the same as in the previous situation. The model runs the additional buffer process:

```
model M(real ta, ts; int N):
   chan item a, b, c;
   run
      G(a, ta),
      B(a, b),
      S(b, c, ts),
      E(c, N)
end
```

Generator `G` has two parameters, channel variable `a`, and variable `ta`, denoting the mean inter-arrival time. An `exponential` distribution is used for deciding the inter-arrival time of new items.

```
proc G(chan item a; real ta):
   dist real u = exponential(ta);
   while true:
      a!time; delay sample u
   end
end
```

The process sends a new item to the buffer, and delays `sample u` time units. Buffer process `B` is a fifo buffer with infinite capacity, as described on Page 75. Server `S` has three parameters, channel variables `a` and `b`, for receiving and sending items, and a variable for the average processing time `ts`.

```
proc S(chan item a, b; real ts):
   dist real u = exponential(ts);
   item x;
   while true:
      a?x; delay sample u; b!x
   end
end
```

An `exponential` distribution is used for deciding the processing time. The process receives an item from process `G`, processes the item during `sample u` time units, and sends the item to exit process `E`.

Exit process `E` is the same as in the previous case (see Page 85). In this case the throughput of the system also equals 1/`ta`, and the *mean flow* can be obtained by doing an experiment and analysis of the resulting measurements (for `ta` > `ts`).

10.3 Two servers

In this section two different types of systems are shown: a serial and a parallel system. In a serial system the servers are positioned after each other, in a parallel system the servers are operating in parallel. Both systems use a stochastic generator, and stochastic servers.

Serial system

The next model describes a *serial* system, where an item is processed by one server, followed by another server. The generator and the servers are decoupled by buffers. The model is depicted in Figure 10.3.

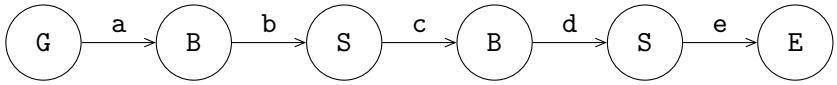


Figure 10.3: A generator, two buffers, two servers, and an exit.

The model can be described by:

```
model M(real ta, ts; int N):
   chan item a, b, c, d, e;
   run
      G(a, ta),
      B(a, b), S(b, c, ts),
      B(c, d), S(d, e, ts),
      E(e, N)
end
```

The various processes are equal to those described in the previous section, Section 10.2.

Parallel systems

In a parallel system the servers are operating in parallel. Having several servers in parallel is useful for enlarging the processing capacity of the task being done, or for reducing the effect of break downs of servers (when a server breaks down, the other server continues with the task for other items). Figure 10.4 depicts the system. Generator process `G` sends items via `a` to buffer process `B`, and process `B` sends the items in a first-in first-out manner to the servers `S`. Both servers send the processed items to the exit process `E` via channel `c`. The inter-arrival time and the two process times

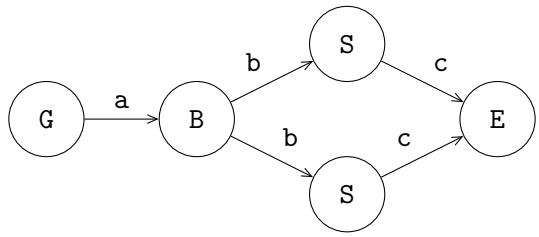


Figure 10.4: A model with two parallel servers.

are assumed to be stochastic, and exponentially distributed. Items can pass each other, due to differences in processing time between the two servers.

If a server is free, and the buffer is not empty, an item is sent to a server. If both servers are free, one server will get the item, but which one cannot be determined beforehand. (How long a server has been idle is not taken into account.) The model is described by:

```
model M(real ta, ts; int N):
   chan item a, b, c;
   run
      G(a, ta),
      B(a, b),
      S(b, c, ts), S(b, c, ts),
      E(c, N)
end
```

To control which server gets the next item, each server must have its own channel from the buffer. In addition, the buffer has to know when the server can receive a new item. The latter is done with a 'request' channel, denoting that a server is free and needs a new item. The server sends its own identity as request, the requests are administrated in the buffer. The model is depicted in Figure 10.5. In this model, the servers 'pull' an item through the line. The model is:

```
model M(real ta, ts; int N):
```

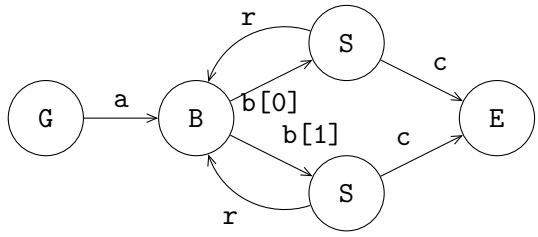


Figure 10.5: A model with two parallel requesting servers.

```
   chan item a; list(2) chan item b; chan item c;
   chan int r;
   run
      G(a, ta),
      B(a, b, r),
      unwind j in range(2):
         S(b[j], c, r, ts, j)
      end,
      E(c, N)
end
```

In this model, an `unwind` statement is used for the initialization and running of the two servers. Via channel `r` an integer value, `0` or `1`, is sent to the buffer.

The items received from generator `G` are stored in list `xs`, the requests received from the servers are stored in list `ys`. The items and requests are removed form their respective lists in a first-in first-out manner. Process `B` is defined by:

```
proc B(chan? item a; list chan! item b; chan? int r):
   list item xs; item x;
   list int ys; int y;
   while true:
      select
```

```
            a?x:
               xs = xs + [x]
         alt
            r?y:
               ys = ys + [y]
         alt
            not empty(xs) and not empty(ys),
              b[ys[0]]!xs[0]:
              xs = xs[1:]; ys = ys[1:]
         end
      end
   end
```

If, there is an item present, *and* there is a server demanding for an item, the process sends the first item to the longest waiting server. The longest waiting server is denoted by variable `ys[0]`. The head of the item list is denoted by `xs[0]`. Assume the value of `ys[0]` equals `1`, then the expression `b[ys[0]]!xs[0]`, equals `b[1]!xs[0]`, indicates that the first item of list `xs`, equals `xs[0]`, is sent to server `1`.

The server first sends a request via channel `r` to the buffer, and waits for an item. The item is processed, and sent to exit process `E`.

```
   proc S(chan? item b; chan! item c;
          chan! int r; real ts; int k):
      dist real u = exponential(ts);
      item x;
      while true:
         r!k;
         b?x;
         delay sample u;
         c!x
      end
   end
```

10.4 Assembly

In assembly systems, components are assembled into bigger components. These bigger components are assembled into even bigger components. In this way, products are built, e.g. tables, chairs, computers, or cars. In this section some simple assembly processes are described. These systems illustrate how assembling can be performed: in industry these assembly processes are often more complicated.

An assembly work station for two components is shown in Figure 10.6. The assembly process server `S` is preceded by buffers. The server receives

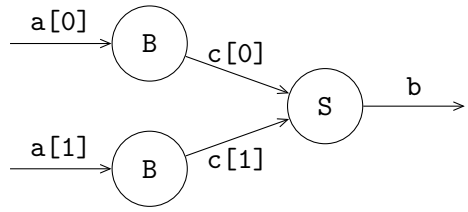


Figure 10.6: Assembly for two components.

an item from each buffer `B`, before starting assembly. The received items are assembled into one new item, a list of its (sub-)items. The description of the assembly server is:

```
proc S(list chan? item c, chan! list item b):
   list(2) item v;
   while true:
      select
         c[0]?v[0]: c[1]?v[1]
      alt
         c[1]?v[1]: c[0]?v[0]
      end
      b!v
   end
```

```
end
```

The process takes a list of channels `c` to receive items from the preceding buffers. The output channel `b` is used to send the assembled component away to the next process.

First, the assembly process receives an item from both buffers. All buffers are queried at the same time, since it is unknown which buffer has components available. If the first buffer reacts first, and sends an item, it is received with channel `c[0]` and stored in `v[0]` in the first alternative. The next step is then to receive the second component from the second buffer, and store it (`c[1]?v[1]`). The second alternative does the same, but with the channels and stored items swapped.

When both components have been received, the assembled product is sent away.

A generalized assembly work station for `n` components is depicted in Figure 10.7.

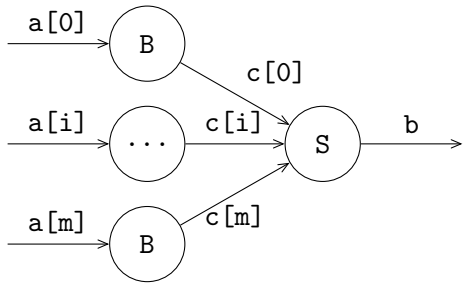


Figure 10.7: Assembly for `n` components, with `m = n - 1`.

The entire work station (the combined buffer processes and the assembly server process) is described by:

```
proc W(list chan? item a; chan! list item b):
   list(size(a)) chan item c;
   run
```

```
        unwind i in range(size(a)):
           B(a[i], c[i])
        end,
        S(c,b)
end
```

The size of the list of channels `a` is determined during initialization of the workstation. This size is used for the generation of the process buffers, and the accompanying channels.

The assembly server process works in the same way as before, except for a generic `n` components, it is impossible to write a select statement explicitly. Instead, an *unwind* is used to unfold the alternatives:

```
proc S(list chan? item c, chan! list item b):
   list(size(c)) item v;
   list int rec;
   while true:
      rec = range(size(c));
      while not empty(rec):
         select
            unwind i in rec
               c[i]?v[i]: rec = rec - [i]
            end
         end
      end;
      delay ...;
      b!v
   end
end
```

The received components are again in `v`. Item `v[i]` is received from channel `c[i]`. The indices of the channels that have not provided an item are in the list `rec`. Initially, it contains all channels `0...size(c)`, that is, `range(size(c))`. While `rec` still has a channel index to monitor, the `unwind i in rec` unfolds all alternatives that are in the list. For exam-

ple, if `rec` contains `[0, 1, 5]`, the `select unwind i in rec ... end` is equivalent to

```
select
   c[0]?v[0]: rec = rec - [0]
alt
   c[1]?v[1]: rec = rec - [1]
alt
   c[5]?v[5]: rec = rec - [5]
end
```

After receiving an item, the index of the channel is removed from `rec` to prevent receiving a second item from the same channel. When all items have been received, the assembly process starts (modeled with a `delay`, followed by sending the assembled component away with `b!v`.

In practical situations these assembly processes are performed in a more cascading manner: two or three components are 'glued' together in one assemble process, followed in the next process by another assembly process.

10.5 Exercises

1. To understand how time and time units relate to each other, change the time unit of the model of Section 10.1.

   (a) Change the model to using time units of one second (that is, one time unit means one second of simulated time).

   (b) Change the model to using time units of twelve seconds (that is, one time unit means twelve seconds of simulated time).

2. Predict the resulting throughput and flow time for a deterministic case like in Section 10.2, with `ta = 4` and `ts = 5`. Verify the prediction with an experiment, and explain the result.

3. Extend the model of Exercise 1 in Section 9.5 with a single deterministic server taking `4.0` time units to model the production capacity

of the factory. Increase the number of products inserted by the generator, and measure the average flow time for

(a) A FIFO buffer with control policy `low = 0` and `high = 1`.

(b) A FIFO buffer with control policy `low = 1` and `high = 4`.

(c) A *LIFO* buffer with control policy `low = 1` and `high = 4`.

Chapter 11

Conveyors

A conveyor is a long belt on which items are placed at the starting point of the conveyor. The items leave the conveyor at the end point, after traveling a certain period of time on the conveyor. The number of items traveling on the conveyor varies, while each item stays the same amount of time on the conveyor. It works like a buffer that provides output based on item arrival time instead of based on demand from the next process.

11.1 Timers

To model a conveyor, you have to wait until a particular point in time. The Chi 3 language has timers to signal such a time-out. The timer is started by assigning it a value. From that moment, it automatically decrements when time progresses in the model, until it reaches zero. The function `ready` gives the boolean value `true` if the timer is ready. The amount of time left can be obtained by reading from the variable. An example:

```
proc P():
   timer t;
   delay 10.0;
   t = timer(5.0); # Get a time-out at time = 15.0
   for i in range(7):
```

```
      write("%f %f %b\n, time, real(t), ready(t));
      delay 1.0
   end
end

model M():
   run
      P()
end
```

Initially, `time` equals `0.0`. The first action of process `P` is to delay the time for `10.0` time units. Now the value of `time` equals `10.0`. Nothing happens to timer `t` as it was already zero. At time `10` timer `t` is started with the value `5.0`. The output of the program is:

```
10.0   5.0    false
11.0   4.0    false
12.0   3.0    false
13.0   2.0    false
14.0   1.0    false
15.0   0.0    true
16.0   0.0    true
```

Timer `t` decrements as time progresses, and it is `ready` at `10.0 + 5.0` units. A process can have more timers active at the same moment.

11.2 A conveyor

A conveyor is schematically depicted in Figure 11.1. Three items are placed on the conveyor. For simplicity, assume the conveyor is 60.0 meter long and has a speed of 1 meter per second. An item thus stays on the conveyor for 60.0 seconds.

Item `0` has been placed on the conveyor 50.0 seconds ago, and will leave the conveyor 10.0 second from now. In the same way, item `1` will leave 30.0 seconds from now, and `2` leaves after 45.0 seconds. Each item has a *yellow*

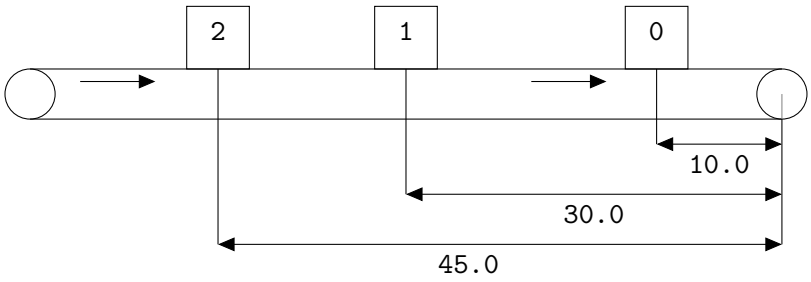


Figure 11.1: A conveyor with three items.

sticker with the time that the item leaves the conveyor. Based on this idea, tuple type `conv_item` has been defined, consisting of a field `item`, denoting the received item, and a timer field `t`, with the remaining time until the item leaves the conveyor:

```
type conv_item = tuple(item x; timer t);

proc T(chan? item a; chan! item b; real convey_time):
   list conv_item xst; item x;
   while true:
      select
         a?x:
            xst = xst + [(x, timer(convey_time))]
      alt
         not empty(xst) and ready(xst[0].t), b!xst[0].item:
            xst = xst[1:]
      end
   end
end
```

The conveyor always accepts new items from channel `a`, and adds the item with the yellow sticker to the list. If the conveyor is not empty, and the timer has expired for the first item in the list, it is sent (without sticker)

to the next process. The conveyor sends items to a process that is always willing to a receive an item, this implies that the conveyor is never blocked. Blocking implies that the items nevertheless are transported to the end of the conveyor.

11.3 A priority conveyor

In this example, items are placed on a conveyor, where the time of an item on the conveyor varies between items. Items arriving at the conveyor process, get inserted in the list with waiting items, in ascending order of their remaining time on the conveyor. The field `tt` in the item denotes the traveling time of the item on the conveyor:

```
type item      = tuple(...; real tt; ...),
     conv_item = tuple(item x; timer t);
```

The predicate function `pred` is defined by:

```
func bool pred(conv_item x, y):
   return real(x.t) < real(y.t)
end

proc T(chan? item a; chan! item b):
   list conv_item xst; item x;
   while true:
      select
         a?x:
            xst = insert(xst, (x, timer(x.tt)), pred)
      alt
         not empty(xst) and ready(xst[0].t), b!xst[0].item:
            xst = xst[1:]
      end
   end
end
```

The conveyor process works like before, except the new item is inserted in the list according to its remaining time, instead of at the rear of the list.

11.4 Exercises

1. Model the system as shown in Figure 11.2 where `T` is a conveyor process with a capacity of *at most* three products and exponentially distributed conveying times with an average of `4.0`.

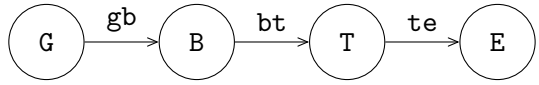


Figure 11.2: A conveyor system.

   (a) Compute the average flow time of products in the system.

2. Model the system as shown in Figure 11.3 with exponentially distributed server processing times with an average of `4.0`.

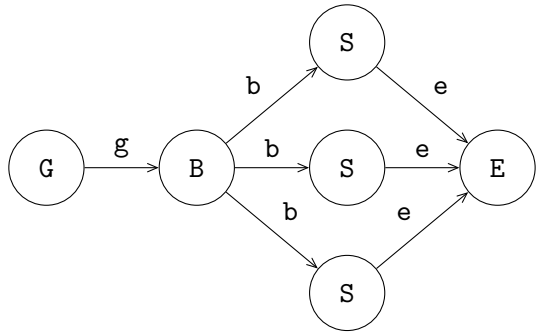


Figure 11.3: A system with three parallel servers.

(a) Compute the average flow time of products in the system.

(b) Are there differences in behavior between both systems? Why (not)?

Index