Chapter 4

Sequential processes

In chapter 2 we described behaviour by labelled transition systems. If behaviour becomes more complex this technique falls short and we need a higher level syntax to concisely describe larger labelled transition systems. In this chapter we describe processes using an extended process algebra. The building blocks of our process algebra are (multi-)actions with data. We provide operators to combine behaviour in both a sequential and nondeterministic way and allow it to be controlled by data parameters. Axioms are used to characterise the meaning of the various constructs.

The language that we get allows to describe all reactive behaviour. In the next chapter we define operators to combine behaviour in a more complex way, especially using the parallel operator. But these operators do not add to the expressivity of the language.

The sort of all processes defined in this and the next chapter is $\mathbb{P}$.

4.1 Actions

As in chapter 2, actions are the basic ingredients of processes. More precisely, every action is an elementary process. Actions can carry data. E.g., a $receive$ action can carry a message, and an $error$ action can carry a natural number, for instance indicating its severity. Actions can have any number of parameters. They are declared as follows:

act $timeout$;
$error : \mathbb{N}$;
$receive : \mathbb{B} \times \mathbb{N}^+$;

This declares parameterless action name *timeout*, action name *error* with a data parameter of sort $\mathbb{N}$ (natural numbers), and action name *receive* with two parameters of sort $\mathbb{B}$ (booleans) and $\mathbb{N}^+$ (positive numbers) respectively. For the above action name declaration, *timeout*, *error*(0) and *receive*$(false, 6)$ are valid actions. Processes cannot appear as parameters of actions.

Actions are events that happen atomically in time. They have no duration. In case duration of activity is important, it is most convenient to think of an action as

MA1	$\alpha\|\beta = \beta\|\alpha$
MA2	$(\alpha\|\beta)\|\gamma = \alpha\|(\beta\|\gamma)$
MA3	$\alpha\|\tau = \alpha$
MD1	$\tau \setminus \alpha = \tau$
MD2	$\alpha \setminus \tau = \alpha$
MD3	$\alpha \setminus (\beta\|\gamma) = (\alpha \setminus \beta) \setminus \gamma$
MD4	$(a(d)\|\alpha) \setminus a(d) = \alpha$
MD5	$(a(d)\|\alpha) \setminus b(e) = a(d)\|(\alpha \setminus b(e))$ if $a \not\equiv b$ or $d \not\approx e$
MS1	$\tau \sqsubseteq \alpha = \mathit{true}$
MS2	$a(d) \sqsubseteq \tau = \mathit{false}$
MS3	$a(d)\|\alpha \sqsubseteq a(d)\|\beta = \alpha \sqsubseteq \beta$
MS4	$a(d)\|\alpha \sqsubseteq b(e)\|\beta = a(d)\|(\alpha \setminus b(e)) \sqsubseteq \beta$ if $a \not\equiv b$ or $d \not\approx e$
MAN1	$\underline{\tau} = \tau$
MAN2	$\underline{a(d)} = a$
MAN3	$\underline{\alpha\|\beta} = \underline{\alpha}\|\underline{\beta}$

Table 4.1: Axioms for multi-actions

the beginning of the activity. If that does not suffice, activity can be modelled by two actions that mark its beginning and end. A declaration of actions describing the beginning and end of an activity a could look like:

act a_{begin}, a_{end};

From now on, we write $a, b, \ldots$ to denote both actions and action names. In concrete models we attach the required number of data arguments to an action name in accordance with the declaration, e.g., $a(1, \mathit{true})$. In more abstract treatments we let actions have only a single parameter, and we typically write $a(d)$ or $a(e)$. If we want to stress that there can be zero or more parameters, we sometimes write $a(\vec{d}), a(\vec{e}), \ldots$.

4.2 Multi-actions

Multi-actions represent a collection of actions that occur at the same time instant. Multi-actions are constructed according to the following BNF grammar. BNF stands for Backus-Naur[1] Form which is a popular notation to denote context-free grammars.

$$\alpha ::= \tau \mid a(\vec{d}) \mid \alpha|\beta,$$

[1]John Backus (1924-2007) developed the first higher level programming language Fortran. Peter Naur (1928-. . .) worked on the development of the very influential programming language Algol 60 for which he received the Turing award.

where, as indicated above, $a(\vec{d})$ is used to stress that the action name can have zero or more data parameters, but in general we leave the small arrow denoting the vector symbol out. See for instance the axioms in table 4.1.

The term τ represents the empty multi-action, which contains no actions and as such cannot be observed. It is exactly the internal action introduced in chapter 2. The multi-action $\alpha|\beta$ consists of the actions from both the multi-actions α and β, which all must happen simultaneously.

Typical examples of multi-actions are the following: τ, $error|error|send(true)$, $send(true)|receive(false, 6)$ and $\tau|error$. We generally write $\alpha, \beta, \ldots$ as variables for multi-actions. Multi-actions are particularly interesting for parallel behaviour. If sequential behaviour is described, multi-actions generally do not occur.

In table 4.1 the basic properties about multi-actions are listed by defining which multi-actions are equal to each other using the equality symbol ($=$). In particular the first three are interesting, as they express that multi-actions are associative, commutative and have τ as unit element. This structure is called a monoid.

Because multi-actions are given by the BNF above, we know the shape of all multi-actions and can use induction on the structure of multi-actions to prove properties of all multi-actions. This form of induction is in general called *structural induction*, illustrated in example 4.2.1 below. This is not always the most convenient form of induction to prove properties on multi-actions. An alternative is induction on the number of actions in a multi-action, which is illustrated in example 4.2.2.

Example 4.2.1. The following lemma $(\alpha|\beta) \setminus \beta = \alpha$ holds for all multi-actions α and β. We prove for any multi-action α that $(\alpha|\beta) \setminus \beta = \alpha$ with induction on the structure of β. So, there are two base cases:

$$(\alpha|\tau) \setminus \tau = \alpha,$$
$$(\alpha|a(d)) \setminus a(d) = \alpha.$$

These follow directly by MA3 and MD3, and MA1 and MD4 respectively. For the induction case we can assume that the lemma holds for smaller multi-actions and prove it for larger ones. Concretely, if we know that the induction hypotheses $(\alpha'|\beta_1)\setminus\beta_1 = \alpha'$ and $(\alpha'|\beta_2)\setminus\beta_2 = \alpha'$ for any multi-action α' hold, $(\alpha|\beta_1|\beta_2)\setminus(\beta_1|\beta_2) = \alpha$ must be proven. This can be shown as follows:

$$(\alpha|\beta_1|\beta_2) \setminus (\beta_1|\beta_2) \overset{\text{MA1, MA2, MD3}}{=} ((\alpha|\beta_2|\beta_1) \setminus \beta_1) \setminus \beta_2 \overset{i.h.}{=} (\alpha|\beta_2) \setminus \beta_2 \overset{i.h.}{=} \alpha.$$

As we have proven all induction steps, we can conclude that in general $(\alpha|\beta)\setminus\beta = \alpha$.

Example 4.2.2. It is obvious that $\alpha\setminus\alpha = \tau$. This can be most conveniently be proven by induction on the number of actions in α. If α has no actions, it must be equal to τ: $\tau\setminus\tau \overset{\text{MD2}}{=} \tau$ (cf., exercise 4.2.4 (2)). If α has at least one action α can be written as $a(d)|\beta$ where β has less actions. So, the induction hypothesis allows us to use $\beta\setminus\beta = \tau$. Now the proof of the inductive case becomes

$$\alpha \setminus \alpha = (a(d)|\beta) \setminus (a(d)|\beta) \overset{\text{MD3}}{=} ((a(d)|\beta) \setminus a(d)) \setminus \beta \overset{\text{MD4}}{=} \beta \setminus \beta \overset{\text{i.h.}}{=} \tau.$$

As $\alpha\setminus\alpha = \tau$ has been proven for α's containing any number of actions, we can conclude it holds for all α.

There are a few operators on multi-actions that turn out to be useful. There is an operator $\underline{\alpha}$ that associates with a multi-action α the multi-set of action names that is obtained by omitting all data parameters that occur in α. We also define operators $\setminus$ and $\sqsubseteq$ on multi-actions that represents removal and inclusion of multi-actions. Here $\equiv$ denotes syntactic equality on action names and $\approx$ denotes equality on data.

Exercise 4.2.3. Simplify the following expressions using the equations in table 4.1.

1. $(a(1)|b(2))\setminus(b(2)|a(1))$.
2. $(a(1)|b(2))\setminus(b(3)|c(2))$.
3. $a(1)|b(2) \sqsubseteq b(2)$.

Exercise 4.2.4. Prove 1. and 2. below using induction on the structure of multi-actions and 3. using induction on the number of actions in α for all multi-actions α, β and γ:

1. $(\alpha|\beta|\gamma)\setminus\beta = \alpha|\gamma$.
2. Every multi-action α can be written as either τ or $a(d)|\beta$.
3. $\alpha \sqsubseteq \alpha = \mathit{true}$.

4.3 Sequential and alternative composition

There are two main operators to combine multi-actions into behaviour. These are the sequential and alternative composition operators. For processes p and q we write $p{\cdot}q$ to indicate the process that first performs the behaviour of p and after p terminates, continues to behave as q. Note that the dot is often omitted when denoting concrete processes.

If a, b and c are actions, the action a is the process that can do an a-action and then terminate. The process $a{\cdot}b$ can do an a followed by a b and then terminate. The process $a{\cdot}b{\cdot}c$ can do three actions in a row before terminating. The three processes are depicted in figure 4.1.

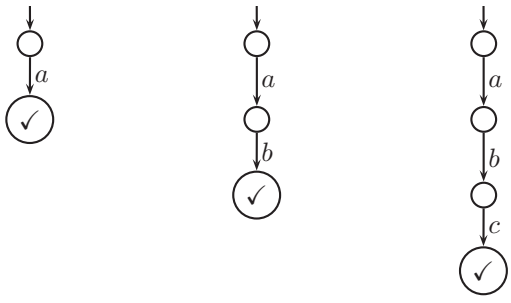


Figure 4.1: Three sequential processes

A1	$x + y = y + x$
A2	$x + (y + z) = (x + y) + z$
A3	$x + x = x$
A4	$(x + y)\cdot z = x\cdot z + y\cdot z$
A5	$(x\cdot y)\cdot z = x\cdot(y\cdot z)$
A6‡	$x + \delta = x$
A7	$\delta\cdot x = \delta$
Cond1	$\mathit{true}\rightarrow x \diamond y = x$
Cond2	$\mathit{false}\rightarrow x \diamond y = y$
THEN‡	$c\rightarrow x = c\rightarrow x \diamond \delta$
SUM1	$\sum_{d:D} x = x$
SUM3	$\sum_{d:D} X(d) = X(e) + \sum_{d:D} X(d)$
SUM4	$\sum_{d:D}(X(d) + Y(d)) = \sum_{d:D} X(d) + \sum_{d:D} Y(d)$
SUM5	$(\sum_{d:D} X(d))\cdot y = \sum_{d:D} X(d)\cdot y$

Table 4.2: Axioms for the basic operators

The process $p+q$ is the alternative composition of processes p and q. This expresses that either the behaviour of p or that of q can be chosen. The actual choice is made by the first action in either p or q. So, the process $a + b$ is the process that can either do an a or a b and the process $a\cdot b + c\cdot d$ can either do a followed by b, or c followed by d as shown in figure 4.2. The alternative composition operator $+$ is also called the choice operator. In table 4.2 some axioms are given, that indicate which processes are equal

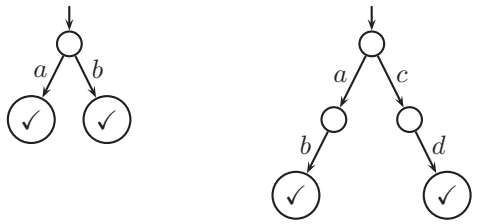


Figure 4.2: Two processes with a choice

to other processes. In the axioms symbols x and y are variables that can be substituted by processes. For the alternative and sequential composition, the axioms A1 to A5 are particularly important. A1 and A2 say that alternative composition is commutative and associative. Practically, this means that is does not matter whether behaviour stands at the left or right of the choice, and that brackets to group more than one choice operator

can be omitted. An interesting axiom is A3, which says that the choice is idempotent. If a choice can be made between two identical processes, there is no choice to make at all.

The axiom A4 says that sequential composition right distributes over the choice. The left distribution, namely $x{\cdot}(y+z) = x{\cdot}y + x{\cdot}z$ is not valid, as it would imply that $a{\cdot}(b+c)$ would be equal to $a{\cdot}b+a{\cdot}c$ which are in general not behaviourally equivalent, as argued in the previous chapter (see figure 2.9). The axiom A5 says that sequential composition is associative. So, we can as well write $a{\cdot}b{\cdot}c$ instead of $(a{\cdot}b){\cdot}c$, as the position of the brackets is immaterial.

The axioms listed in table 4.2 and elsewhere are valid for strong bisimulation. If we provide axioms that hold for other process equivalences, this will explicitly be stated. Using the axioms we can show the process $a{\cdot}b + a{\cdot}b$ equal to $a{\cdot}(b + b)$. This goes as follows:

$$a{\cdot}(b + b) \stackrel{\text{A3}}{=} a{\cdot}b \stackrel{\text{A3}}{=} a{\cdot}b + a{\cdot}b.$$

In the first step we take the subexpression $b + b$. It is reduced to b using axiom A3 by substituting b for x. Henceforth b is used to replace $b + b$. In the second step, axiom A3 is used again, but now by taking $a{\cdot}b$ for x. Compare this to the proof in figure 2.8.

As a more elaborate example, we show that $((a + b){\cdot}c + a{\cdot}c){\cdot}d$ and $(b + a){\cdot}(c{\cdot}d)$ are equal.

$$\begin{aligned}((a + b){\cdot}c + a{\cdot}c){\cdot}d &\stackrel{\text{A4}}{=} (a{\cdot}c + b{\cdot}c + a{\cdot}c){\cdot}d \stackrel{\text{A1,A3}}{=}\\ &(a{\cdot}c + b{\cdot}c){\cdot}d \stackrel{\text{A4}}{=} ((a + b){\cdot}c){\cdot}d \stackrel{\text{A5}}{=} (b + a){\cdot}(c{\cdot}d).\end{aligned}$$

Exercise 4.3.1. Derive the following equations from the axioms A1-A5:

1. $((a + a){\cdot}(b + b)){\cdot}(c + c) = a{\cdot}(b{\cdot}c)$,
2. $(a + a){\cdot}(b{\cdot}c) + (a{\cdot}b){\cdot}(c + c) = (a{\cdot}(b + b)){\cdot}(c + c)$.

We use the shorthand $x \subseteq y$ for $x + y = y$, and write $x \supseteq y$ for $y \subseteq x$. This notation is called *summand inclusion*. It is possible to divide the proof of an equation into proving two inclusions, as the following exercise shows.

Exercise 4.3.2. Prove that if $x \subseteq y$ and $y \subseteq x$, then $x = y$.

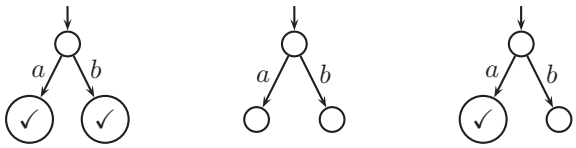
4.4 Deadlock

A remarkable but very essential process is *deadlock*, also called *inaction*. It is denoted as δ and cannot do any action. In particular, it cannot terminate. The properties of deadlock are best illustrated by the axioms A6‡ and A7 in table 4.2. Axiom A7 says that it is impossible to go beyond a deadlock. So, the x in $\delta{\cdot}x$ can be omitted because it cannot be reached.

The axiom A6‡ says that if we can choose between a process x and δ, we must choose x because δ has no first action that would cause the δ to be chosen. The axiom A6‡ is designated with ‡ to indicate that it is only sound in an untimed setting, i.e., in case x represents a process in which no explicit reference to time is made. If one

restricts this axiom to multi-actions, i.e., $\alpha + \delta = \alpha$ with α a multi-action, one obtains a weaker axiom, called A6. A6 holds also in the timed settings as presented in chapter 8.

The deadlock can be used to prevent processes from terminating. So, the process $a + b$ can terminate, whereas the process $a{\cdot}\delta + b{\cdot}\delta$ cannot, and $a + b{\cdot}\delta$ has both a terminating branch and one that cannot terminate. The tree graphs belonging to these processes are depicted below.



Deadlock is not very often used as a specification primitive, as specifying that a system has a deadlock is strange because this is undesired behaviour. The deadlock is generally the result of some communicating parallel processes. It shows that there is some incompatibility in these processes such that at a certain moment no actions can be done anymore.

The most common use of deadlock in a specification is to prevent a process from terminating. For instance an action $error$ can be added to a specification to indicate a state that should never be reached. By generating the state space and by inspecting that the error action does not occur in it, it can be shown that the undesired state can not be reached indeed. But often, the behaviour after the $error$ action is of no concern. This behaviour is made unreachable by putting a deadlock after the error: $error{\cdot}\delta$.

Later when we introduce time, it will turn out that a stronger deadlock than δ exists. One feature of δ is that it lets time pass. The stronger variant can even let time come to a halt.

Exercise 4.4.1. Prove the following implication, assuming that x represents an untimed process.

$$x + y = \delta \text{ implies } x = \delta.$$

This implication is known as *Fer-Jan's*[2] *lemma*. It was the first process-algebraic identity verified using proof checkers.

4.5 The conditional operator

Data influences the run of processes using the conditional operator. For a condition c of sort $\mathbb{B}$ (boolean) and processes p and q we write $c \to p \diamond q$ to express *if* c *then* p *else* q. The condition c must consist of data, and it is not allowed to use processes in c. The axioms Cond1 and Cond2 in table 4.2 are obvious. If c is $true$, then the behaviour is p, and otherwise it is q.

The else part of the condition can be omitted, meaning that nothing can be done in the else part. This is expressed by the axiom THEN‡, also only valid when x does not refer to time. When x contains time, the else part must be a timed deadlock.

[2]Fer-Jan de Vries (1956–. . .) is a researcher in logic and semantics with a strong interest in infinitary systems.

To say that if the water level is too high, an alarm is sounded, and otherwise an ok message is sent, is described as follows:

$$(waterLevel{>}limit) \to soundAlarm \diamond sendOK.$$

Here, $waterLevel$ is a local variable representing the water level.

Case distinction can also neatly be described. Suppose there is a data variable *desiredColour* which indicates which colour a signal should get. Using actions such as *setSignalToRed* the desire is transformed in actions to set the signal to the required colour:

$$\begin{aligned}&(desiredColour{\approx}Red) \to setSignalToRed\\ +\ &(desiredColour{\approx}Yellow) \to setSignalToYellow\\ +\ &(desiredColour{\approx}Green) \to setSignalToGreen.\end{aligned}$$

The axioms Cond1 and Cond2 are very handy, when used in combination with *case distinction* or *induction* on booleans. There are exactly two booleans, $true$ and $false$, which means that to prove a property for all booleans, it suffices to prove it only for $true$ and $false$. More concretely, in order to show $c \to x \diamond x = x$ we must show $true \to x \diamond x = x$, which follows directly from Cond1, and $false \to x \diamond x = x$ which follows directly from Cond2.

Exercise 4.5.1. Describe a process that if a given natural number n is larger than 0, can do a $down$ action, if n is larger than 100 can perform a $too\_large\_warning$ and that can always do an up action.

Exercise 4.5.2. Derive the following equations:

- $c \to x \diamond y = \neg c \to y \diamond x$;
- $c \vee c' \to x \diamond y = c \to x \diamond (c' \to x \diamond y)$;
- $x{+}y \sqsupseteq c{\to}x{\diamond}y$;
- if assuming that c holds, we can prove that $x = y$ then $c \to x \diamond z = c \to y \diamond z$.

4.6 The sum operator

The sum operator $\sum_{d:D} p(d)$ is a generalisation of the choice operator. The notation $p(d)$ is used to stress that d can occur in the process p. Where $p + q$ allows a choice among processes p and q, $\sum_{d:D} p(d)$ allows to choose $p(d)$ for any value d from D. If D is finite, e.g., equal to $\mathbb{B}$, then the sum operator can be expressed using the choice. In this case the following is valid:

$$\sum_{c:\mathbb{B}} p(c) = p(true) + p(false).$$

For sums over infinite domains, e.g., $\sum_{n:\mathbb{N}} p(n)$, it is not possible anymore to expand the sum operator using the choice operator.

The sum operator can be used for many purposes, but the most important one is to model reading data values. Modelling a (one time usable) buffer that can read a message to be forwarded at a later moment, yields the following:

$\sum_{m:Message} read(m)\cdot forward(m)$.

A commonly made mistake is to not place the sum operator directly around the action in which the reading takes place. Compare the following two processes, where reading takes place with actions $read_1$ and $read_2$.

$\sum_{m_1:Message} read_1(m_1)\cdot \sum_{m_2:Message} read_2(m_2)\cdot forward(m_1, m_2)$.
$\sum_{m_1,m_2:Message} read_1(m_1)\cdot read_2(m_2)\cdot forward(m_1, m_2)$.

In the first (correct) process, the message m_2 is chosen when the action $read_2$ takes place. In the second process, the message m_2 to be read is chosen when action $read_1$ takes place. When doing $read_2$, the value to be read is already fixed. If this fixed value is not equal to the value to be read, a deadlock occurs.

The axioms for the sum operator given in table 4.2 are quite subtle. The axiom SUM2 did exist, but turned out to be a reformulation of general logic principles and has therefore been omitted. We defer full treatment of these axioms to chapter 9. In order to use them it is necessary that data variables that occur in the sum operator must not bind variables in terms that are substituted for process variables as x, y and z. For variables written as $X(d)$ it is allowed to substitute a term with a data variable d even if d becomes bound by a surrounding sum.

So, for the axiom SUM1 no process containing the variable d can be substituted for x. This is another way of saying that d does not occur in x (or more precisely, any process substituted for x). Therefore, the sum operator can be omitted, as no real choice needs to be made. This is essentially the same as what axiom A3 expresses.

Exercise 4.6.1. Specify a (one time usable) buffer that reads a natural number, and forwards it if the number is smaller than 100. Otherwise it should flag an overflow.

Exercise 4.6.2. The axiom SUM1 resembles the axiom A1. With which axioms do SUM3, SUM4 and SUM5 correspond?

4.7 Recursive processes

With the description of one time usable buffers in the previous section it already became apparent that continuing behaviour must also be described. This is done by introducing process variables and defining their behaviour by equations. Consider the following specification, which describes the alarm clock at the left in figure 2.2:

act $set, alarm, reset$;
proc $P = set\cdot Q$;
$Q = reset\cdot P + alarm\cdot Q$;

This declares process variables P and Q (often just called processes, which explains the use of the keyword **proc**). The process variable P corresponds to the situation where the alarm clock is switched off, and the process variable Q corresponds to the state where the alarm clock is set.

If in a set of equations defining a process there are only single variables at the left we speak of a *recursive specification*. The variables at the left are called the defined

process variables. If every occurrence of a defined process variable at the right is preceded by an action, we speak about a *guarded recursive specification*. A guarded recursive specification defines the behaviour of the process variables that occur in it. In the example given above, the behaviour of P and Q is neatly defined.

The keyword **init** can be used to indicated the initial behaviour. In accordance with figure 2.2 this ought to be variable P.

init P;

While interacting with their environment, processes store information that can later influence their behaviour. For this purpose process variables can contain parameters in which this information can be stored. Data and processes are strictly distinguished. This means that there cannot be any reference to processes in data parameters.

We can transform the alarm clock such that it sounds its alarm after a specified number of *tick* actions have happened.

act $set{:}\mathbb{N}$; $alarm, reset, tick$;
proc $P = \sum_{n:\mathbb{N}} set(n)\cdot Q(n) + tick\cdot P$;
$Q(n{:}\mathbb{N}) = reset\cdot P + (n{\approx}0) \rightarrow alarm\cdot Q(0) \diamond tick\cdot Q(n{-}1)$;
init P;

Note that the value of n is used in process Q to determine whether an alarm must sound or whether a *tick* action is still possible.

A guarded recursive specification with data also uniquely defines a process. More precisely, it defines a function from the data parameters to processes. E.g., the 'process' Q above is actually a function from natural numbers to processes. The equation must be understood to hold for any concrete value for the parameters. So, given the equation for Q above, the following are also valid by taking for n respectively 0, m, $n+1$ and $23k+7$ and simplifying the result.

$$\begin{aligned}
&Q(0) = reset\cdot P + alarm\cdot Q(0);\\
&Q(m) = reset\cdot P + (m{\approx}0) \rightarrow alarm\cdot Q(0) \diamond tick\cdot Q(m{-}1);\\
&Q(n{+}1) = reset\cdot P + tick\cdot Q(n);\\
&Q(23k+7) = reset\cdot P + tick\cdot Q(23k+6).
\end{aligned}$$

Below, we describe a process that maintains an unbounded array in which natural numbers can be stored. There are actions set, get and $show$. The action $set(n,m)$ sets the nth entry of the array to value m using the function update operator. After an action $get(n)$ an action $show(m)$ shows the value m stored at position n in the array.

act $set : \mathbb{N} \times \mathbb{N}$;
$get, show : \mathbb{N}$;
proc $P(a{:}\mathbb{N}{\rightarrow}\mathbb{N})$
$= \sum_{n,m:\mathbb{N}} set(n,m)\cdot P(a[n{\rightarrow}m])$
$+ \sum_{n:\mathbb{N}} get(n)\cdot show(a(n))\cdot P(a)$;

Another example is the specification of a sorting machine. This machine reads arrays of natural numbers, and delivers sorted arrays with exactly the same numbers. The predicate *sorted* expresses that the numbers in an array are increasing and the predicate

equalcontents expresses that each of the arrays a and a' contain the same elements. But note that *equalcontents* does not necessarily preserve the number of occurrences of numbers.

act $read, deliver : \mathbb{N} \to \mathbb{N}$;
map $sorted, equalcontents, includes : (\mathbb{N} \to \mathbb{N}) \to \mathbb{B}$;
var $a, a' : \mathbb{N} \to \mathbb{N}$;
eqn $sorted(a) = \forall i{:}\mathbb{N}.a(i) \leq a(i{+}1)$;
$equalcontents(a, a') = includes(a, a') \wedge includes(a', a)$;
$includes(a, a') = \forall i{:}\mathbb{N}.\exists j{:}\mathbb{N}.a(i){\approx}a'(j)$;
proc $P = \sum_{a:\mathbb{N}\to\mathbb{N}} read(a)\cdot$
$\sum_{a':\mathbb{N}\to\mathbb{N}}(sorted(a')\wedge equalcontents(a, a')) \to deliver(a')\cdot P$;

Sometimes processes have a large number of parameters. In that case it is convenient to only write changing parameters in the right-hand side of processes by an explicit assignment. The following example shows how that can be done.

proc $P(n_1, n_2, n_3{:}\mathbb{N})$
$= \sum_{m:\mathbb{N}} change\_par_1(m)\cdot P(n_1{=}m)$
$+ \sum_{m:\mathbb{N}} change\_par_2(m)\cdot P(n_2{=}m)$
$+ \sum_{m:\mathbb{N}} change\_par_3(m)\cdot P(n_3{=}m)$
$+ change\_no\_par\cdot P()$;

The parameters that are not mentioned at the right side remain unchanged. Note in particular $P()$ at the end, which describes that no value of any parameters is changed.

We only allow the process assignment notation in recursive invocations at the right-hand side if the process variable of this invocation is equal to the one at the right-hand side. So, $P(n{:}\mathbb{N}) = a.Q()$ is not allowed. The reason is that the parameters of Q and P can differ making it unclear how the assignments must be interpreted.

There is a ugly snag in the use of use of assignments. Consider the process

$$P(x{:}\mathbb{N}) = \sum_{x:\mathbb{N}} a\cdot P(x{=}x)$$

To which x's do the x's in $x{=}x$ refer? The answer is that the first x in $x{=}x$ is a placeholder for a variable in the left-hand side of the equation. The second x in $x{=}x$ is an ordinary variable which is bound by its closest binder, i.e., the x in the sum. So, the equation above must be read as

$$P(x{:}\mathbb{N}) = \sum_{y:\mathbb{N}} a\cdot P(x{=}y)$$

Note that this also indicates that an assignment $x{=}x$ cannot always be removed from a right-hand side. This is only possible if the x at the right-hand side refers to the parameter x of the equation.

This finishes the treatment of sequential processes. We have now seen all the operators to specify sequential behaviour. Using recursion we can specify iterative behaviour and by using data parameters in these equations they are suitable to describe even the most complex real life systems.

Exercise 4.7.1. Describe the behaviour of a buffer with capacity 1 that iteratively reads and forwards a message. Add the option to empty the buffer when it is full, by a specific *empty* action.

Exercise 4.7.2. Describe a simple coffee machine that accepts 5 and 10 cent coins. After receiving at least 10 cents, it asks whether cream and sugar needs to be added and then serves the desired coffee, after which it repeats itself.

Exercise 4.7.3. Adapt the predicate $equalcontents(a, a', n)$ such that it also preserves the number of occurrences of data elements, assuming that the arrays contain n elements.

★**Exercise 4.7.4.** Describe a coffee and tea machine that accepts coins (1 cent, 5 cent, 10 cent, 20 cent, 50 cent, 1 euro and 2 euro). Coffee costs 45 cent, tea costs 25 cent. The machine can pay back, but there is only a limited amount of coins (it knows exactly how many). It is necessary to develop a way how to accept coins, return change and deliver beverages when the machine is low on cash.

4.8 Axioms for the internal action

The axioms provided up till now in this chapter are valid for strong bisimulation. If we want to remove internal actions while transforming and simplifying processes, we need axioms to get rid of the internal action. These are provided in this section.

In the context of rooted branching bisimulation, the axioms in table 4.3 can be used, provided no time is used in x, y and z, which as elsewhere is indicated by the use of ‡. The axiom W‡ is pretty obvious. It says that a trailing τ after a process can be omitted. The axiom BRANCH‡ is more intriguing and generally seen as the typical axiom for branching bisimulation. It says that behaviour does not have to be apparent at once, but may gradually become visible. Concretely, instead of seeing $y + z$ at once, it is possible to see first the behaviour of y, and after some internal rumble (read τ), the total behaviour of $y + z$ becomes visible.

W‡	$x \cdot \tau = x$
BRANCH‡	$x \cdot (\tau \cdot (y + z) + y) = x \cdot (y + z)$

Table 4.3: Axioms for τ valid in rooted branching bisimulation for untimed processes

The characterising axioms for rooted weak bisimulation are found in table 4.4. These are also only valid in an untimed setting.

All equivalences in the Van Glabbeek spectrum have their equational characterisation. In table 4.5 axiomatic characterisations are provided for failures equivalence, trace equivalence, language equivalence and weak trace equivalence. Note that the axioms for trace and weak trace equivalence also hold for timed processes.

W‡	$x{\cdot}\tau = x$
W2‡	$\tau{\cdot}x = \tau{\cdot}x + x$
W3‡	$x{\cdot}(\tau{\cdot}y + z) = x{\cdot}(\tau{\cdot}y + z) + x{\cdot}y$

Table 4.4: Axioms for τ valid in rooted weak bisimulation for untimed processes

Failures eq.	F1‡	$a{\cdot}(b{\cdot}x{+}u) + a{\cdot}(b{\cdot}y{+}v) =$ $a{\cdot}(b{\cdot}x{+}b{\cdot}y{+}u) + a{\cdot}(b{\cdot}x{+}b{\cdot}y{+}v)$
	F2‡	$a{\cdot}x + a{\cdot}(y{+}z) = a{\cdot}x + a{\cdot}(x{+}y) + a{\cdot}(y{+}z)$
Trace eq.	RDIS	$x{\cdot}(y + z) = x{\cdot}y + x{\cdot}z$
Language eq.	Lang1‡	$x{\cdot}\delta = \delta$
	RDIS	$x{\cdot}(y + z) = x{\cdot}y + x{\cdot}z$
Weak trace eq.	RDIS	$x{\cdot}(y + z) = x{\cdot}y + x{\cdot}z$
	WT‡	$\tau{\cdot}x = x$
	W‡	$x{\cdot}\tau = x$

Table 4.5: Axioms for some other equivalences for untimed processes

Exercise 4.8.1. Derive the following equations using the axioms valid in rooted branching bisimulation.

- $a{\cdot}(\tau{\cdot}b + b) = a{\cdot}b$;
- $a{\cdot}(\tau{\cdot}(b + c) + b) = a{\cdot}(\tau{\cdot}(b + c) + c)$;
- If $y \subseteq x$, then $\tau{\cdot}(\tau{\cdot}x + y) = \tau{\cdot}x$.

See exercise 4.3.2 for the definition of $\subseteq$.

Exercise 4.8.2. Derive the axioms for rooted branching bisimulation from those for rooted weak bisimulation. Similarly, derive the axioms for rooted weak and branching bisimulation from those of weak trace equivalence.

4.9 Historical notes

Our notation for sequential processes stems from the Algebra of Communicating Processes [25] (for an overview see [18, 58]). We essentially presented here basic process algebra, (BPA), also called context-free process algebra, consisting of actions and the alternative- and sequential composition operator. Basic process algebra is interesting because it strongly resembles context-free grammars. Remarkably, all bisimulations

are (efficiently) decidable for such processes whereas all other equivalences are undecidable [77] (cf., section 10.4). Note that the use of a general sequential composition is not common. Most process algebras use an action prefix operator $a{:}p$ which is theoretically much simpler.

To this basic language the constant δ was added (denoted as NIL, $STOP$ or $\mathbf{0}$ elsewhere [12, 101, 133, 135]). As the axioms A6 and A7 suggest, the constant δ in process algebra has similar properties as the number 0 in number theory, and hence, investigations started to add a constant behaving like 1. This constant is denoted as ϵ or $\mathbf{1}$ and satisfies equations $\epsilon{\cdot}x{=}x$ and $x{\cdot}\epsilon{=}x$ [174]. It has even been tried to merge the properties of δ and ϵ but that has never been very successful [136]. We have not added ϵ because it combines badly with time. With a constant ϵ it would be easy to write $c{\rightarrow}p \diamond \epsilon$, saying that if c holds p must be executed, and otherwise the process must terminate to continue with subsequent behaviour. This is not that easily possible without this constant.

Combining data and processes was generally done in an ad hoc way in the early days of process algebras. Exceptions were process-algebraic specification languages, such as LOTOS [106, 68] and PSF [127]. The use of some form of conditional or if-then-else operator has been very common. The notation that we use looks quite like the one used in the guarded command language [110]. In μCRL, and several languages related to CSP, the conditional was denoted as $p \lhd c \rhd q$ [84].

The use of a sum operator over possibly infinite domains is relatively rare. More often the sum is restricted to finite domains only [68, 127]. The generality of our operator allows it to be used for several purposes, in particular for reading inputs from unbounded domains. A popular alternative for this is the use of input and output actions, often written as $a!3$ and $a?x{:}\mathbb{N}$ meaning that the value 3 is sent via channel a, and a value received via channel a is put into variable x [91]. In [119] it was proven that the expressivity of input/output actions is strictly less than that of the sum operator, which coincides with our experience that the unbounded sum operator is far more versatile. In combination with only the conditional operator it is straightforward to describe input and output with constraints, select minimal or maximal values from the input or describe detailed timed behaviour. The axioms that we provide here were first proposed in [83] and strongly influenced by the encoding of μCRL in Coq [29].

There are completely different approaches to incorporate data in processes. As one particular curiosity we mention process algebras with signals and state operators [14].

In mCRL2 there is a strict distinction between data and processes. Processes cannot be used in data, especially not in conditions and processes cannot be sent around through channels. This restriction can be relaxed leading to a whole new family of higher order process calculi such as CHOCS [168] and the higher order π-calculus [160]. In such process calculi, not only processes can be sent around but also, they can be run by receiving processes.

The axioms for weak bisimulation were provided in [133], those for branching bisimulation stem from [72]. An abundance of axioms for weaker equivalences can be found in [71, 70]. The use of guarded recursive equations with unique solutions is typical for ACP [25, 18].