Process Calculi: LOTOS

2.1 Introduction

A number of notations have been developed within concurrency theory for specification of concurrent systems, e.g. process calculi [96, 148],[1] temporal logics [136], Petri Nets [169] and extended finite state machines [105]. We will particularly focus on process calculi, as realised in the specification language LOTOS [101].

LOTOS (Language Of Temporal Ordering Specification) was defined during the 1980's by a standardisation committee chaired by Ed Brinksma of the University of Twente. The most significant influence on the design of the language was a number of previously defined process calculi, including CCS, CSP, CIRCAL [144] and ACP [15]. From amongst this list, the two most direct influences were CCS [148] and CSP [96]. In fact, the language is largely a composite of these two previous process calculi.

The language has two main parts: a behavioural part (sometimes also referred to as the process algebraic part) and a data part. The role of the behavioural part is to specify the order in which actions can occur; for example, you may specify that component B of a system receives messages from component A and then either passes the messages on to a further component, C, or loses the message. The data part on the other hand defines the data types that can be used in the behavioural part. For example, a queue data type might be defined. This queue type may then be used as an input queue by component B. Thus, when an action occurs at component B to indicate a message has arrived, the message will be added to the queue. The data part

---

[1]We prefer the term process calculus to process algebra, because in fact, the approach we present is not that advanced in algebraic terms. In particular, we do not consider algebraic proof systems. Although, the reader should be aware that the term, process algebra, is often used in the literature to describe very similar approaches to the one we highlight.

of LOTOS uses an abstract data typing language called ACT-ONE; see [24] for an introduction to this notation.

A process of restandardisation has been undertaken. One particular area of redefinition is the data part, which in its original form was seen to be very cumbersome and a hindrance to the uptake of the language. The ACT-ONE notation has been replaced with a functional notation. We discuss these revisions in Chapter 6.

It is quite easy though to view the behavioural and data parts as distinct. In fact, here we are almost exclusively interested in the behavioural part. We use the term *full LOTOS* (which we shorten to *fLOTOS*) to refer to the full language with data types and the term *basic LOTOS* (which we shorten to *bLOTOS*) to refer to the language without data types (i.e. just the behavioural part). We also subdivide basic LOTOS, because the full behavioural language contains a lot of syntax that is somewhat cumbersome to carry around when looking at the theoretical properties of the language. Thus, our main point of focus is a subset of bLOTOS that we call *primitive basic LOTOS* (which we shorten to *pbLOTOS*).

The next section (Section 2.2) introduces two specification examples that we use to illustrate formal description in LOTOS. Then Section 2.3 introduces pbLOTOS; and, finally, Section 2.4 presents example specifications written in pbLOTOS.

2.2 Example Specifications

A simple communication protocol and the Dining Philosophers problem are used as running examples. Both of these are standard examples of concurrent behaviour and readers who are familiar with them can safely skip this section.

2.2.1 A Communication Protocol

The communication protocol comprises three main components: a *sender process*, a *receiver process* and a *medium* (or channel). These components are depicted in Figure 2.1. The specification task here is to firstly model the behaviour of the medium (e.g. its ability to lose messages) and then to give sender and receiver process specifications that support reliable communication. The specification will use timeouts, sequence numbering and acknowledgement in order to do this.

The sender process obtains messages to send (also called packets or frames) from outside the protocol system (in terms of a layered protocol model, messages to send would be obtained from a previous layer in the protocol stack). The computation steps of the sender are: request a message from outside the system, successfully send the message (perhaps with some retransmission)
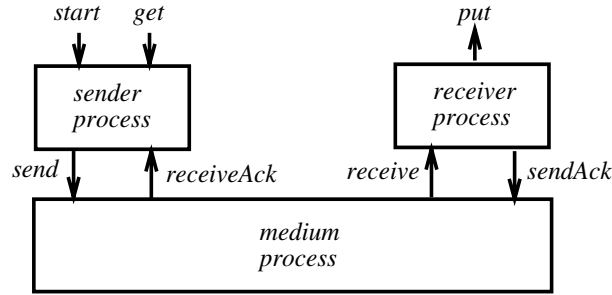
Fig. 2.1. Components in the Communication Protocol

and then request a new message. Thus, the protocol is a *stop and wait* protocol [187]; it waits for the current message to be successfully sent before it requests a new message to send.

Transmission using the protocol is initiated by a request to start from outside the protocol. The sender then obtains a message to send and sends it. Getting a message to send is identified by an event *get* being performed by the sender process with the environment and sending is identified by an event *send* occurring between the sending process and the medium. The medium then relays the message to the receiver. Successful transmissions cause an event *receive* to occur at the receiver process. However, the medium may lose the message, in which case no such event is able to occur. In addition, the receiver sends acknowledgements through the medium (so the medium is a duplex channel). Sending and receiving these acknowledgements are identified by the events *sendAck* and *receiveAck*, respectively. Successfully received messages are passed out of the system on the receiver side using the event *put*.

We consider two variants of this basic scenario. The first assumes a reliable acknowledgement medium. The second assumes that acknowledgements can be lost. We discuss these in turn.

- **Reliable Acknowledgement.** In this class of protocol, messages sent from the sender to the receiver may be lost, but acknowledgements will always be relayed successfully. This assumption simplifies the protocol considerably and avoids the necessity for sequence numbers. The sender process will still have to set a timer when it sends a message. If the timer expires, the message is assumed lost in transit and is resent.
- **Unreliable Acknowledgement.** The second variant assumes that acknowledgements may be lost. The troublesome scenario for such a protocol is that an acknowledgement is lost, the sender times out and retransmits the original message, which is successfully transmitted to the receiver. The receiver will have no way of knowing that this is a retransmission and will blindly pass it to higher layers, resulting in delivery of a duplicate message. Stop and wait protocols, which can lose acknowledgements, typically use

alternating bit sequence numbering in order to obtain reliable communication. A sequence number of zero or one is associated with every message. This means that retransmissions can be distinguished from transmission of new messages when an acknowledgement is lost, because the sequence number of a retransmission will have the same sequence number as the previously received message.

2.2.2 The Dining Philosophers

The Dining Philosophers scenario has been used for many years as an illustration of the problems associated with scheduling shared resources in concurrent systems. The version of the problem that we seek to specify is given by the following scenario.[2]

> Four philosophers (Aristotle, Buddha, Confucius and Descartes) are sitting around a table, which has a large plate of rice in the middle. Philosophers alternately think and eat. To be able to eat they must first pick up two chopsticks, one in their left hand and one in their right. They can pick up the chopsticks in any order, either right hand first or left hand first. Because there are only four chopsticks, not all of them can eat at the same time.

The table is depicted in Figure 2.2. A formal description of this problem will describe all the possible behaviours in which the four philosophers can engage.

2.3 Primitive Basic LOTOS

The Nature of LOTOS Specification. A major objective of formal description is not to over-specify and to allow implementation freedom by being non-prescriptive about aspects of realisation. Such avoidance of over-specification is at the heart of the process calculus approach. In particular, it is important that the correct interpretation is imposed on LOTOS descriptions. Specifically, they should be viewed as expressing the "externally visible possible behaviour" of a system. Specifications should be viewed as *black boxes*; they describe the order of possible external interaction, but do not prescribe how that interaction order is internally realised. Any physical system that realises the external behaviour is a satisfactory implementation.

The concept of the *environment* that a specification evolves in is central in obtaining this interpretation. The term environment refers to the behaviour that the *external observer* of a system wishes to perform. Note that this external observer could be either human or mechanical. Conceptually, a LOTOS

---

[2]This scenario is based upon a Dining Philosophers specification associated with the SEDOS tool set.
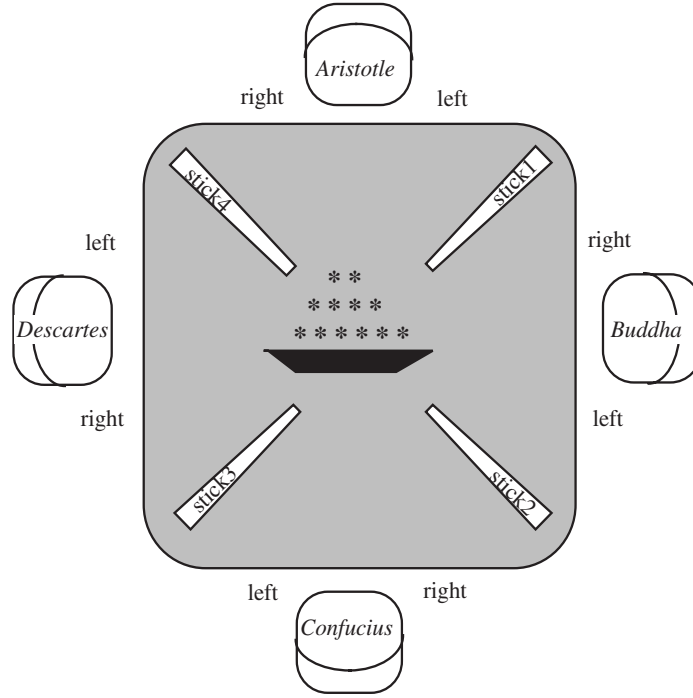
Fig. 2.2. The Dining Philosophers

specification only defines "possibilities" for evolution of a system and it is through interaction with a particular environment that these possibilities are resolved and realised. For example, if an environment cannot offer an action that a specification *must* perform, a deadlock will ensue.

As an illustration, we might view a LOTOS specification, called S, in the form depicted in Figure 2.3; i.e. as a black box with two interaction points between the specification and the environment, g and h. Such interaction points are called *gates* (the term *port* is also sometimes used). The set of all gates of a specification defines the *interface* to the specification. It is only through gates in this interface that an external observer can interact with the specified system.

Gates reference "locations" at which interactions can take place. At such gates, *actions* are performed. We say more shortly about this concept, but they can be thought of as interaction activities, e.g. passing a value, sending a message or pressing a button. In fact, the latter of these yields a nice pictorial representation of interaction between environment and specification. LOTOS descriptions define the order in which actions can be offered at gates; e.g. it might be that an action at gate g can only be offered once an action at gate h has been performed. Thus, typically, actions are only offered intermittently
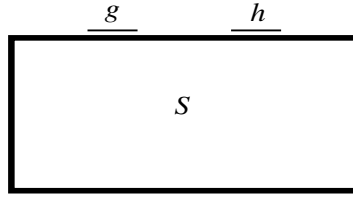
Fig. 2.3. Black Box Interpretation of a LOTOS Specification

at gates. We can view the offering of an action to the environment as the popping up of a button. For example, Figure 2.4 depicts the situation when an action is offered at gate g, but not at gate h. The environment can decide to push the button or to leave it unpushed. We could also have situations such as that depicted in Figure 2.5, where both buttons are up and the external observer has a choice of actions to perform.
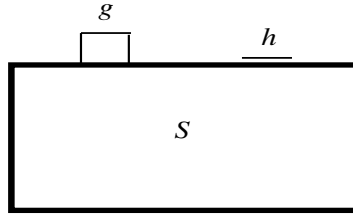

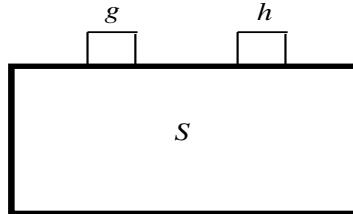**Fig. 2.4.** Action Offering as Buttons Popping Up

Fig. 2.5. Choice of Action Offers

We use this button-pushing analogy a number of times in our presentation of LOTOS.

Behaviour Expressions. As indicated already, we introduce pbLOTOS by working through the main constructs of the language. As also indicated already, we are interested in deriving *behavioural* specifications. As a reflection of this, the main unit of pbLOTOS specification is a *behaviour*. The operators that we introduce characterise the possible behaviour expressions that can be written in pbLOTOS. The set of all possible pbLOTOS behaviour expressions is denoted Beh; the variables B, B', B'', B_1, B_2, ... range over the set Beh; i.e. when we refer to such a variable it is implicitly assumed to be in Beh; e.g. $B \in Beh$.

There is one behaviour expression that we can highlight immediately; it is the null behaviour expression,

   stop

which is a distinguished behaviour that performs no actions. In fact, it is synonymous with deadlock. *stop* is typically used to terminate a nonnull behaviour; i.e. it indicates that a point has been reached at which no more behaviour can be performed.

Behaviour Trees. We use a general notation, which we call behaviour trees, in order to depict the allowable evolutions of a behaviour expression. One of the semantics that we consider in the next chapter, *labelled transition systems*, has similarities to behaviour trees and can be seen as a formalisation of some aspects of behaviour trees.[3] Examples of behaviour trees are presented in Figure 2.6. The exact meaning of this graphical notation is made clear as we introduce the LOTOS constructs. Such a representation of behaviour is helpful for simple specifications, but becomes unmanageable when specifications become complex, e.g. if a large amount of recursive behaviour is included in a specification.
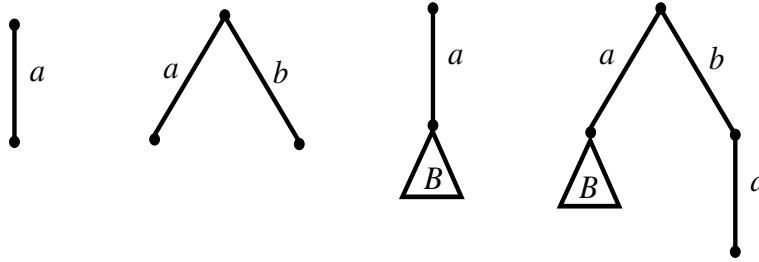

Fig. 2.6. Example Behaviour Trees

---

[3]In fact, labelled transition systems are more general than behaviour trees, because their underlying connectivity can be a graph; i.e. can contain cycles.

2.3.1 Abstract Actions

The first major principle is to assume the existence of a universe of *observable actions* (these are also called *external actions*). For example, in specifying a communication protocol we might assume the following observable actions exist.

- *send*, which references the instant that a message is transmitted from a sender process to a communication medium;
- *receive*, which references the instant that a message is passed from the communication medium to a receiver process;
- *timeout*, which references the instant that a sender process times out waiting for an acknowledgement;
- And similarly, *sendAck*, *receiveAck*, *get*, *put* etc;

and, in specifying the Dining Philosophers problem, we might assume the following observable actions:

- *pick*, which references the instant that a chopstick is picked up off the table; and
- *put*, which references the instant that a chopstick is put back onto the table.

The set of all such actions is denoted *Act*; i.e. this is the set of all possible actions that can be written; this set will clearly be infinite. *Act* is sometimes called the alphabet of actions. The variables u, v, x, y, z and their super- and subscripts, e.g. x', x'', x_1, x_2, $\ldots$, range over *Act*.[4] However, although *Act* is infinite, the set of actions used in a particular pbLOTOS specification is finite; i.e. a finite subset of *Act*. Assuming that a particular pbLOTOS specification is being considered, the set of all actions in the specification is denoted $\mathcal{L}$; i.e. the labels arising in the specification.

In pbLOTOS, actions and gates are synonymous. This is because no data is passed as part of an action, so, the name of the gate at which an action is performed completely defines the action performed at that gate. As a reflection of this, for pbLOTOS, the terms gate and action can be used interchangeably.

It is important to note that actions are *atomic*; they are atomic units of observation and cannot be divided in time. A consequence of this is that no two actions can occur at the same time and, thus, the occurrence of two actions cannot overlap. For example, a *send* and a *sendAck* or a *pick* and a *put* cannot happen at the same time. The atomicity of actions clearly has important consequences for the modelling of concurrency; we discuss these consequences in Section 2.3.6.

The restriction to atomic actions does not limit expressiveness, because nonatomic activities can be specified in terms of the actions that delimit the

---

[4]Not only is this convention employed in this chapter, it is followed throughout the book, unless otherwise stated.

activity; i.e. rather than defining an action that has duration, we can specify the atomic instant at which the activity starts and the atomic instant at which it stops. For example, rather than specifying that a philosopher eats, we specify that at some instant he starts eating (which could be marked with an action *pick*) and at some instant he stops eating (which could be marked with an action *put*).

Actions are a fundamental abstraction device. Systems are described in terms of such abstract entities rather than physical realisations; e.g. a communication protocol is described in terms of abstract actions rather than the physical mechanisms that realise the tasks of sending, receiving, timing-out etc.

A special distinguished action, i, is also used; it denotes an *internal action*; i.e. an action that is hidden from the external observer. The occurrence of an internal action is not externally visible. Thus conceptually, no button is raised when it is offered or pushed when it is performed. It is important to note though that although an i action is not externally visible, it may "indirectly" affect behaviour that is externally visible. Typically, an i action will represent an internal decision, resolution of which prescribes a particular visible behaviour.

The internal action has a number of roles. Firstly, it enables *information hiding*; actions that are observable at one level of specification can be transformed into hidden actions at another level. Thus, behaviour that should not be visible can be hidden. Such hiding supports a form of abstraction, because the complexity of a part of the system is abstracted away from, by hiding it, when specifying another part. In addition, internal actions play a central role in creating nondeterminism; see Section 2.3.4.

Internal actions also prove to be important when (behavioural) equivalences are defined. In particular, two specifications with different internal behaviour may achieve the same "observable" behaviour and could, thus, be considered equivalent.

Observable actions can be transformed into i using a hiding operator, which takes the form:

$$hide \ x_1, \ldots, x_n \ in \ B$$

and states that wherever any of the actions $x_1, \ldots, x_n$ arise during the evaluation of the behaviour B they will be replaced by i. Thus, the gates $x_1, \ldots, x_n$ are removed from the interface of behaviour B. For example, if we assume B' models the behaviour of a sending process and contains an action *timeout*, we might wish to hide the *timeout* from all observers outside the sender; i.e.

$$hide \ timeout \ in \ B'$$

This hiding reflects the reality of networked communication, where, for example, the receiver process would be unable to observe a timer expiring in the sender. We use a, b, c, d, e and their super- and subscripts, e.g. a', a'', a_1, a_2, ..., to range over $Act \cup \{i\}$.

Actions are the basic unit of LOTOS specification and, typically, when performing a formal description using LOTOS, a set of actions in the problem domain would be located. Having identified the constituent actions of the specification we would like to order them in someway, i.e. to define the "temporal order" in which actions can occur (after all this is what basic event ordering models are about). The pbLOTOS operators allow us to do this. Thus, we postulate a universe of actions and then order them according to a set of primitive operators. Standard operators are: *sequence*, *choice*, *process instantiation* and *concurrency*.

2.3.2 Action Prefix

Basic sequencing of actions is defined in LOTOS using *action prefix*, which has the general form

$a \, ; B$

where a is an action from $Act \cup \{i\}$ and B is a behaviour. Thus, $a \, ; B$ is a behaviour that will perform action a and then behave as B. We can depict the effect of this construct using the behaviour tree shown in Figure 2.7. Thus, action offers are attached to line segments in behaviour trees and unspecified behaviour, such as B, is depicted using a triangle.

In terms of pushing buttons, we can also view $a \, ; B$ as a black box with a gate a (and gates for all the external actions in B). The button a is initially the only button raised; if the environment pushes a then the black box behaves as B (e.g. new buttons will be raised).

Fig. 2.7. A General Behaviour Tree Depicting Action Prefix

As an example, we may wish to specify that our medium process will perform a *send* action with the sender process and then perform a *receive* action with the receiver process (this behaviour is depicted in Figure 2.8):

$send \, ; receive \, ; stop$

Notice the use of the distinguished behaviour *stop* to terminate the action offering of the sender. This behaviour states that the action *receive* cannot happen before the action *send* and, following the action *receive*, no more
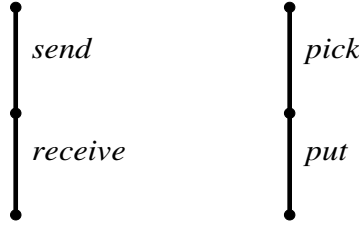
Fig. 2.8. Behaviour Trees of Action Prefix

actions will be offered. By way of clarification, this behaviour can be derived from the general form for action prefix by repeated application. In fact, as a reflection of this, the behaviour is actually a shorthand for the following fully bracketed behaviour

  $send\,;\,(\,receive\,;\,(\,stop\,)\,)$

where the repeated application is made explicit.

   Alternatively, we might want to specify the following behaviour (depicted in Figure 2.8) for a dining philosopher

  $pick\,;\,put\,;\,stop$

indicating that a philosopher cannot put his chopstick down until he has picked it up.

2.3.3 Choice

Choice is denoted

  $B_1\ []\ B_2$

and states that either behaviour B_1 or behaviour B_2 will be performed. The choice of which behaviour to perform is determined by the initially offered action of the two behaviours. Typically, all such actions will be offered to the environment, which will choose which to perform; this decision will resolve the choice.

   The necessity to offer such choices largely arises because of the move to systems that contain concurrency. A behaviour offering a choice of a number of observable actions to perform is really offering a menu of possible inter-actions from which concurrently executing objects can select. The behaviour is defining the set of actions to which it is willing to react. Such choices are not typically associated with sequential systems which are, in comparison to parallel systems, closed. The interaction choices between components are pre-determined in sequential systems.

   As an example of choice, we may wish to specify the sender behaviour depicted to the left in Figure 2.9:
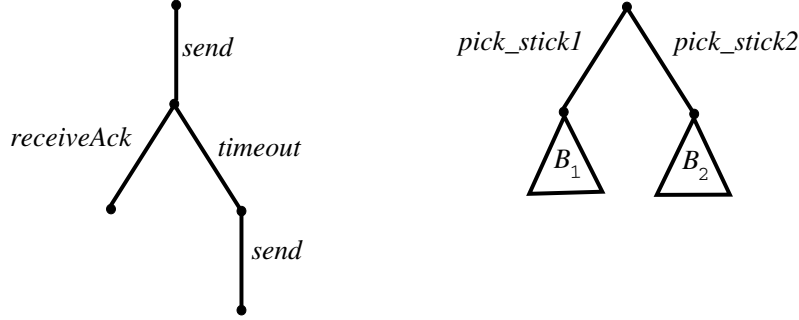
Fig. 2.9. Examples of Choice in Behaviour Trees

$$send\,;\,(\,receiveAck\,;\,stop\,\rlap{\,}[]\,\,timeout\,;\,send\,;\,stop\,)$$

This states that after a *send* the sender will either receive an acknowledgement or time out and retransmit, by performing another *send*. Each of the alternatives is completed by stopping.

We can also picture choice in terms of buttons popping up. For example, this behaviour yields a black box with gates *send, receiveAck* and *timeout* and it is initially in the state depicted in Figure 2.10(i). If the environment performs a *send* then the box progresses to the state depicted in Figure 2.10(ii). So, there is now a choice for the environment: does it press *receiveAck* or *timeout*? (In fact, in more advanced versions of this behaviour we hide *timeout* and do not make this choice externally visible, but for illustrative purposes we leave it visible here.) If the environment presses *receiveAck* no more actions will be offered; i.e. all buttons will be depressed. However, if *timeout* is pressed, the send button pops up and we progress to the (external) state depicted in Figure 2.10(i).

This is only a snapshot of the full behaviour of the sender and is far from complete. For example, after timing out we would actually like to specify that the behaviour recurses back to the start in order to resend. We have to wait until we have a few more constructs before we can express such behaviour.

In a similar way, we could specify the behaviour depicted on the right in Figure 2.9 as

$$pick\_stick1\,;\,B_1\,\rlap{\,}[]\,\,pick\_stick2\,;\,B_2$$

i.e. a philosopher can either pick up stick 1 or stick 2.

2.3.4 Nondeterminism

Nondeterminism goes hand in hand with concurrency. Because, in concurrent systems, components can evolve independently of one another, choices made
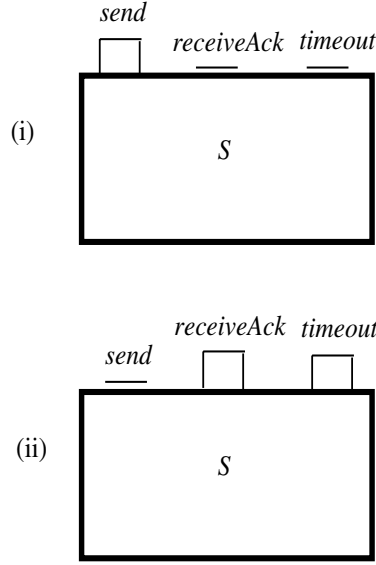
Fig. 2.10. Examples of Choice in Black Boxes

inside one component can create nondeterminism for the component's environment (i.e. all components that evolve in parallel with it). This is because components cannot "look inside" other components to see why they make a particular choice, thus, to the environment, hidden choices seem nondeterministic.

Another way of viewing this is that components are autonomous and thus, they make decisions for themselves, which are not "explained" to their environment. This does not mean that overall behaviour is nondeterministic; the emergent behaviour could be deterministic. Specifically, it will be deterministic if the environment can handle all the nondeterministic possibilities, i.e. if nothing is unexpected. Although many hidden choices are taking place in a car engine, (while faults do not occur) its emergent behaviour is predictable, once it has been explained to the driver by reading the car manual or passing a driving test.

Nondeterminism is defined in LOTOS as a special case of choice. Specific forms of choice yield a nondeterministic resolution of the alternatives. The main forms are:

(i) $i\,;\,B_1\;[]\;i\,;\,B_2$

(ii) $i\,;\,B_1\;[]\;x\,;\,B_2$

(iii) $x\,;\,B_1\;[]\;i\,;\,B_2$

(iv) $x\,;\,B_1\;[]\;x\,;\,B_2$

where x denotes an observable action and (ii) and (iii) are mirror images of each other; so, there are really three basic forms. Notice that these first three classes of nondeterminism could be created by hiding some actions in an otherwise deterministic behaviour. In addition, parallel composition can create nondeterminism, as we discuss in Section 2.3.6. The three basic forms are depicted in Figure 2.11.
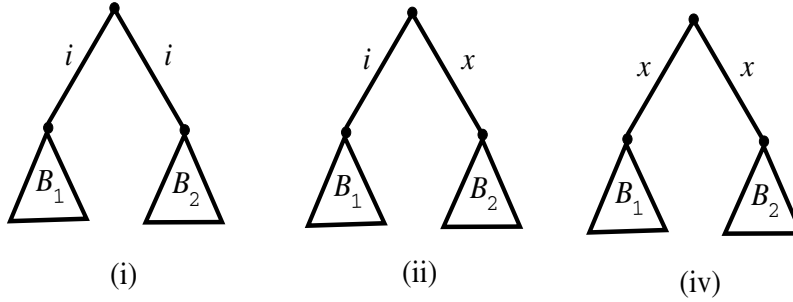

Fig. 2.11. General Forms of Nondeterminism in Behaviour Trees

The nondeterminism arises because selection between the two initial actions of the choice is beyond the control of the environment. For example, in (iv), when the external observer performs an x he or she has no control over whether the specification evolves to B_1 or to B_2. As a reflection of this, a nondeterministic choice is also referred to as an internal choice.

Each of these three forms yields a different variant of nondeterministic behaviour. Firstly, notice that forms (i) and (iv) are symmetric, while (ii) is nonsymmetric, in the sense that the left branch starts with an internal action, while the right branch starts with an observable action. We now consider each in turn.

- In (i), the initial evolution of the behaviour is completely hidden from the external observer; in terms of button-pushing, no buttons are raised. Thus, a wholly internal choice will be made to either evolve to behaviour B_1 or to evolve to behaviour B_2.
- In (ii), the initial evolution could also be completely hidden from the external observer; i.e. the left branch could be taken immediately and no buttons will be raised. However, if the external observer is quick enough to interact with the behaviour she could perform action x and evolve to B_2. However, if the external observer is either not quick enough or unable to perform an x, the behaviour will eventually evolve to B_1. Conceptually, the button x is raised, to see whether the environment can push it, and then, at some point, retracted (i.e. depressed). Critically though, because we are not yet in the business of quantitative time specification, the time

point at which x is retracted is not stated. Effectively, the specification says that, if the environment has not performed the x by some unspecified time point, it will be retracted.

- In (iv), the point of initial evolution of the behaviour is always externally visible; i.e. an x action will be offered and the corresponding button will be raised. However, the choice of evolving to B_1 or to B_2 after performing x is made internally and hence nondeterministically.

It is important to note the difference between a deterministic choice (sometimes referred to as an external choice) and a nondeterministic choice. For example, you should convince yourself that the following two behaviours, which are depicted in Figure 2.12, are different.
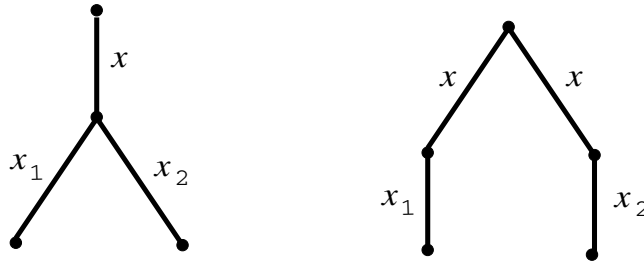

Fig. 2.12. A Deterministic and a Nondeterministic Choice

$$x\,;\,(\,x_1\,;\,stop\,[]\,x_2\,;\,stop\,)\quad\text{and}\quad x\,;\,x_1\,;\,stop\,[]\,x\,;\,x_2\,;\,stop$$

Specifically, after performing an x action, the first behaviour will offer an external choice between performing an x_1 or an x_2, whereas, after performing an x action, the second behaviour will offer one of x_1 or x_2 to the environment, but, crucially, not the choice between both. In terms of button-pushing, we can depict the two behaviours as the two alternative sequences of black box states depicted in Figure 2.13, where the arrows indicate evolution of the system and, in particular, the two arrows in the nondeterministic black box indicate a choice of internal evolution.

Nondeterminism plays a number of roles in process calculi. In general it acts as an abstraction device. For example, nondeterminism is often introduced when, at a certain level of system development, we wish to abstract away from a particularly complex mechanism. A good example of this is in modelling loss in a communication medium. For example, the medium in our running example might be specified as follows,
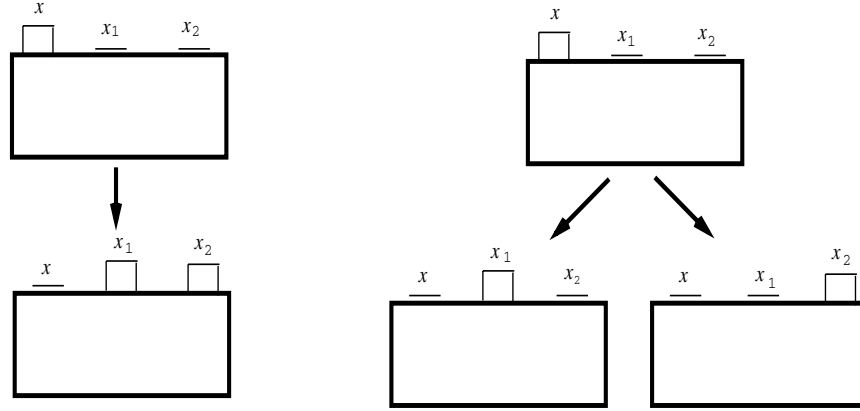
$$send\,;\,(\,i\,;\,B_1\,[]\,receive\,;\,B_2\,)$$

Fig. 2.13. Deterministic and Nondeterministic Choice in Black Boxes

which will perform a *send* action with the sender process (i.e. a message is sent to the medium) and then it will either nondeterministically decide to lose the message, represented by the i action, or pass the message on, represented by offering the *receive* action, with which the receiver process may interact.

What we are really doing here is abstracting away from the specific mechanism by which loss occurs in a communication medium. We are stating that some internal mechanism could occur and result in the message being lost, but, at the particular level of abstraction we are considering, we are not interested in how this happens. Notice that a complete specification of the mechanics of loss would probably require the physical laws of noise and attenuation on communication lines to be expressed.

There is also a sense in which nondeterminism is used in specification to allow implementation freedom. A nondeterministic choice between evolving to B_1 or to B_2 can be viewed as stating that implementations that behave as either B_1 or B_2 are satisfactory. Such a specification is stating that the specifier does not mind whether the system behaves as B_1 or as B_2. Such nondeterminism may then be refined out during development. This is the motivation behind refinement relations, such as reduction; see Section 5.1.6.2.

2.3.5 Process Definition

Basic Form. The basic unit of modularity is the process. The syntax for process definition is:

$$P\,[x_1, \ldots, x_n] := B$$

where $P \in PIdent$, the set of process identifiers. This states that the process identifier P is bound to the behaviour B. The list $x_1, \ldots, x_n$ indicates the

actions that are observable in B. $[x_1, \ldots, x_n]$ can be thought of as denoting the interface of B, i.e. the actions that can be interacted with; it defines the buttons that must be made available in a black box implementation of the process P.

Instantiation of the behaviour B is performed through reference to P in a behaviour expression (we also talk about invocation of processes; the terms instantiation and invocation are interchangeable). In the process of instantiating P, we can alter the action names. For example, the definition

$P[x, y, z] := B$ (a)

could be invoked by referencing

$P[u, x, w]$ (b)

which has the effect of instantiating behaviour B in such a way that, whenever it is specified to offer an x it offers a u, whenever it is specified to offer a y it offers an x and whenever it is specified to offer a z it offers a w. The terms *formal* and *actual* gates, are often used to refer to these action lists. Thus, in the above example, x, y and z are formal gates of the process P, whereas u, x and w are actual gates of the process instantiation.

In terms of black boxes, the definition of P, expression (a), can be depicted as in Figure 2.14, whereas instantiation of P, expression (b), can be depicted as in Figure 2.15.
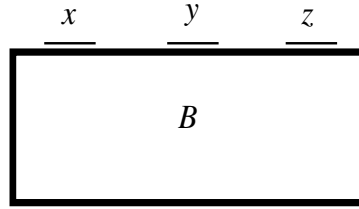
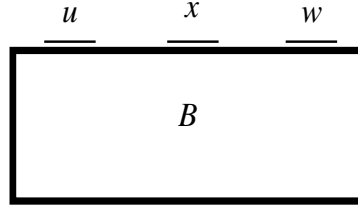

Fig. 2.14. Process Definition Black Box

Notice also that, in $P[x, y, z] := B$, the behaviour B can reference P and thus create recursion. As an example, consider the behaviour

$P[z, w] := z\,;\, w\,;\, P[w, z]$

which, on invocation as

$P[x, y]$

yields the infinite behaviour depicted in Figure 2.16. Infinite behaviour is created by recursive process invocation. Notice also, recursive behaviour cannot

Fig. 2.15. Process Instantiation Black Box

be finitely represented in a tree. Thus, to denote such executions, we need to use what could be called behaviour graphs as a generalisation of the trees used to this point.
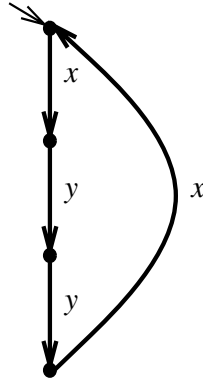
Fig. 2.16. An Infinite Behaviour

Sometimes when we use process definition and instantiation we drop the action list. So, we write

$$P := B$$

as a shorthand for

$$P\,[x_1, \ldots, x_n] := B$$

We only use this shorthand when the action list plays no role; i.e. action names are not renamed on process instantiation. Thus, a process definition such as

$$P := x\,;\, y\,;\, stop$$

cannot be invoked as

$$P\,[z, w]$$

Example. As an example of process definition and instantiation, the medium in the communication protocol example could be specified as a process called $Medium$ and defined as follows.

$Medium[send, receive]\ :=$
$\quad send\,;\,(\,i\,;\,Medium\,[send, receive]\,[]\,receive\,;\,Medium\,[send, receive]\,)$

This is actually a one-slot medium; i.e. it deals with one message at a time. A one-slot medium is suitable for a stop and wait protocol because only one message is in transit between the sender and receiver at any time. The process named $Medium$ performs a $send$ action with the sender. Then the medium will either nondeterministically lose the message or offer the action $receive$ with which the receiver may interact. After either of these alternatives, the behaviour recurses by invoking $Medium$ again, thus preparing the process for the next $send$ action from the sender.

We can also specify a possible behaviour for the sender process as follows.

$Sender\,[get, send, receiveAck] :=$
$\quad get\,;\,send\,;\,Sending\,[get, send, receiveAck]$

$Sending\,[get, send, receiveAck] :=$
$\quad hide\ timeout\ in\ (\ receiveAck\,;\,Sender\,[get, send, receiveAck]$
$\qquad\qquad\qquad\qquad []$
$\qquad\qquad\qquad timeout\,;\,send\,;\,Sending\,[get, send, receiveAck]\,)$

The top-level process here is $Sender$, which invokes the process $Sending$. We use the convention that process identifiers are written with a capital first letter, whereas action names are written with a small first letter. The process $Sender$ obtains a message to deliver by performing the action get (remember this interaction takes place between the $Sender$ and its environment); it transmits the message by performing $send$ and then it invokes $Sending$.

The role of $Sending$ is to ensure successful transmission of the message sent. In order to do this, $Sending$ waits for an acknowledgement (the $receiveAck$) action; if it does not arrive in time a timeout occurs and the message is resent, modelled by offering the action $send$ again. Notice that, if a $receiveAck$ is successfully received then the recursive call takes us back to $Sender$, indicating that the message has been successfully transmitted, and we are ready to send another message. In contrast, after timing out and resending, we recurse back to the start of $Sending$ and try for an acknowledgement to the resend.

Although they are the same actions, the two references to $send$ are conceptually different: the $send$ in $Sender$ is an initial transmission, whereas the $send$ in $Sending$ is a retransmission of an old message. This distinction is justified because an initial transmission $send$ is preceded by the action get.

You can think of the effect of the action *get* as being to fill the send buffer with a new message.

In this example, you should notice the approach of invoking a subprocess, which enables a repetition to be set up by recursing on the name of the subprocess. The definition and invocation of the process *Sending* is just such an example. In terms of state machines, *Sending* can be viewed as a state back to which the machine iterates. In fact, all constituent behaviours of a pbLOTOS specification can be viewed as states. For example, the behaviour expression:

$$get\,;\,send\,;\,Sending\,[\ldots]$$

can be viewed as a state from which a transition labelled *get* can be performed and the system evolves into state,

$$send\,;\,Sending\,[\ldots]$$

which is a state from which a transition labelled *send* can be performed and the system evolves into state,

$$Sending\,[\ldots]$$

To complete the set of processes in the communication protocol example, the receiver process and acknowledgement mediums can be specified as follows:

$$Receiver\,[put, receive, sendAck] :=$$
$$receive\,;\,put\,;\,sendAck\,;\,Receiver\,[put, receive, sendAck]$$

$$AckMedium\,[sendAck, receiveAck] :=$$
$$sendAck\,;\,receiveAck\,;\,AckMedium\,[sendAck, receiveAck]$$

As indicated earlier, we have assumed a reliable acknowledgement medium. Thus, the receiver simply receives messages and sends acknowledgements and the acknowledgement medium passes these messages on (and does not lose any of them).

Divergence. An important technical issue arising through recursion is the possibility of infinite internal behaviour, which is called *divergence*. The following behaviours give different examples of the phenomenon.

1. $P := i\,;\,P$
2. $P := x\,;\,stop\,[]\,i\,;\,P$
3. $P := x\,;\,stop\,[]\,y\,;\,hide\ y\ in\ P$
4. $hide\ x\ in\ (\,y\,;\,B\,[]\,z\,;\,P\,)$ where $P := x\,;\,x\,;\,P$
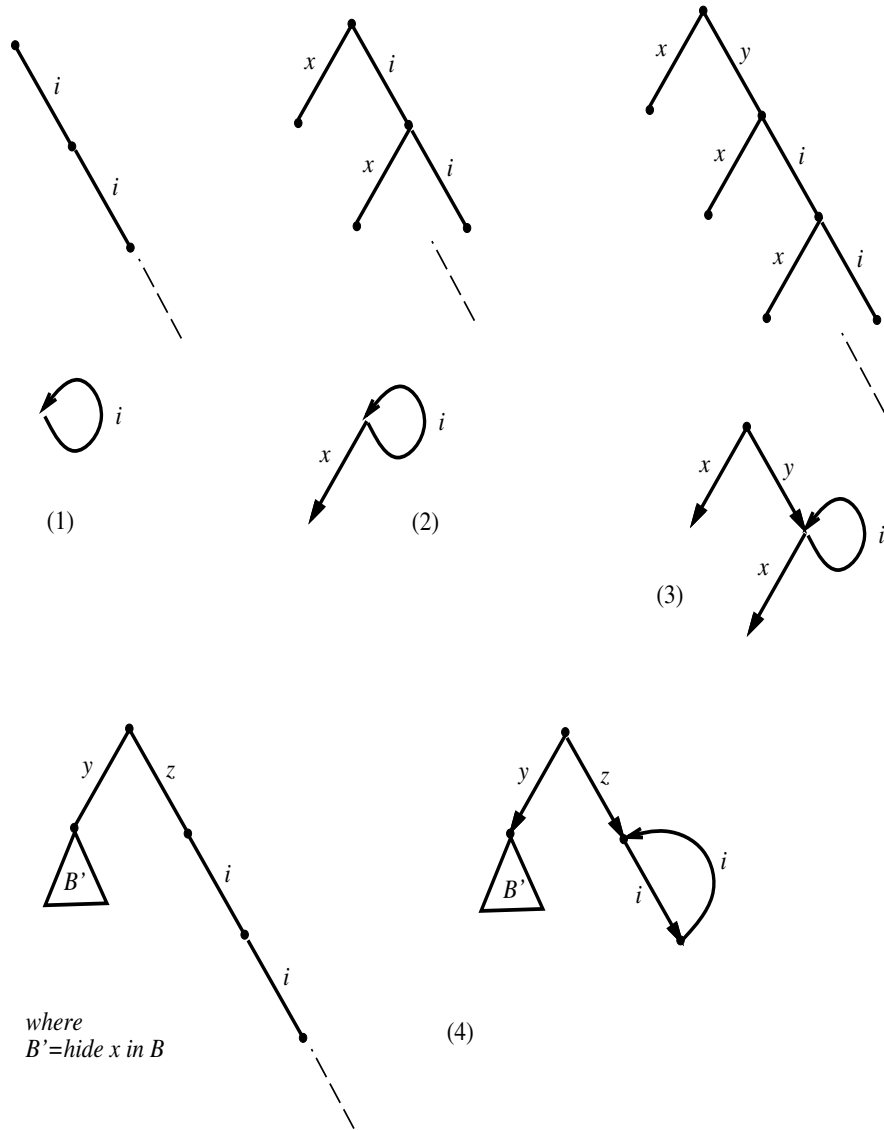
where
B'=hide x in B

Fig. 2.17. Divergent Behaviour

The behaviour of these expressions is depicted in Figure 2.17; both infinite expansions and cyclic depictions are presented.

Thus, (1) is a straightforward form of divergence; (2) offers the possibility, at every state, of not diverging by performing an x, but nonetheless it could diverge if the environment is never willing to perform an x; (3) shows how hiding can create divergence; and (4) shows how hiding can create divergence deep inside a subprocess.

The correct interpretation of divergence is a hotly debated issue, with one school of thought viewing divergence as degenerate in the extreme [96]. For example, extreme cases of divergence, where the recursive call is not even "guarded" by an internal action, are certainly problematic, viz:

$$P := P$$

We discuss these issues in some depth in Sections 5.1.4 and 7.2.6.

Relabelling. In order to correctly model the effect of process instantiation, another operator is required called *relabelling*. This has the form,

$$B\,[y_1/x_1, \ldots, y_n/x_n]$$

and has the effect of relabelling the x_is with the y_is in the behaviour B (i.e. buttons are renamed). Notice that $x_i \in Act$ for all $1 \leq i \leq n$, so internal actions cannot be relabelled; this also applies to the special (pseudo internal) action δ, which we introduce in Section 2.3.7. In the standard LOTOS language, the relabelling operator is not available to the specifier, rather it is used in defining the semantics of the language. This is because the basic form, i.e. renaming through binding actual gate names to formal gate names is viewed to be more usable from the specifier's point of view. However, the basic form is really just syntactic sugar for direct application of the relabelling operator. In particular, an invocation,

$$P\,[y_1, \ldots, y_n]$$

of a process definition,

$$P\,[x_1, \ldots, x_n] := B$$

can always be rewritten using our simplified form of process instantiation and relabelling, i.e.,

$$P\,[y_1/x_1, \ldots, y_n/x_n]$$

with a process definition,

$$P := B$$

This approach of completely dividing the mechanisms for process invocation and the mechanisms for relabelling leads to more elegant semantic definitions and is thus, generally used in our chapters on semantics. However, the basic form is kept for presentation of examples.

2.3.6 Concurrency

2.3.6.1 Independent Parallelism

We begin with a special case of concurrency; this is given by the operator |||, and has the general form,

$B_1 ||| B_2,$

which states that the two behaviours B_1 and B_2 evolve independently in parallel. Independent in this context means that there is no shared behaviour, which would arise if B_1 and B_2 performed some actions together.

   We might, for example, use this construct to specify that the behaviour of two philosophers that do not share chopsticks are independent:

$pick\_stick1$; $pick\_stick2$; put_stick1 ; put_stick2 ; $stop$ |||

$pick\_stick3$; $pick\_stick4$; put_stick4 ; put_stick3 ; $stop$

Of course, if they share a chopstick there would be some overlapping behaviour; we come to this situation shortly.

   How though should we view such independent behaviour? In fact, the choice of interpretation to put on parallelism is one of the main issues in our discussion of semantics. However, here we focus on what is the standard interpretation: *interleaving*.

   Interleaved interpretations of concurrency are justified by our assumption that all actions are atomic. Specifically, as discussed earlier, a direct consequence of actions being assumed to be atomic is that no two actions can occur simultaneously. Thus, in terms of action occurrences, there is no true simultaneity and any execution path through the specification will be a linear sequence of actions. As an illustration, consider the following simple example,

x ; $stop$ ||| y ; $stop$

This behaviour specifies that the action x will be offered independently in parallel with the action y. Now, assuming atomicity of actions, we know that the occurrences of x and y cannot overlap, which implies that one must occur before the other. So, we obtain the following relationship,

x ; $stop$ ||| y ; $stop$ $\equiv$ x ; y ; $stop$ [] y ; x ; $stop$

where $\equiv$ means "are equivalent" (we make precise such notions of equivalence later in this book). This states that x occurring in parallel with y is the same as either the occurrence of x being followed by the occurrence of y or the occurrence of y being followed by the occurrence of x. Thus, interleaving allows parallelism to be expressed in terms of sequence and choice and although behaviours may be "truly in parallel", no two actions occur "truly" simultaneously. This interpretation allows us to depict independent concurrency very easily; a depiction of x ; $stop$ ||| y ; $stop$ is given in Figure 2.18.

   Figure 2.19 shows the following larger example of interleaved parallelism.
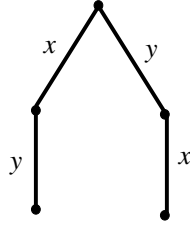
Fig. 2.18. Interleaved Parallelism

$(x \,;\, y \,;\, stop) \,|||\, (y \,;\, z \,;\, stop)$
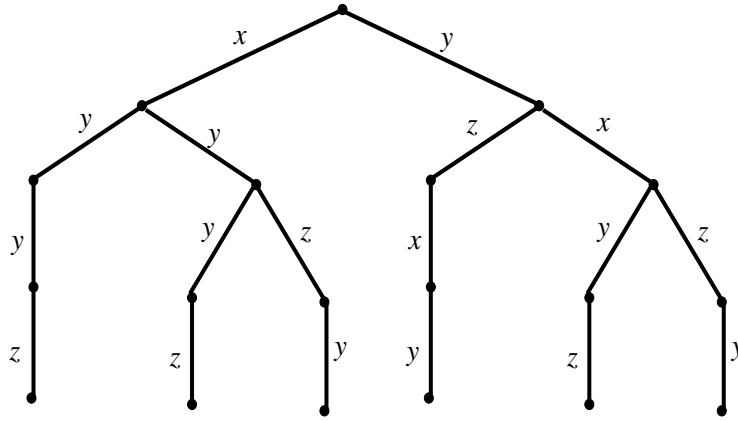
Fig. 2.19. Interleaved Parallelism 2

We have once again mapped concurrency to sequence and choice, but this time the possible alternatives are far greater. This is characteristic of interleaving: the number of states in the interleaved representation increases very rapidly as the complexity of the parallel behaviour increases. Also, notice that the y actions on both sides of the parallel behaviour occur independently. These two actions would have to be explicitly identified for synchronisation if they were to occur together. In addition, notice that some subbranches of the complete behaviour are repeated. Such repetitions could be pruned out.

Nondeterministic choices can also be created through parallel composition. For example, if the environment performs the action x, then the behaviour,

$(x \,;\, stop) \,|||\, (x \,;\, y \,;\, stop)$

will internally decide whether to offer just an x and then a y or an external choice between y and x, with the former evolving to offering an x and the

latter evolving to offering a y. This behaviour is depicted in Figure 2.20. You should also notice that a similar instance of nondeterminism is embedded into the behaviour depicted in Figure 2.19; the nondeterminism is on the action y. Forms of nondeterminism based on internal actions can also be created through parallel composition.
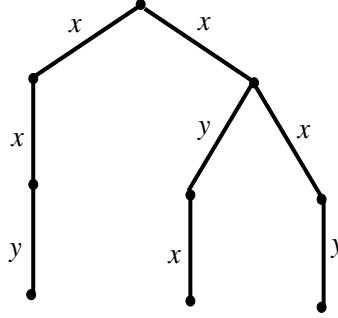

Fig. 2.20. Interleaving Creating Nondeterminism

2.3.6.2 General Form

As already stated, independent parallelism is one specific class of concurrent behaviour. Concurrency, in its most general form, is denoted

$$B_1 \ |[x_1, \ldots, x_n]| \ B_2$$

with the operator parameterised on the observable actions that must be synchronised, the $x_1, \ldots, x_n$. This behaviour states that B_1 and B_2 evolve independently in parallel subject to the synchronisation of actions $x_1, \ldots, x_n$; i.e. an action x_i $(1 \leq i \leq n)$ appearing in either B_1 or B_2 can only be executed if it synchronises with an x_i in the other behaviour. Also notice that internal actions cannot synchronise; this is because they are not externally visible and, thus, cannot be interacted with.

As examples, consider the following behaviours.

(i) $(x\,;\,y\,;\,stop)\ |[y]|\ (y\,;\,z\,;\,stop)$

(ii) $(x\,;\,y\,;\,stop)\ |[x]|\ (x\,;\,z\,;\,stop)$

(iii) $(x\,;\,stop)\ |[y]|\ (z\,;\,stop)$

(iv) $(x\,;\,y\,;\,stop)\ |[y]|\ (z\,;\,stop)$

(v) $(x\,;\,y\,;\,w\,;\,stop)\ |[x,y]|\ (x\,;\,z\,;\,stop)$

(vi) $(x\,;\,y\,;\,stop)\ |[y]|\ (i\,;\,z\,;\,stop)$

Their behaviour trees are depicted in Figure 2.21. For each of the behaviours, independent parallelism is constrained by synchronisation of the actions in the gate set.

- (i) Synchronisation on y results in a totally sequential behaviour. In particular, notice that the y on the right-hand side of the behaviour cannot occur without the left-hand side offering a y with which to synchronise. Thus, the x must be performed first.
- (ii) Synchronisation on x still allows y and z to be arbitrarily interleaved once x has occurred.
- (iii) The interleaving of the two behaviours is unconstrained because y does not appear in either behaviour.
- (iv) The interleaving of x and z is unconstrained, but because the y action is identified for synchronisation and does not appear on both sides of the parallel operator, it does not occur. The behaviour on the left-hand side is, in fact, unable to proceed once the x action has been performed; i.e. it is locally deadlocked.
- (v) Firstly, the x action is synchronised on, then the z action can occur, but, once again, the y action is deadlocked. The inability to perform a y blocks the w action from being offered as well.
- (vi) The interleaving of x and i is unconstrained by the synchronisation on y and the z action can follow the occurrence of i. However, the y action is blocked from being offered.

Generalised parallelism, $|[\ldots]|$, has two special cases, one of them we have seen already:

$$B_1 \;|||\; B_2,$$

which is equivalent to writing, $B_1 \;|[\ ]|\; B_2$, i.e. general parallel composition with an empty synchronisation set; and,

$$B_1 \;||\; B_2$$

which is equivalent to generalised parallelism with a synchronisation set containing all the actions of B_1 and B_2 or alternatively containing all the actions in $\mathcal{L}$. Thus, $|||$ gives the composition of independent concurrent threads and $||$ gives fully synchronised parallelism.

As illustrations of fully synchronised parallelism, consider the following behaviours (behaviour trees for which are presented in Figure 2.22).

(i) $(\,x\,;\,y\,;\,stop\,) \;||\; (\,x\,;\,y\,;\,z\,;\,stop\,)$

(ii) $(\,x\,;\,y\,;\,z\,;\,stop\,) \;||\; (\,z\,;\,y\,;\,z\,;\,stop\,)$

(iii) $(\,x\,;\,stop\,[]\,y\,;\,stop\,[]\,z\,;\,w\,;\,stop) \;||\; (\,x\,;\,stop\,[]\,i\,;\,y\,;\,stop\,)$

Thus, (i) can successfully synchronise on x and then y, but then is unable to progress, as the z action is only offered on one side of the parallel composition.
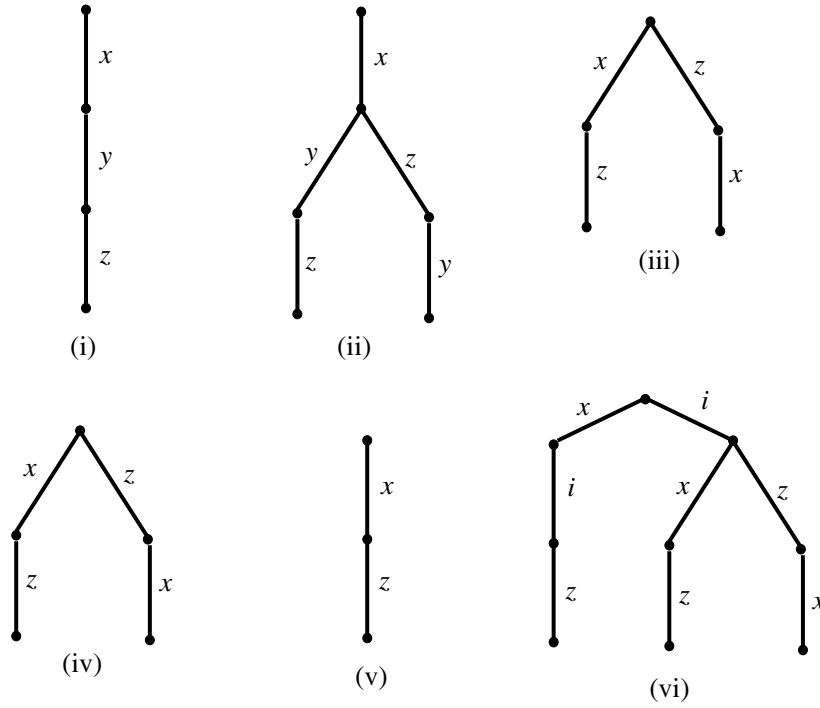
Fig. 2.21. Generalised Parallelism
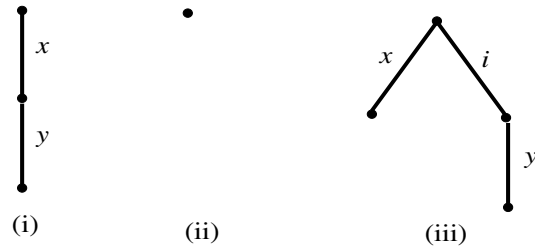
Fig. 2.22. Fully Synchronised Parallelism

In contrast, (ii) can perform no actions and thus induces the indicated trivial behaviour tree, because it cannot even synchronise on an initial action. (iii) illustrates how choice and internal actions behave through fully synchronised parallelism. In particular, notice how one of the branches of the left-hand choice is blocked because z is not offered as an initial action of the right-hand choice. In addition, the internal action is not subject to synchronisation, so appears in the resultant behaviour.

As a reflection of the relationship between generalised parallelism and the two operators $|||$ and $||$, only $|[\ldots]|$ is included in pbLOTOS; $|||$ and $||$ are viewed as derived operators; i.e. $||| = |[\ ]|$ and $|| = |[x_1, \ldots, x_n]|$ where $\{x_1, \ldots, x_n\} = \mathcal{L}$.

2.3.6.3 Example

As a more concrete illustration of parallel composition, in the communication protocol example we might compose the two mediums together to form a duplex medium as follows,

$$DupMedium\,[send, receive, sendAck, receiveAck] :=$$
$$Medium\,[send, receive] \,|||\, AckMedium\,[sendAck, receiveAck]$$

This states that the behaviour of the two mediums is independent, which is as expected, because the two directions of communication do not affect each other. We can now define the top-level behaviour of the protocol as follows:

$$(\,(\,Sender\,[get, send, receiveAck] \,|||\, Receiver\,[put, receive, sendAck]\,)$$
$$|[send, receive, sendAck, receiveAck]|$$
$$DupMedium\,[send, receive, sendAck, receiveAck]\,)$$

So, the *Sender* and *Receiver* processes evolve independently, but communicate by synchronising on the commmon gates $send, receive, sendAck$ and $receiveAck$ through the duplex medium.

2.3.6.4 Why Synchronous Communication?

As indicated already, in LOTOS concurrent threads of control interact by *synchronous communication*. Synchronous message passing is chosen because it can be viewed as the primitive mechanism from which other communication paradigms, e.g. asynchronous communication or remote procedure call, can be defined.

The particular class of synchronous communication employed in LOTOS is *multiway* synchronisation. Thus, any number of behaviour expressions can be involved in a synchronous communication. For example, in the following behaviour,

$$P\,[x, y] \,|[x]|\, Q\,[x] \,|[x]|\, R\,[x, z]$$

all three of the processes, P, Q and R, have to synchronise in order to perform the action x.

The LOTOS multiway synchronisation plays an important role in the *constraint-oriented* style of specification [171, 195]. The term constraint-oriented is used to refer to an incremental system development style in which system specifications are refined by imposing behavioural constraints on the

system. This is done by composing the system in parallel with a piece of behaviour, which reflects this constraint. It is suggested that such an approach offers a powerful incremental development methodology [195].

You should also be aware of the important role that hiding plays in relation to multiway synchronisation. Specifically, hiding is used to "close off" an interaction and prevent further synchronisation on a particular action. This implies a very specific order to the application of operators in constraint-oriented styles of specification. In particular, an interaction cannot be hidden until the behaviour has been fully constrained through parallel composition.

2.3.7 Sequential Composition and Exit

Action prefix defines sequencing for actions, however, we would also like to define sequential composition of complete behaviours. This is supported by the *sequential composition* operator (also called *enabling*),

$$B_1 >> B_2$$

which will evolve as B_1 then; if B_1 terminates successfully, it will behave as B_2. The concept of successful termination is pivotal here. We do not wish $B_1 >> B_2$ to evolve to B_2 unless B_1 completes its evolution. In particular, if B_1 is in a deadlock state we would wish $B_1 >> B_2$ to evolve to the same deadlock state. Thus, we introduce a special distinguished behaviour,

$$exit$$

to denote successful termination. For example, consider the following behaviour expressions.

(i) $(x \,;\, y \,;\, exit) \;>>\; (z \,;\, stop)$

(ii) $(x \,;\, y \,;\, stop) \;>>\; (z \,;\, stop)$

(iii) $(x \,;\, stop \,[] \, y \,;\, exit) \;>>\; (x \,;\, stop)$

(iv) $(x \,;\, exit \,|||\, y \,;\, exit) \;>>\; (z \,;\, stop)$

(v) $(x \,;\, stop \,|||\, y \,;\, exit) \;>>\; (z \,;\, stop)$

(vi) $(x \,;\, exit \,|[x]|\, y \,;\, exit) \;>>\; (z \,;\, stop)$

(vii) $x \,;\, y \,;\, exit$

Behaviour trees for expressions (i) to (vi) are depicted in Figure 2.23 and expression (vii) is shown in Figure 2.24. We consider each of the examples in turn.

- (i) The left-hand behaviour is performed first (x followed by y), then an internal action is performed (reflecting the successful termination at $exit$) and this is followed by the right-hand behaviour (performing the z action).

- (ii) Only the left-hand behaviour is performed here. This is because the left-hand behaviour does not successfully terminate, i.e. there is no $exit$, so, the right-hand behaviour is not enabled.
- (iii) Only one branch of the choice successfully terminates and thus, only this branch is postfixed with the right-hand behaviour.
- (iv) It is important to note that, in the behaviour on the left side of $>>$ both sides of the parallel composition successfully terminate. Thus, in whatever state the left-hand behaviour terminates, it will be followed by the right-hand behaviour.
- (v) In contrast to (iv), because only one side of the parallel composition concludes with an exit, the behaviour on the left of $>>$ cannot successfully terminate and, thus, the right-hand behaviour cannot follow. This is an important aspect of sequential composition. Both branches of a parallel composition must successfully terminate in order for the whole of a parallel composition to successfully terminate. With some thought you will be able to convince yourself that this correctly reflects the behaviour of concurrent threads of execution.
- (vi) Because it is in deadlock after performing action y, the left-hand side of this behaviour is not able to successfully terminate.
- (vii) The behaviour successfully terminates by performing a δ action (see Figure 2.24). However, there is no behaviour to enable, thus, the δ action is left dangling. δ is a special action used to signal successful termination and, thus, enable a sequential composition. It is really a semantic device, which enables sequential composition to work. We postpone a full discussion of its behaviour until we consider actual semantic approaches. However, you should note that δ cannot be explicitly used by a specifier, thus, $\delta \notin Act$; it is a distinguished event, which has some similarities to i.

As a more concrete example of the use of successful termination, consider the Dining Philosophers example. We might want to specify that a philosopher can only perform the behaviour of putting his chopsticks down once he has performed the behaviour of picking his chopsticks up:

$$(\, pick\_stick1 \, ; \, exit \, ||| \, pick\_stick2 \, ; \, exit \,)$$
$$>> \, (\, put\_stick1 \, ; \, stop \, ||| \, put\_stick2 \, ; \, stop \,)$$

Notice that this expresses that the chopsticks can be picked up and put down in any order, modelled by actions on stick 1 and stick 2 being placed independently in parallel. But, it is only after both chopsticks have been picked up that a successful termination can occur and we can evolve to putting down the chopsticks. This behaviour is depicted in Figure 2.25.

As suggested by this example, the main role of the sequential composition operator is in enabling specifiers to subdivide their specifications into phases. Here we have decomposed into a picking-up phase and a putting-down phase and there will be a synchronisation (the successful termination) before moving between phases. We could specify this example using just action prefix and
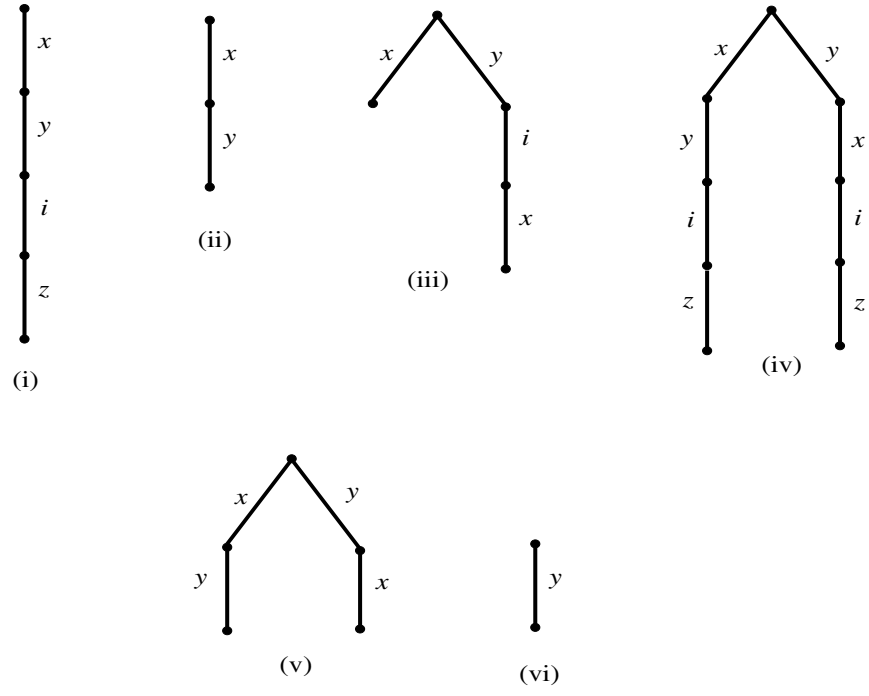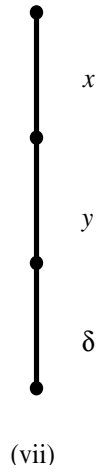
Fig. 2.23. Sequential Composition and Exit

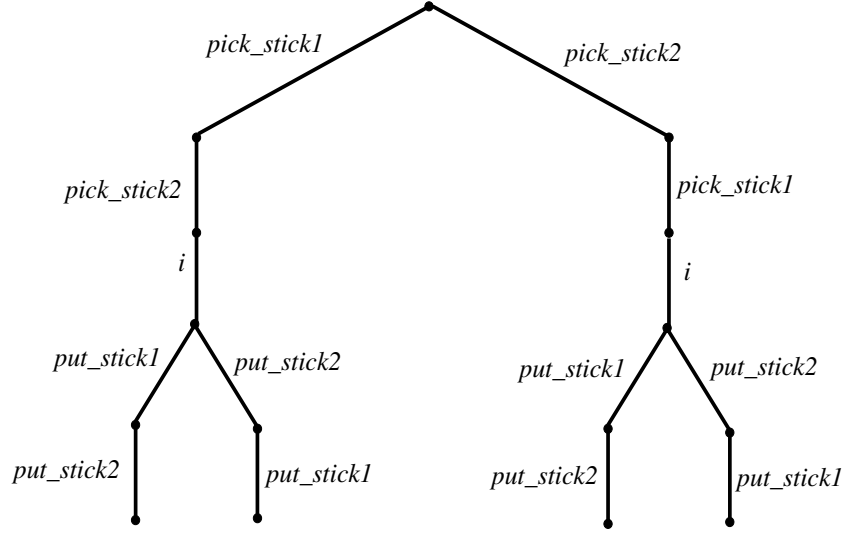**Fig. 2.24.** Further Sequential Composition and Exit Illustration

Fig. 2.25. Sequential Composition in the Dining Philosophers

choice (and possibly concurrency), but the specification would be far more complex and difficult to understand without the high-level description provided by $>>$. For example, the following is a specification of this behaviour that avoids the use of $>>$,

$$(\mathit{pick\_stick}1 \,;\, \mathit{pick\_stick}2 \,;\, i \,;\, (\mathit{put\_stick}1 \,;\, \mathit{stop} \,|||\, \mathit{put\_stick}2 \,;\, \mathit{stop}))$$
$$[] \, (\mathit{pick\_stick}2 \,;\, \mathit{pick\_stick}1 \,;\, i \,;\, (\mathit{put\_stick}1 \,;\, \mathit{stop} \,|||\, \mathit{put\_stick}2 \,;\, \mathit{stop}))$$

and more complex specifications can be given.

2.3.8 Syntax of pbLOTOS

This section brings together the constructs that we have introduced to give an abstract syntax for pbLOTOS. The syntax defines an arbitrary pbLOTOS specification $S \in pbLOTOS$ as follows.

$$S \; ::= \; B \; \mid \; B \; where \; D$$
$$D \; ::= \; (\, P \, [x_1, \ldots, x_n] := B \,) \; \mid \; (\, P \, [x_1, \ldots, x_n] := B \,) \, D$$
$$B \; ::= \; stop \; \mid \; exit \; \mid \; a \,;\, B \; \mid \; B_1 \; [] \; B_2 \; \mid \; B_1 \; |[x_1, \ldots, x_n]| \; B_2 \; \mid$$
$$\qquad B_1 \; >> \; B_2 \; \mid \; hide \; x_1, \ldots, x_n \; in \; B \; \mid \; B \, [y_1/x_1, \ldots, y_n/x_n] \; \mid$$
$$\qquad P \, [x_1, \ldots, x_n]$$

$a \in Act \cup \{i\}$, $x_i, y_i \in Act$, $D \in DefList$ (the set of pbLOTOS definition lists), $P \in PIdent$ (the set of process identifiers) and $B \in Beh$. The set of pbLOTOS

definitions is denoted *Defs*. Also note that the body of a definition (denoted B above) could contain a reference to the process identifier (P), thus setting up a recursive behaviour.

So, the top-level structure of a *pbLOTOS* specification (the first clause) is either a behaviour or a behaviour and an associated list of process definitions. Definitions have the expected form. There can be many such definitions. Behaviours are the main syntactic construct; they can be constructed using any of the operators and constructs that we have introduced.

In many circumstances we simplify the expression of action sets and mappings in parallel composition, hiding and relabelling operators, as follows,

$$B_1 \ |[G]| \ B_2 \ \triangleq \ B_1 \ |[x_1, \ldots, x_n]| \ B_2$$

$$\text{where, } G = \{\, x_1, \ldots, x_n \,\}$$

$$\textit{hide } G \textit{ in } B \ \triangleq \ \textit{hide } x_1, \ldots, x_n \textit{ in } B$$

$$\text{where, } G = \{\, x_1, \ldots, x_n \,\}$$

$$B\,[H] \ \triangleq \ B\,[y_1/x_1, \ldots, y_n/x_n]$$

$$\text{where, } \ H \ : \ Act \cup \{i, \delta\} \ \longrightarrow \ Act \cup \{i, \delta\}$$

$$\text{and} \qquad H(a) \ \triangleq \ \textit{if } a = x_i \ (1 \le i \le n) \textit{ then } y_i \textit{ else } a$$

We typically simplify presentation, by assuming the following operator precedences,

action prefix > choice > parallel composition > enabling > hiding > relabelling

where $A > B$ states that A binds more tightly than B. So, for example, the expression:

hide y in y ; *x* ; *stop* [] *z* ; *exit* ||| *x* ; *exit* >> *z* ; *stop*

would be fully parenthesised as:

hide y in ((((*y* ; (*x* ; *stop*)) [] (*z* ; *exit*)) ||| (*x* ; *exit*)) >> (*z* ; *stop*))

It should also be pointed out that the different operators of pbLOTOS can be subdivided according to their character. For example, we can view the operators, *stop*, action prefix, choice and process instantiation as "low-level" (more primitive) operators, whereas the operators, such as parallel composition, enabling, hiding and relabelling, can be viewed as more "high-level" operators. This is in the sense that the high-level operators facilitate high-level specification structuring. Such high-level structuring may, for example, reflect real-world identifiable components. Identification of such components in specifications using just low-level operators is often less straightforward.

In fact, a behaviour expressed using high-level operators can typically be mapped to an equivalent behaviour expressed purely in terms of the low-level

operators. A very important example of such a mapping is the *expansion* law for a calculus [101, 148], which relates parallel composition to action prefix and choice. In effect, the expansion law realises the interleaved interpretation of parallelism. As a reflection of this, parallel composition cannot so naturally be viewed as a high-level operator when true concurrency semantics are being considered, which they are in Chapter 4.

The term *monolithic* specification is often associated with a specification expressed purely in terms of the low-level operators. Thus, there is a clear distinction between specifications expressed in a constraint-oriented style, as highlighted in Section 2.3.6.4, and those expressed in a monolithic style.

2.4 Example

The following is a specification in pbLOTOS of the behaviour of the communication protocol with reliable acknowledgement. The top-level behaviour of the protocol is specified as a process called *Protocol*.

$Protocol\,[start, get, put] :=$

$\quad start\,;$
$\quad\quad hide\ send,\ receive,\ sendAck,\ receiveAck\ in$
$\quad\quad\quad (\,(\,Sender\,[get, send, receiveAck] \,|||\, Receiver\,[put, receive, sendAck]\,)$
$\quad\quad\quad\quad\quad |[send, receive, sendAck, receiveAck]|$
$\quad\quad\quad\quad DupMedium\,[send, receive, sendAck, receiveAck]\,)$

So, the action *start* initiates the behaviour of the protocol. This causes the processes *Sender*, *Receiver* and *DupMedium* to be invoked according to the required parallel composition. Notice, all the actions *send*, *receive*, *sendAck* and *receiveAck* are hidden from outside the protocol. Such hiding reflects the fact that the actions involved in implementing the protocol are hidden from users of the protocol. The behaviour of the sender could be specified as follows (this is the behaviour we discussed earlier).

$Sender\,[get, send, receiveAck] :=$
$\quad get\,;\ send\,;\ Sending\,[get, send, receiveAck]$

$Sending\,[get, send, receiveAck] :=$
$\quad hide\ timeout\ in\ (\,receiveAck\,;\ Sender\,[get, send, receiveAck]$
$\quad\quad\quad\quad\quad\quad\quad [\,]\ timeout\,;\ send\,;\ Sending\,[get, send, receiveAck]\,)$

The behaviour of the receiver could be specified as follows.

$Receiver\,[put, receive, sendAck] :=$
$\quad receive\,;\ put\,;\ sendAck\,;\ Receiver\,[put, receive, sendAck]$

The top-level behaviour of the medium could be specified as

$DupMedium\,[send, receive, sendAck, receiveAck] :=$
    $Medium\,[send, receive] \;|||\; AckMedium\,[sendAck, receiveAck]$

which in turn uses the following sending medium,

$Medium[send, receive] :=$
    $send\,;$
      $(\,i\,;\; Medium\,[send, receive]$
        $[]\;\; receive\,;\; Medium\,[send, receive]\,)$

and the following acknowledgement medium,

$AckMedium\,[sendAck, receiveAck] :=$
    $sendAck\,;\; receiveAck\,;\; AckMedium\,[sendAck, receiveAck]$
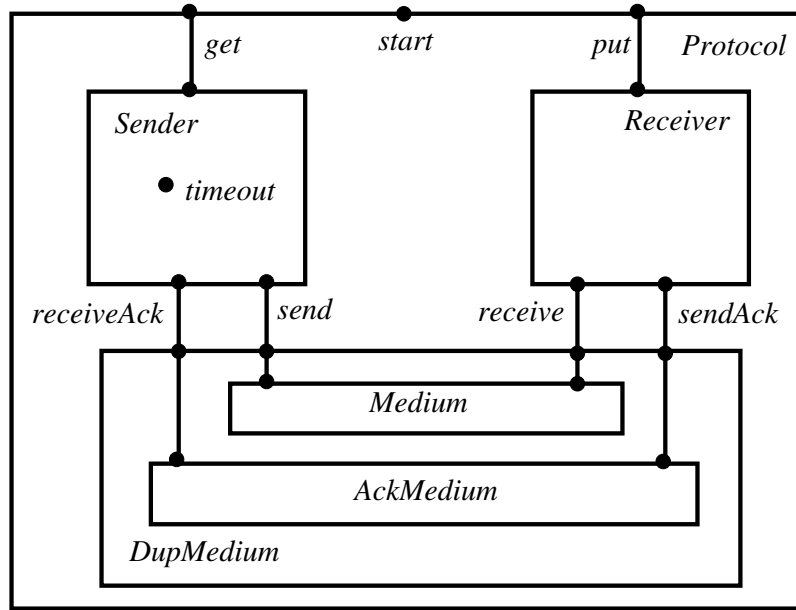
Fig. 2.26. Box Diagram of the Communication Protocol

We can explicitly depict the structure of the protocol specification using a box diagram, such as in Figure 2.26. This diagram shows the process structure of the specification; you should notice that the structure closely resembles our original depiction of the protocol in Figure 2.1. The process *Sending* is

not included in this diagram as it sets up a mutual recursion with *Sender*, which is difficult to depict. Such diagrams as these are only really useful for representing static process structure, the dynamics of process instantiation quickly become unrepresentable.

The interface of each process is directly represented. For example, the process protocol has three external gates, *start*, *get* and *put*, which are distinguished from internal gates by being linked to the exterior of the *Protocol* box. In addition, the fact that subprocesses interact on external gates is indicated by a line segment, e.g. the attachment of *get* to *Sender*. In contrast, *start* is not interacted with by any subprocess. All actions in the specification are depicted (apart from the *i* action in *Medium*). Notice also that the diagram shows that *timeout* is completely internal to the process *Sender* and that the gates *receiveAck*, *send*, *receive* and *sendAck* are hidden from the interface of the process *Protocol*.

Finally, the top-level behaviour of the protocol specification will invoke the process *Protocol* and thus initiate its evolution, e.g.

$$Protocol\,[sstart, ssend, rreceive]$$

where the external gates of the protocol have been renamed, *start* to *sstart*, *get* to *ssend* and *put* to *rreceive*.