BEHAVIOURAL SUBTYPING IN PROCESS ALGEBRA

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Charles Peter Briscoe-Smith
29 November 2000

Abstract

Subtyping relations embody a notion of substitutability, and are an important tool in formal methods. The downward simulation relation is well-known and widely used as a subtyping and refinement relation for state-based approaches, but there is no single relation which is widely accepted to be *the* subtyping relation for the behavioural setting; however, there are several candidate relations.

Developments such as the multi-viewpoint specification method of RM-ODP encourage the use of several different formal methods in a single project. Thus, it becomes important to obtain implementations of a single notion of subtyping in several different paradigms. In this thesis, we attempt to find a process algebraic relation which corresponds to the state-based subtyping relation, downward simulation.

While trying to achieve this, we define a translation between a state-based notation and a process algebra, and we uncover some of the similarities and differences between these two specification paradigms. In particular, we investigate the meaning of undefined behaviour in each setting. When a state-based specification does not define an operation, the intent is that the system's behaviour is unspecified (not well-defined) if that operation is invoked. In a process algebra with refusals semantics, however, a process is implicitly specified to refuse any action it is not specified to accept, and such a refusal constitutes well-defined behaviour. As part of our translation, we devise a method of representing the unspecified behaviour of the state-based world in a process definition.

Finally, we use our translation to prove that the presence of subtyping between a pair of state-based specifications implies reduction between their LOTOS translations, but that the presence of reduction does not imply subtyping. We conclude that reduction itself does not correspond to state-based subtyping, but that any relation which does must be based on a stronger semantics than reduction, such as bisimulation semantics.

Acknowledgements

I would like to express my thanks to John Derrick and Howard Bowman, my PhD supervisors, and to Tim Regan and Ben Strulo of BT research labs. The initial idea for this work came from a discussion between the five of us, and they have all given me plenty of help and advice along the way.

Funding for this work was provided by British Telecommunications research laboratories, Martlesham Heath, Ipswich.

Contents

List of Figures

Chapter 1

Introduction and motivation

Over the course of the past few decades, information technology has become an integral part of the industrialised world. Commodities, currencies, and company shares, the lubricants of any capitalist economy, are traded on the world's stock markets mainly by means of electronic transactions. Food distribution, a function now performed largely by the large supermarket chains, is organised and managed by means of computerised stock control and delivery systems. Computers are used in the control and management of such vital infrastructure as the supply of power, water and natural gas. Indeed, daily life is dominated by information technology.

In many cases, it is difficult or impossible to have a single computer run the entire system. In the case of, for example, a bank's network of cash dispensing machines, or the stock management system of a large supermarket chain, the systems are *distributed*—the complete system comprises a number of geographically separated nodes, communicating about the tasks they are performing.

Communications must take place over some medium—typically used media are wired networks, such as leased lines or the public telephone network; optical fibre, such as those provided by some cable operators; radio, such as cellular telephone networks or satellite; and physical media such as the traditional "box full of magnetic tapes" used for transmitting backups to an off-site location. All these methods of communication are prone to failure, however, and a system architect must take into account that communications may fail at any time. This introduces a new class of error cases which the system must be designed to withstand, and this consequently imposes an additional burden on the system architect.

Despite the amount of research which has been put into the problem of producing correct systems and the number of known and proven protocols and techniques, information technology does not work faultlessly, and faulty systems are still sometimes deployed. In some cases, already-deployed systems are found to be faulty. For example, many older computing systems were expected to (and in some cases did) exhibit problematic behaviour at or around the start of the year 2000, due to a systematic design flaw which was prevalent, even entrenched, when these systems were being built. In other cases, the deployment of a system is delayed when unforeseen faults are found. For example, the UK passport office had a large backlog of unprocessed applications in June and July 1999; "teething trouble" in the passport office's new computer system was blamed for the delays in issuing passports. In [Bro75], Brooks details some of the management-related reasons why computer systems are often delivered late or not working correctly.

Formal, mathematically based design methods are believed to be able to reduce the number of faults in finished software systems. They can do this in ways ranging from providing the system architect with insight into the design problem, to (potentially) enabling a rigorous, formal proof that the system meets its specification.

In practice, it is not possible to formally prove a real system correct, not least because the ultimate definition of "correct" operation must necessarily be informal. Formal methods can, however, be used as an aid in many stages of the system development process; they are another tool for the system's specifiers, designers and implementors.

Typically, formal methods are used to capture requirements of the system, concisely and unambiguously, in the form of a formal specification; to refine the specification by changing it in accordance with a refinement relation; and to use the specification to generate an implementation, or to guide the implementation process.

Formal methods for computer system design fall into broad categories, depending on the paradigm upon which they are based. In this dissertation, we will focus on two of the most-used paradigms, state-based specification and behavioural (particularly process algebraic) specification.

In the following sections, we will first examine the ISO (International Standards Organisation) reference model for distributed systems, and then introduce the two groups of formal methods, state-based and process algebraic specification, which we will examine further. We will then introduce the notion of *subtyping* on which we will be focusing throughout the dissertation.

1.1 Distributed systems

There are many desirable properties which distributed systems can support. Users can access remote resources via the system from many physical locations, processes can migrate from machine to machine to balance workloads, and sections of applications can be added to and removed from the system while it is running, allowing upgrades and alterations to be effected without impacting the system's users.

"Open distributed processing" (ODP) is a term used to describe those distributed systems based on open standards and protocols. ISO has standardised a reference model for ODP, named "RM-ODP"[ISO93, Lin91].

RM-ODP defines a series of five *viewpoints* from which a system can be specified. Each viewpoint is concerned with different aspects of the system, and the entire system can be described more easily by producing specifications dealing with the aspects visible from each of these viewpoints. For example, the computational viewpoint deals with the operation of the processing nodes making up the system, while the information viewpoint can be used to specify the information being processed and how it moves between system components.

Because the viewpoints all focus on different aspects of the system's operation, the system's specification from each of those viewpoints will be different. There will certainly be some overlap, because the same system is being specified in each case, but not all aspects of the system will be represented in all of the viewpoint specifications.

In addition, some viewpoints are more naturally specified using different notations than other viewpoints. There is no single formal method which can be shown to be the "best" and suitable for specifying all systems[ST95]. Even in a single project is seems likely that different notations might be used for different viewpoints. For example, a state-based notation such as Z is often used to specify the ODP information viewpoint, while a behavioural notation such as LOTOS might be used, in the same project, to specify the same system from the computational viewpoint. Unfortunately, the greater convenience of multiple viewpoint specifications comes with a price: it is possible for the specifications in a project to be inconsistent—that is, it might not be possible to build a real system which implements all of the project's specifications at once. To check that there are no inconsistencies between the specifications, it is useful to be able to translate specifications between notations, and between specification paradigms[BDLS96, BD99].

Heterogeneous groups of processing nodes can be used in RM-ODP, and the structure used allows components to be added, removed and changed dynamically. RM-ODP uses a distinguished object called a *trader* to enable the pieces of a distributed application to locate one another. Without a trader, many objects would likely be written to contain a hard-coded location for each service needed, making it difficult to replace services or move them between nodes. By requiring all the objects in the system to look up the services they need via the trader, services can be moved at will, the only changes required being in the trader's database.

The trader works as follows. When an object providing a service is created, it performs an *export* operation on the trader, passing it a description of the service it offers. The trader maintains a database of currently available service offers. When a client wishes to find a service, it performs an *import* operation on the trader, specifying what kind of service it wants, and the trader responds with copies of a number of suitable service offers. The client can then choose one of the service offers, and start communicating with the service.

Examples of the kinds of services we might trade for include hardware-based resources such as input and output devices, which, by their nature, must be sited in a single place; high-performance computing servers or hardware accelerators for specific tasks; information repositories, of which it would be inefficient to keep copies on every machine; secure processes or data, such as authentication servers for checking passwords; and objects containing knowledge which should be kept encapsulated, such as a network router's knowledge of local network topography.

A trader with an appropriate database can also be useful to users directly, to find, say, relevant modules in a software library, or resources on the Internet[Goo95].

Clearly, the trader must have some method of deciding whether a particular service offer fulfils the requirements of a given import operation. Since the service must be usable by the object performing the import operation, the trader should use subtyping to make this decision. (As we will see later in this chapter, subtyping relates each type to those other types for which it can be substituted.)

In the trading context, types are service types, descriptions of services offered and requested. The client and the trader must both understand a common service type description language in which service providers will register their services with the trader, and in which the client will describe the service it wants.

Such a service description language needs to be able to describe the structure of the interface between the client and the server, consisting of the operations the client can invoke on the server. It should also convey some idea of the meaning of those operations, to prevent two offers providing similar interfaces but different semantics from being related by the type system when they actually do quite different jobs. Other aspects of the service can be represented, too; these are usually called *service attributes*, and are typically such things as quality of service, cost, security or administrative affiliation.

When a client sends a description of the service it wants to the trader, the trader must somehow match this to the offers it has in its database. If it cannot find an offer matching exactly the requested service, it should look for offers of similar services, providing all the facilities that the client wanted, but possibly having other features that the client will not use. It is looking for a service which has all of the behaviour the client asked for, and which the client could use without knowing that it was any different from the service type it requested. The relationship between the service requested and the service returned by the trader should be *subtyping*.

In addition to the translations between specification notations which we mentioned above, it would also be useful to have specification refinement relations, particularly subtyping relations, which could be considered in some sense "equivalent" in each of the paradigms.

We will discuss subtyping in more detail later in this chapter. Before that, we will first introduce two common specification paradigms in more detail: the state-based method and process algebra.

1.2 The state-based specification paradigm

One of the important formal specification models is that of *state-based specification*. State-based specification languages such as Z[Spi92] and VDM[Jon86] may owe their popularity in part to the similarity of their central concept—that state is specified explicitly, and operations are then defined to manipulate that state—to traditional imperative and object-oriented implementation languages.

State-based specification notations concentrate on describing the state present within an object or system, and on specifying how that state may change when operations are invoked. A simple Z specification might specify its state with a schema (a boxed structure specifying the name and type of some variables and an optional logical predicate) as follows:

$$\begin{array}{|l}\hline \textit{NumberSet} \\ S : \mathbb{P}\,\mathbb{N} \\ \hline \end{array}$$

This schema specifies the state of an object. The state is named *NumberSet*, and consists of a set, named S, of natural numbers, as drawn from the set $\mathbb{N}$. The meaning of this schema is that, at any time while the object whose state is *NumberSet* exists in the system, its state can be taken to represent a set of natural numbers. Over time, as the system operates, the state may change.

We can add a second part to this state specification, placing extra conditions on the state:

$$\begin{array}{|l}\hline \textit{NumberSet} \\ S : \mathbb{P}\,\mathbb{N} \\ \hline |\,S\,| < 100 \\ \hline \end{array}$$

In this case, we have specified that S always contains fewer than 100 elements. This condition, the *invariant*, applies to the state whenever the state is not currently being changed by an operation. Whenever an operation begins, the state satisfies the invariant; when the operation completes, having potentially changed the state, the new state must also satisfy the invariant. (We do not consider the state during the progress of an operation, since we model each operation invocation as a single, atomic unit.)

The exact interpretation of the invariant may vary depending on the situation. In some cases, it can be seen as an additional condition of the predicate of each operation (we will explain operation predicates below); alternatively, it can be seen as a condition against which the operation predicates can be checked—if, ignoring the invariant, an operation can be invoked in an invariant-satisfying state and then transform it into a state which does not satisfy the invariant, the operation's predicate is considered to be in error.

In Z, operations are specified as in this example:

$$\begin{array}{|l}\hline \textit{AddNumber} \\ \Delta \textit{NumberSet} \\ n? : \mathbb{N} \\ \hline S' = S \cup \{n?\} \\ \hline \end{array}$$

Here, we are specifying an operation named *AddNumber*, which updates the set S to include the number, "$n?$", passed to the operation.

The first part of the schema specifies the data on which the operation operates. The Δ before the reference to *NumberSet* conventionally indicates that this operation may change the state given in *NumberSet*. The state present before the operation commences, the *before-state*, has the name given in the specification of *NumberSet*, while the state which results from applying the operation, the *after-state*, has a prime added to its name; in this case, S is the set comprising the before-state, while the corresponding set in the after-state is named S'.

Also mentioned in the first part of the schema is "$n?$". The question mark conventionally indicates that this item of data is an argument or input, which is specified by the environment and passed to the operation when it commences. Results and outputs are specified similarly, but are named using an exclamation mark instead; for example, "$r!$".

The second part of the schema is the *predicate*. This specifies what constitutes the "correct" behaviour of the operation. In this case, the predicate specifies that S' (in the after-state) is the union of S (in the before-state) with a set containing just the operation's argument, "$n?$".

In general, there are four items that an operation's predicate links together: the object's before-state, its after-state, the operation's arguments, and its result values.

The final piece necessary to complete a specification is the initial state of the object. In Z, this is given in a style similar to that used for an operation, but using an operation name ending with *Init*, as in this example:

$$\begin{array}{|l}
\hline
\textit{NumberSetInit} \\
\textit{NumberSet} \\
\hline
S = \varnothing \\
\hline
\end{array}$$

(Note that we do not use Δ here: we are not specifying an operation but a condition to be imposed on a state.) This schema indicates that in its initial state, the object we are specifying has its value S equal to the empty set. In some cases, more than one state of the system may satisfy the predicate given in the *Init* schema; if this is the case, the system may validly start in any such state.

In the above description of Z, we used the words "object" and "system" to denote the structure being specified. In Z, this usage is acceptable, because a single specification is only capable of capturing one "object". That is, Z does not provide for a single specification to be structured into smaller, independent parts in the way that is familiar from object oriented programming languages. Object-Z[Ros92, CDD^{+}90] is an extension of the Z notation which allows schemas to be grouped into "classes", and allows a system made up of multiple objects to be specified.

Z is a complex notation, containing many features necessary for a practical state-based specification notation. Because of the complexity of Z, we will not be using it in the main body of this thesis. Instead, we will use a much simpler state-based specification notation which we will introduce in chapter 3.

1.3 The process algebraic specification paradigm

Process algebraic specification is, in a sense, the opposite way around from state-based specification. Whereas a state-based specification describes an object's state, and operations are described in terms of the states in which they may occur and the changes they make to the state, an object described using a process algebra has no explicitly-described state at all. Instead, process algebras describe only the behaviour of objects; in particular, the allowable orderings of interactions which the object may take part in. Thus, process algebras specify the system from the point of view of how objects interact. For this reason, process algebraic specification is considered a form of behavioural specification.

The forms of behavioural specification which we will examine are all given in terms of *actions*: named instantaneous interactions of a process with its environment. Actions occur instantaneously, and no two actions can occur at the same instant. Generally, actions represent *observed* interactions—events which have taken place—rather than "attempts" or "plans" to interact. That is to say, an action does not occur unless the event represented by the action actually occurs: in the usual case where actions are used to represent interactions between two or more parties, the action does not occur unless all the parties actually interact. However, definite internal changes of state can be represented as *internal actions*, even if they cannot be observed. The notations we will use are not concerned with time or duration—a period of one millisecond elapsing between two actions is represented no differently than a period of one week.

One of the simplest forms of behavioural specification is the *labelled transition system*. A labelled transition system is a directed graph, with one distinguished node—the starting node—and with a label associated with each arc. An example of a labelled transition system is shown in figure 1. In diagrams of labelled transition systems, we indicate the starting node with a short arrow from outside the body of the graph.

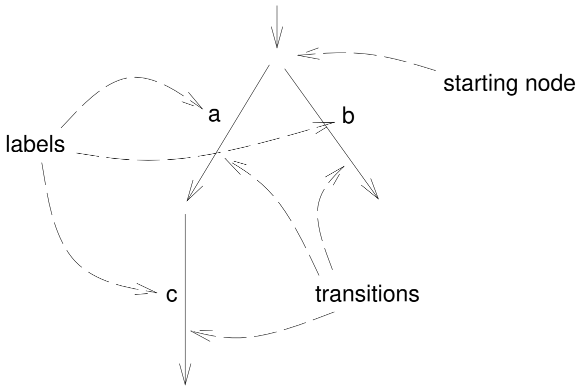

Figure 1: Example of a labelled transition system (the dotted arrows are not part of the labelled transition system.)

This labelled transition system represents a process which can engage in one of two external interactions, named a and b. It need not engage in either of these interactions, and it cannot engage in both. If the process engages in the interaction labelled a, we say it has performed the action a, and its state changes as indicated by the graph. After performing the action a, the process's state is such that it is then able to perform the action c; again, this is not required.

Alternatively, the process can begin by performing the action b. Because actions are considered atomic, instantaneous, and distinct in time, there is no possibility that the process could perform more than one action at once; it must choose either a or b. After performing the action b, the process reaches a node in the graph from which no arc departs. As a result, it cannot perform any more actions.

The semantics of labelled transition systems are simple, and they are therefore used as the target of the behavioural semantics for some other behavioural specification notations. In particular, LOTOS's semantics are defined in terms of labelled transition systems, as we will see when we introduce the language in chapter 3.

Labelled transition systems are very "low level", and are not easy to specify systems with. We will now briefly introduce CSP[Hoa85], a process algebra. CSP specifications describe the behaviour of processes, as labelled transition systems do. For example, the following CSP specification describes a process which behaves similarly to the process given by the labelled transition system in figure 1:

$$(a \to c \to STOP \mid b \to STOP)$$

CSP processes can be combined to give more complex behaviour in a more manageable way than is possible with labelled transition systems. For example, processes can be placed in parallel using *interleaving* behaviour. The resulting behaviour is formed from the behaviours of the two processes being interleaved, in such a way as to represent that the two are occurring independently, alongside one another. The actions of each of the two processes must be ordered according to its own specification, but the actions of the two processes may otherwise be intermixed in any order.

For example, the following CSP specification is the result of interleaving two processes, $a \to STOP$, which can only perform the action a, and $b \to STOP$, which can perform the single action b.

$$a \to STOP \mid\mid\mid b \to STOP$$

The behaviour of this process is as follows. It can perform the action a first, then b, or it can

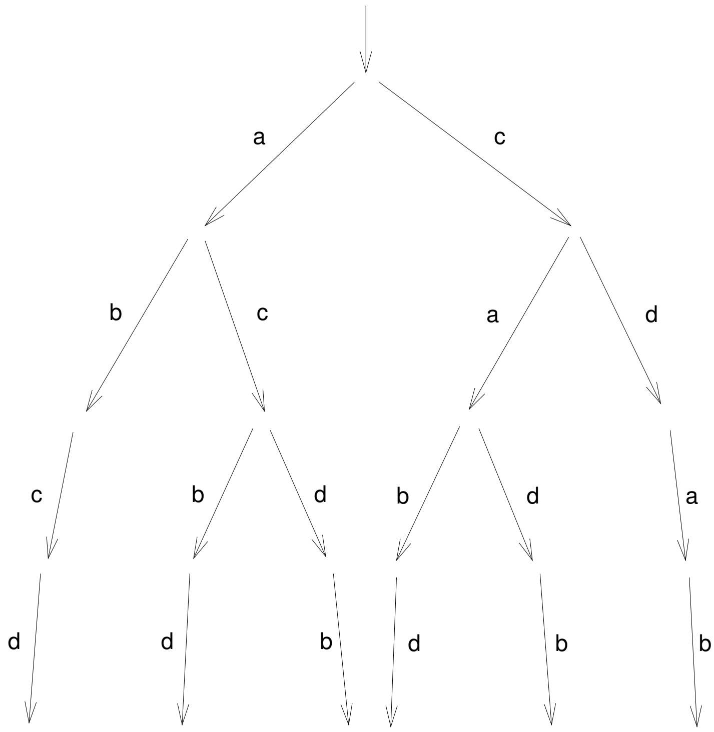

Figure 2: Result of interleaving $a \rightarrow b \rightarrow STOP$ with $c \rightarrow d \rightarrow STOP$

perform b first, followed by a. (As with all processes, it need not perform all of these actions; because CSP is not concerned with time, any process might, at any time, stop and perform no more actions. This can happen if, for example, the process needs the co-operation of another process to perform some action, but that process is not able to take part in the interaction.)

More complex interleavings have more complex results. The following CSP specification's behaviour is illustrated by the labelled transition system in figure 2.

$$a \rightarrow b \rightarrow STOP \mid\mid\mid c \rightarrow d \rightarrow STOP$$

In this thesis we will be using the process algebra LOTOS[BB87, ISO89], which is used in a similar manner to CSP. The subset of LOTOS which we will use is described in detail in chapter 3.

1.4 Subtyping

Subtyping is an important concept in each of the areas discussed above: in the trading function of an RM-ODP distributed system, and in both state-based and process algebraic formal specification.

In distributed systems built using an RM-ODP trader, the trader's clients request services by specifying the type of service they require. The trader then finds a service which it can substitute for the one the client requested without the client knowing that the service it receives is any different from that which it requested. More accurately, the service supplied must meet the specification supplied by the client. In most cases, this means that it must support certain operations and must exhibit certain behaviour when those operations are invoked.

When developing and refining formal specifications, in either the state-based or the process algebraic setting, it is often important to verify that the process described by a newly developed specification preserves given properties of an existing specification. It is useful to be able to check that the new process definition can be substituted for the existing process in a system without invalidating any of the assumptions the remaining parts of the system have made about that process. This can be done in two steps. Firstly, limit the assumptions which may be made about the object to those properties of the object deducible from its specification—in practice this means ignoring the object's implementation and only using it according to its specification. Secondly, ensure that any object which is an implementation of the new specification must necessarily also be an implementation of the original specification.

We can view each of these two situations as requiring that objects meet specifications: in the case of the ODP trader, the client passed a service specification to the trader. The trader must then return an object which meets that specification. In the case of developing specifications, we must ensure that any object which meets one specification must necessarily meet a second specification, too.

Alternatively, we can view this as substitutability: the object returned by the trader must be able to substitute effectively for the object the client was expecting. Similarly, in a situation where an object meeting the second specification is expected, it must be possible to substitute an object meeting the first specification and be sure that it will meet the second specification too.

These situations call for a relation ensuring substitutability. Such relations are normally called subtyping relations. We can define subtyping informally as follows: if a type A is a subtype of a type B, this means that any object which is of type A is necessarily also of type B. Since types are effectively specifications of how an object must behave, subtyping gives the relation we need for ODP trading and for specification development.

The set of properties required to be preserved when substituting one object for another in a given situation depends on the exact interpretation placed upon the specification notations being used. Thus, there are different subtyping relations corresponding to these different interpretations. Because of the view that each of these relations, different though they are, defines a particular notion of substitutability, we can view each of them as a subtyping relation.

The concept of subtyping is familiar from object oriented programming languages[FM96, FM94]. There has been much work on type systems for object oriented languages, for instance [BCC+95]. We can make use of some of this work when studying subtyping relations for use in ODP trading and for specification refinement. However, object oriented subtyping has a more limited scope than the subtyping we are interested in.

Specifically, the subtyping relations used in object oriented programming languages are generally designed only to ensure type-safety. They do not make any attempt to constrain behaviour; they merely ensure that any possible method call in the program must be to a well-defined method and that the arguments and return values of the call are of the correct number and are themselves properly typed. This constitutes the relation we will refer to as *signature-based subtyping*, because it constrains only the signature of the methods, not their behaviour.

Most languages implementing a subtyping relation will include signature-based subtyping as part of their subtyping relation to ensure type-safety. If they do not (as, for example, Eiffel[Mey92] does not) type-safety cannot necessarily be guaranteed without a more detailed analysis of the specific program being compiled. Although we will not be concerned with signature-based subtyping in the remainder of this thesis (apart from revisiting it briefly in chapter 2), it is a weak relation which is useful in practice for providing type-safe subtyping in programming and specification languages. We will generally assume that signature-based subtyping is to be required in addition to the subtyping relations we will propose.

1.5 Problem description

The above description of subtyping gives a brief outline of signature-based subtyping. In this thesis, we will be concerned with the more tricky problem of *behavioural subtyping*—the problem of determining whether, if type A is substituted for type B in a system, the new system will behave in "the same manner" as the original system.

Signature-based subtyping is not, on its own, sufficient for the trading and specification scenarios outlined above. As a light-hearted example of what could happen when only signature-based subtyping is used, consider the situation where an artist and a cowboy are both asked to perform an operation named *draw*. The results in each case will be markedly different: the cowboy's *draw* operation involves drawing his gun, while the artist's *draw* operation produces a painting.

The message here is that an operation's signature on its own does not give sufficient information for us to be able to tell whether it is substitutable for another operation with a given signature. We need to know something about the behaviour of the operation before we can make a decision.

This is clearly a stronger notion of subtyping than the signature-based approach. Signature-based subtyping is strong enough to ensure that substituting a subtype for its supertype will not cause the resulting system to fail with a type error, and this is clearly a prerequisite for behavioural subtyping.

Our objective is to obtain a behavioural subtyping relation on processes described using a process algebra. A signature-based relation seems to be inadequate for this setting, so we will examine the possibility of a type system based on the behaviour of objects. By "behaviour", we refer to specification of the allowable orderings of operations on the object. We will use the LOTOS[ISO89, BB87, Tur93] specification language to express objects' behaviours, although CSP, CCS or any other process algebra could easily be substituted.

If we consider the type of an object to be its corresponding LOTOS behaviour, we would like to represent subtyping by one of the standard LOTOS relations such as (bi)simulation, extension, reduction or trace preorder. Such relations have been extensively studied, and their properties are well known.

In the world of state-based specification, the downward simulation relation and its instantiations such as the Z refinement relation are generally understood to represent subtyping. In this thesis, we will refer to the "state-based subtyping relation", which we take to be the Z refinement relation. This relation also forms the basis of the behavioural subtyping relation proposed by Liskov and Wing[LW94], as we shall see in chapter 2. However, there is no such general agreement on the identity of a behavioural subtyping relation, although it has been suggested that the LOTOS extension relation is behavioural subtyping[Rud92].

Ideally, we would like to identify a LOTOS relation which is equivalent to state-based subtyping, but which relates LOTOS processes instead of state-based specifications. Since it is not possible to directly compare state-based specifications with processes, "equivalence" in this case is meaningless. We will have to settle for something less than equivalence.

Our approach will be to try to find a LOTOS relation which "corresponds" to state-based subtyping. The conditions under which we consider a LOTOS relation to correspond to state-based subtyping are as follows. We will first define a translation which maps state-based specifications to LOTOS specifications. Then, we will attempt to find a LOTOS relation which relates two translated specifications exactly when the state-based specifications from which they were translated are related by the state-based subtyping relation. In other words, if the state-based specification A is a subtype of a second state-based specification B, then the LOTOS translation of A should be related to the LOTOS translation of B by the relation representing behavioural subtyping. Similarly, when state-based subtyping does not hold, the LOTOS subtyping relation should also not hold. If this is the case, then the LOTOS subtyping relation corresponds to state-based subtyping. This idea is illustrated in figure 3.

The ability to translate specifications from state-based notations into LOTOS may also be useful in its own right. For example, the ability to use the tools available for manipulating and checking LOTOS processes could then be applied to state-based specifications, too.

When specifying a single system from multiple viewpoints in RM-ODP, it is useful to be able to write the viewpoint specifications in different notations, and even in different paradigms. The ability to translate all the specifications for a system into a single language enables consistency

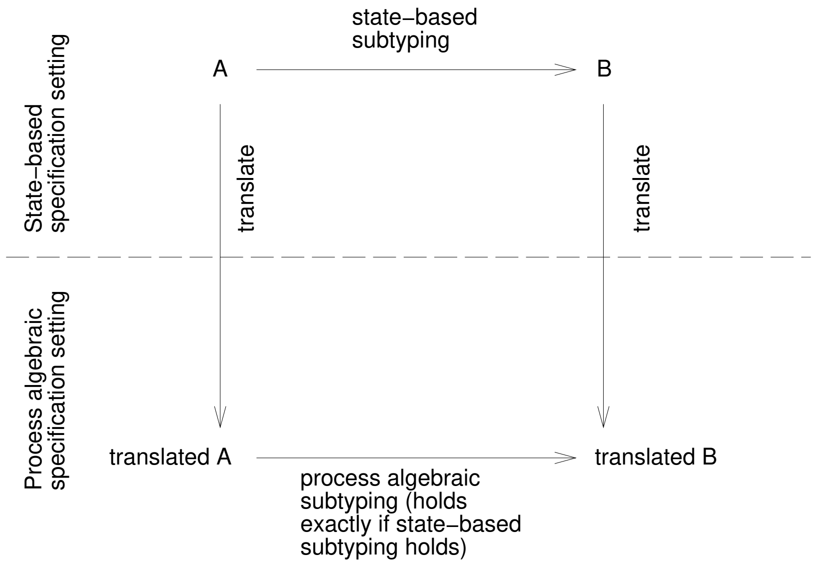

Figure 3: Correspondence between relations over different domains

checks to be carried out[BBDS96]. In addition, having a single notion of subtyping which can be instantiated separately in different paradigms would help the development process.

1.6 Thesis summary

Chapter 1 has provided some context and motivation for our study of formal methods for distributed systems, and, in particular, outlined why subtyping is an important concept in both RM-ODP and in the two varieties of formal methods we will be dealing with. We have seen that, while the state-based specification world has a well-established notion of subtyping, as captured by relations such as the Z refinement relation, there are a multitude of refinement relations on process algebraic specifications, many of which characterize various notions of subtyping.

We then outlined the task we will attempt in this thesis: to find a process algebraic relation corresponding to the Z refinement relation. We now summarize the remainder of this thesis.

Chapter 2 describes the characteristics of the state-based and process algebraic specification methods in more detail and introduce briefly some of the notations available. We describe the basic concepts of two well-known specification notations, Z, a state-based specification notation, and CSP, a process algebra, and demonstrate what labelled transition systems are. We then describe in detail the concept of subtyping. We discuss several subtyping relations which are commonly used or described in the literature, including downward simulation and Liscov and Wing's behavioural subtyping relation.

Chapter 3 presents the notations and relations we will be using throughout the thesis. In particular, we will present a simple state-based specification notation based loosely on the notation of Liskov and Wing[LW94]. This notation allows the specification of the state space and starting state of a type, and to give the predicate for each operation on that type.

We next give a subtyping relation for our state-based specification notation. This is, in fact, the Z refinement relation, a relation very similar to the downward simulation relation, and necessary for Liskov and Wing's subtyping relation to hold. (Formally, Liskov and Wing's relation implies the Z refinement relation.)

We now have the languages we will be translating both from and to. We discuss several basic similarities and differences between the state-based and process algebraic settings, and, in particular, compare the characteristics of data output in several languages in the literature.

Chapter 4 opens with a discussion of some possible ways of constructing a translation between languages, followed by a description of a very simple method of translating state-based specifications into labelled transition systems. This simple translation does not, in fact, work as we would like it to, as it is not discriminating enough to generate distinct translations from state-based specifications with distinct behaviour. Despite that, it is sufficient to help us evaluate a number of LOTOS development relations as possible subtyping relations, which we do next. We eliminate several relations due to undesirable properties, and conclude that the relation known as *reduction* is the only one of the relations we examined which is suitable for use as a subtyping relation.

We find, however, that despite the advantages that reduction has, and the similarities of its properties to those of state-based subtyping (which we take to be Z refinement), it does not correspond to subtyping. In discussing the reasons for this, we conclude that our translation is not adequate. Specifically, we note that the interpretation of certain behaviour, which we call "unspecified" behaviour, is radically different between the state-based and process algebraic settings. We discuss a method of representing the state-based interpretation of unspecified behaviour in labelled transition systems, and outline (informally at this stage) how our translation may be modified to include this method.

Chapter 5 begins by introducing a development of the very simple translation introduced at the beginning of chapter 4. This new translation remedies the previous translation's failure to distinguish between our example specifications. It does this by including a representation of data—specifically, the arguments and results of operations—in the process produced.

We then discuss a further problem with this new translation arising from the fact that it treats values returned from operations exactly the same way as arguments passed to operations. When an operation can nondeterministically return one of a number of possible values, this treatment leads to the process's environment being able to choose which of those values it receives. We discuss two possible ways to avoid this kind of unintentional determinism, involving modifications to the LOTOS language which our translation targets.

Chapter 6 brings together the simple translation first given in chapter 4, the improvement we made in chapter 5 to include argument and result values, the treatment of unspecified behaviour given in chapter 4, and a formulation of nondeterminism based on the ideas in the LOTOS variants given in chapter 5. We note some remaining problems with the translation and work through an example translation.

Chapter 7 presents a proof that, for every pair of state-based specifications, where the first is a subtype of the second, the translation of the first specification will be a reduction of the translation of the second. The proof is inductive over any abstraction relation which can be used to show that state-based subtyping holds between the untranslated specifications. We also present a counterexample showing that, if a translated specification is a reduction of a second translated specification, it is not necessarily the case that the first untranslated specification is a subtype of the second.

Chapter 8 concludes the dissertation with a summary of our results, the conclusions we draw from them, a summary of some of the limitations of our work, and some suggestions for further work on the subject.

Appendix A gives a brief introduction to a subset of the LOTOS language. Since LOTOS is standardised, we do not include this introduction in the body of the thesis; however, its presence ensures that we describe all the languages we use.

1.7 Summary

In this chapter, we outlined some of the problems encountered in building large scale distributed computer systems, and introduced some of the methods available to tackle these problems: the RM-ODP framework for the specification and design of distributed systems, and two kinds of formal specification method: state-based specification and process algebra.

We introduced the ODP trading function, describing how it allows distributed systems to operate even when components of the system change. We examined the notion of service types and saw that the trader must have some notion of subtyping in order to be able to decide whether a given service fulfils a given service request.

We discussed the notion of state-based specification, giving briefly some examples using the Z notation. We exemplified the process algebraic approach by showing some labelled transition systems and CSP process specifications.

We informally defined subtyping as substitutability: if any object a of type A can be substituted directly for any object b of type B and the resulting system is capable of operating using object a just as it was originally capable of operating using object b, then type A can be said to be a subtype of type B. The existing formal models of subtyping are all formalisations of this intuitive definition.

Finally, we presented the problem which this thesis attempts to address, and outlined the steps we will take to solve the problem in the following chapters.

Chapter 2

Specification techniques and subtyping relations

Our aim in this thesis is to identify a process algebraic relation which corresponds to the state based subtyping relation, which we take to be the Z refinement relation, or downward simulation. We plan to do this by producing a translation from a state based notation to a process algebra. The purpose of this chapter is to explain *why* we consider the downward simulation relation to be a good state based subtyping relation.

To do this, we first provide a more detailed outline of the state based and process algebraic specification methods. We then describe the idea behind subtyping, and discuss how it has been implemented in the state based and process algebraic settings.

2.1 Process algebra

Before we begin looking at process algebra, let us introduce the example we will use to describe it. Imagine a coffee vending machine. The machine has a coin slot for 5p coins and one for 10p coins. For 15p, it will dispense a cup of coffee, or for 10p it will deliver a cup of tea. As many have done before, we will specify the behaviour of this machine with process algebra.

Process algebras are one kind of behavioural specification technique. As we outlined in chapter 1, process algebras describe a behaviour in terms of communications[Mil89]. Changes of state are represented as "hidden" or "internal" communications, so any activity—communication or state change—in a system can be represented in a process algebra by a single notion, the *action*. An action is an atomic representation of the idea that "something happens". It is the primitive unit of *observable behaviour*.

In our coffee machine example, we can identify several interactions which can take place. A customer can insert a 5p coin, which we will identify with an action labelled $take5p$. Similarly, we can identify an action $take10p$ corresponding to the insertion of a 10p coin. When the customer has inserted an appropriate amount of money, they can receive a cup of coffee (which we will label *givecoffee*) or tea (*givetea*).

An act of communication may take place within a single part of the system being described, between two separate parts, or between the system and its environment. Depending on the level of abstraction at which the system is viewed, the same act of communication could be viewed as internal to a component (in a very high-level view), between two separate components, or between the system and its environment (in a low-level specification of a small part of a larger system).

We can see this kind of difference if we imagine that our coffee machine has a separate subsystem called *coinbox* which takes coins and reports a total when appropriate, and a *dispenser* which dispenses drinks in response to these reports. (We shall name the complete machine *coffeemachine*.) Let us invent two more actions for these two subsystems to use to communicate:

*total*10*p* and *total*15*p*.

When *cashbox* has taken a 10p and a 5p, it can report *total*15*p* to the *dispenser*, which would then *givecoffee*. The *total*15*p* action is internal to the *coffeemachine* process; a customer cannot observe it directly, although they might guess at its existence by observing the behaviour of *coffeemachine* as a whole. When we test only the *cashbox* or *dispenser* component on its own, however, the *total*10*p* and *total*15*p* actions are observable; *cashbox* uses those actions to communicate with its environment, which normally includes the *dispenser* process.

Note that we have not mentioned any way for the customer to tell the machine to deliver coffee or tea. If there were no way for the customer to order appropriately, the machine would have to guess, after two 5p coins had been inserted, whether to supply tea or whether to wait for another 5p and then supply coffee. A customer who wanted coffee could encounter a race situation, attempting to insert another 5p quickly before the machine dispensed tea.

We have arranged the specification this way partly in order to simplify the example, and partly to demonstrate that a specification can leave out such important details and still be useful. In the previous chapter, we discussed RM-ODP and its viewpoint model. This example demonstrates how a viewpoint specification might work. Our specification concentrates on those aspects required to ensure that the machine takes the correct money before dispensing beverages. Another specification could be written specifying more details of the customer interface, such as refunding money when a "cancel" button is pressed, giving change, and allowing the customer to order the drink they want.

For the moment, assume that we can specify *cashbox* and *dispenser*, each as a separate process which can interact with its environment. We can put them together using a technique called *parallel composition* to obtain a specification consisting of the two processes interacting with one another (using the *total*10*p* and *total*15*p* actions) and their environment (using the *take*5*p*, *take*10*p*, *givecoffee* and *givetea* actions).

At this stage, an observer can still see the communications between the two processes. We can prevent this by a second technique called *hiding*. If we take our pair of processes, compose them in parallel, and hide the *total*10*p* and *total*15*p* actions, the result is as before, except that the environment cannot observe either of the hidden actions. This process, with its internal details encapsulated, we label *coffeemachine*, and is a specification of the complete system.

Process algebraic specifications describe the behaviour of a process, which is taken to represent the behaviour of a system or subsystem. (Our *coffeemachine* process is intended to describe the behaviour of a real coffee machine, in a simplified form.) As we have seen, a process specification is generally built up from component behaviours, which in turn may be composed of other behaviours. We can identify a small number of primitive building blocks—communication, choice and parallel composition—from which all other behaviours can be built. In practice, a process algebra will provide a number of other constructs to aid specification; LOTOS, for example, has notation for constructs known as enabling and disabling, whose effect could be built up instead from the primitive constructs, although that would often result in a very complex specification.

A process communicates by performing actions. Communications internal to a process are represented by an "internal action", written i in LOTOS or τ in CSP and CCS. Internal actions are those which are hidden, and thus cannot be observed from outside the process, but which can cause changes of state. These changes of state are also not directly observable, but can in turn affect the process's future behaviour.

The end result is that, as in our coffee machine example, specifications are usually made up of a collection of parallel processes, communicating with one another and with their environment. Since parallel composition is used to aggregate the processes within one process specification, and could be used to aggregate that behaviour with others to form a more complex specification, it makes sense that the interaction between a process and its environment is considered to be an instance of parallel composition, too. Our coffee machine can be considered to be composed in parallel with the customers it is to serve. The customers can observe the machine only by interacting with it; this means that the definition of parallel composition is closely linked with the notion of *observable behaviour*. For instance, it is due to the way parallel composition is defined that internal actions are not observable.

Parallel composition of processes is arguably the most important notion in process algebra. A behaviour specification is not considered to define what the process actually does, but what it can potentially do. Its actual behaviour is a function of both its own specification and the behaviour of its environment. The process and its environment are considered to be composed in parallel with one another; thus the behaviour of the process depends on its environment.

In our vending machine example, the definition of *coffeemachine*, in CSP, might be as follows:

$$dispenser = (total15p \rightarrow givecoffee \rightarrow dispenser \mid total10p \rightarrow givetea \rightarrow dispenser)$$

$$\begin{aligned} cashbox = (take5p \rightarrow (take5p \rightarrow (&take5p \rightarrow total15p \rightarrow cashbox \mid total10p \rightarrow cashbox) \\ &\mid take10p \rightarrow total15p \rightarrow cashbox) \\ \mid take10p \rightarrow (take5p &\rightarrow total15p \rightarrow cashbox \mid total10p \rightarrow cashbox)) \end{aligned}$$

$$coffeemachine = (dispenser \mid\mid cashbox) \setminus \{total10p, total15p\}$$

(The notation $P \mid\mid Q$ denotes P composed in parallel with Q. $P \setminus S$ denotes P with all the actions in set S hidden.)

We can see that, in the same way that *dispenser* and *cashbox* are composed together in parallel, we could compose *coffeemachine* in parallel with, say, *customer*, or perhaps even *crowd*, a process consisting of a hundred *customer*s interleaved with one another. (Interleaving is a kind of parallel composition where the processes run independently, not interacting with one another.)

The specification above does not capture quite the behaviour we intended. When the actions $total10p$ and $total15p$ are hidden, they become internal and thus are performed autonomously by the *coffeemachine*. This leads the the following undesirable behaviour: the customer may insert two 5p coins, and then the *cashbox* process has a choice between another $take5p$ action or a $total10p$ action. This choice may be resolved immediately by *cashbox* performing $total10p$, which *dispenser* is also able to do. Therefore, we have a race condition, where the machine may either dispense tea or wait for the customer to insert another 5p, but the choice will be *nondeterministic*. Even if the customer is ready to insert another 5p immediately (by offering $take5p$ immediately) the machine may still choose to perform $total10p$ and then *givetea*. This kind of subtle problem can easily arise in process algebraic specifications.

There are a number of relations defined between process specifications, most of which try to codify a notion of *observation*. They typically relate a process to those other processes which exhibit all the behaviour the first process can exhibit when the processes are observed in a certain way. For example, the trace preorder relates processes according to the traces (sequences of actions) they can perform, as follows:

$$P <_{tr} Q \text{ exactly if } \mathrm{Tr}(P) \subseteq \mathrm{Tr}(Q)$$

($<_{tr}$ denotes the trace preorder; $\mathrm{Tr}(P)$ denotes the set of traces which P can perform.)

The trace preorder and other relations used to aid the development of process algebraic specifications are known as *development relations*. We will examine some of these relations in more detail in §4.3.

Many of the concepts described above were introduced by Milner[Mil89], who also describes the specification language CCS and the *bisimulation* relation between processes (which we will discuss further below).

Another important and influential process algebra is CSP, introduced by Hoare [Hoa85]. Several introductory books on process algebra are based on CSP, for example [Ros98] and [Sch00].

LOTOS[BB87, Bow96] is a process algebra standardised by ISO[ISO89]. It draws heavily on the prior experience of CSP, CCS and other process algebras. Although this thesis purports to deal with LOTOS, we will not, in fact, use LOTOS as defined by ISO; instead, we will use a subset of LOTOS which we will define in §3.3.

While each of these process algebras has its own individual syntax and supplies a different set of operators, their semantics are very similar. Labelled transition systems are often used as a "lowest common denominator" behavioural technique, since it is possible to translate specifications written in any process algebra to labelled transition systems.

Labelled transition systems, however, describe processes at a much lower level than CCS, CSP or LOTOS; they can capture less information about the structure the designer had in mind. They are useful, however, because most process algebras can easily be translated to labelled transition systems. Since it is intuitively obvious how a labelled transition system is supposed to work, and different people's interpretations of them are very similar, labelled transition systems are often used to define semantics for the process algebras. Many development relations, including all those we will examine in this thesis, are (or can easily be) defined over labelled transition systems, allowing them to be applied to more than one specification technique.

Since work on development relations has, in part, been continued separately for each process algebra, a large number of development relations are now defined. Most of these development relations were constructed as a formal method of testing whether an abstract implementation conforms to the requirements of a formal specification[Bri88].

Some examples of important development relations are reduction and extension[BS86, BSS87], testing equivalence[Led92], bisimulation[Mil89] and failures-divergences[Hoa85]. We will define some of these relations in §4.3 where we examine them in much more detail. Several of these development relations are also described in [Bow96], while [vG90] analyses the relations and proposes a partial ordering of relations based on their strength—for example, bisimulation is stronger than reduction because reduction relates all pairs of processes that bisimulation relates, and more.

As we will see below when we discuss subtyping relations, most of these development relations embody a notion of substitutability. There are a number of different ideas of what should be substitutable for what, so most of these relations are applicable in at least one situation.

The reduction relation, one of the most useful relations in LOTOS, does not allow the interface of a process to be enlarged; that is, it does not allow a process to actively respond to actions which a less reduced process could not respond to. This is a consequence of the purpose of the reduction relation, which is to ensure that a more reduced process differs from a less reduced process only in that it exhibits less nondeterminism. To achieve this, the more reduced process must be able to perform only traces which the less reduced process could perform, and only be able to refuse actions which the less reduced process can refuse; that is, the less reduced process is used as the maximal set of activities (traces and refusals) which any reduced process can exhibit.

It is often desirable that behaviour should be allowed to be added when a process is developed, and it can also be desirable for a subtype to exhibit behaviour which its supertype could not exhibit. For reasons which we will show later, reduction seems a good relation to use for subtyping; one of its drawbacks, however, is that it does not allow new behaviour.

2.2 State based specification

While process algebras are unusual to most programmers, state based specification notations are more familiar, based, as they are, on the same model as imperative programming. A state based specification consists of a description of the allowable states of the system and a specification of each of the operations which the system may perform, together with constraints on the state before and after the operation[WD96].

A few of the major state based notations are Z[WD96, PST96], Object-Z[CDD+90], and VDM[Jon86].

For example, we could specify a vending machine similar to that shown in the previous section, but in Z, as follows. Firstly, we define the state space of the machine:

CoffeeMachine
$balance : \mathbb{N}$

The machine simply keeps track of the amount of excess money it has taken (in pence).

Next, we define its initial state:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CoffeeMachineInit}} \\ \hline
\textit{CoffeeMachine} \\ \hline
\textit{balance} = 0 \\ \hline
\end{array}$$

We can now define operations over this state. The operation *Take*5*p* is analogous to the effect of the *take*5*p* action in the process algebra example.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Take}5p} \\ \hline
\Delta \textit{CoffeeMachine} \\ \hline
\textit{balance}' = \textit{balance} + 5 \\ \hline
\end{array}$$

Recall that *balance* represents the *balance* component of the state (in this case, the only component) before the operation; $balance'$ represents the same state component after the operation.

To make our *Take*5*p* operation more faithful to the behaviour of the original, let us modify the state schema to ensure that the machine cannot take more than 15p in advance:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{CoffeeMachine}} \\ \hline
\textit{balance} : \mathbb{N} \\ \hline
\textit{balance} \leq 15 \\ \hline
\end{array}$$

To complete the example, let us define *Take*10*p*, *GiveTea* and *GiveCoffee*:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Take}10p} \\ \hline
\Delta \textit{CoffeeMachine} \\ \hline
\textit{balance}' = \textit{balance} + 10 \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{GiveTea}} \\ \hline
\Delta \textit{CoffeeMachine} \\ \hline
\textit{balance}' = \textit{balance} - 10 = 0 \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{GiveCoffee}} \\ \hline
\Delta \textit{CoffeeMachine} \\ \hline
\textit{balance}' = \textit{balance} - 15 = 0 \\ \hline
\end{array}$$

Note that under the normal interpretation of Z specifications, this example should be interpreted differently from the CSP example. In the CSP example, the *givetea* and *givecoffee* actions are offered by the process at any appropriate point, while in the Z example, *GiveTea* and *GiveCoffee* would have to be explicitly invoked by the environment. There is also the issue that, in the CSP example, the *total*10*p* and *total*15*p* internal actions can occur within the machine, limiting the subsequent choices of the environment; this does not occur in the above Z example.

We can see that, in this example, we introduced some arbitrary-looking terms in the operation predicates and in the state schema to make the example's behaviour conform with the behaviour of the CSP version of the example. By some simple alterations to the Z specification, we can, for example, remove the limit of 15p paid in advance, or allow tea to be bought even if more than 10p has already been paid. By adding new operations, we can add new types of beverage or permit the insertion of different denominations of coin.

While it would be possible to make similar changes to the CSP version of the specification, the nature of process algebra would make such changes more difficult, and the modifications would have to be more invasive. Rewriting the CSP specification to use data, variables and assignment would mitigate these problems to an extent, but not eliminate it. By not using data in our process algebraic specification, we have merely exaggerated an existing difference in the suitability of the two paradigms to different problems.

Although the state based specification style seems very different from the process algebraic approach, a very similar set of systems can be described in each. A state based specification can be translated into a behavioural specification by writing a behaviour description corresponding to each state and taking as the translated specification the behaviour corresponding to the starting state. A process algebraic specification can be translated into a state based specification by labelling each behaviour the process can evolve to, creating a state corresponding to each, and specifying operation predicates which preserve the operation orderings found in the behavioural specification. Of course, there is no guarantee that either of these approaches will result in a well-formed specification; an infinitely long specification is one possible outcome. We will discuss the topic of translation between specification notations in more detail later in this chapter.

Many commonly used specification languages incorporate aspects of the state-based method. The Unified Modelling Language (UML) [BJR$^+$00b], for example, apart from the support it offers for designing class-based systems, also allows constraints to be imposed on the system's behaviour by means of the Object Constraint Language (OCL) [BJR$^+$00a]. An OCL expression may represent an invariant applied to an object's state, or a precondition or postcondition applied to an operation (we will explain these terms in the next subsection). The ability to constrain the state of a specification in this way is characteristic of state-based specification languages.

There are other ways to constrain state, however. Temporal Logic of Actions (TLA) [Lam94], for example, uses a method with aspects of both the state-based and behavioural approaches. TLA specifications may use operation predicates and invariants, like state-based techniques, but can also assert properties of the process's behaviour over time. For example, a state predicate may be quantified over the states through which the process will pass during its execution. The TLA "always" operator specifies that some state predicate will hold for every future state the process can evolve to, while the "sometime" operator specifies that the process must eventually reach some state where the predicate will be true.

As with process algebras, several relations have been defined over state based specifications. These relations are known as refinement relations, and are intended to indicate when one specification has all the requirements of another specification—that is, when one specification is the same as another, but is more specific in its requirements.

The relation which is probably most widely used is downward simulation[WD96, HHS86, HH90]. Downward simulation is used as the main refinement relation in Z, where it defines the allowable ways in which a specification may be developed. We will define the downward simulation relation below, and examine the definition of the Z refinement relation in detail in §3.2.2.

2.2.1 Undefined behaviour vs. guards

As mentioned above, state based specifications can place constraints on the system's state just before and just after each operation occurs. The constraints on the after-state (collectively known as the *postcondition*) specify the effects of the operation in the expected way, but it is not so obvious what the constraints on the before-state (the *precondition*) can achieve.

We saw preconditions in some of the operations in our example above. For example:

GiveTea
$\Delta \mathit{CoffeeMachine}$
$balance' = balance - 10 = 0$

In this case, the precondition is $balance = 10$. The precondition is the necessary and sufficient condition for the existence of an after-state satisfying the postcondition: if $balance \neq 10$, the operation's predicate cannot be made to hold by any value of $balance'$, but if $balance = 10$, the assignment $balance' = 0$ will satisfy the predicate.

In fact, there are several interpretations of the meaning of the precondition. The two main possibilities are that the precondition can specify *when* the operation may occur, or that it can specify the conditions under which the operation's results are well defined[Str95]. The former approach, Strulo's "firing conditions" interpretation, is reminiscent of Dijkstra's guarded commands. It is used in the Object-Z[CDD$^+$90] language. The latter approach is the one used in most state based notations, including Z, and is the one which will cause problems for us later in the thesis.

In the "firing conditions" approach, the object specified by the state based specification behaves like a process algebraic object; it accepts or refuses operations based on whether the operation's precondition holds. The object's behaviour is well defined in the case where the precondition is false: it does not perform the operation. In our example, this interpretation would result in the state based specification having behaviour very similar to the process algebraic version (subject to the issue of nondeterminism mentioned earlier); the *GiveTea* operation would not be able to occur unless $balance = 10$.

In the conventional, "undefined behaviour" approach, the object's behaviour is not well defined if an operation is performed on the object when the operation's precondition is false. Thus, in our example, it would be permissible to invoke the *GiveTea* operation at any time, but the results would only be well-defined in the case where $balance = 10$. Before implementing this specification, it would seem to be advisable to refine the specification by enlarging *GiveTea*'s precondition in order to ensure that an implementation dealt sensibly with cases outside the existing precondition. This approach is in contrast to the behaviour of a process algebraic object, whose behaviour is always defined unless its specification incorporates nondeterminism.

When we consider a given state based refinement relation, it usually implements some notion of substitutability or subtyping. Given that we know of two different interpretations of the case where an operation's precondition does not hold, we can see that the exact definition of a refinement relation depends on which interpretation is being placed on false preconditions. Thus, each refinement relation implies one particular interpretation of false preconditions.

In chapter 4, we will see how this causes problems when we try to create a translation from a state based notation, which uses the "undefined behaviour" approach, to a process algebra, whose interpretation is similar to that of the "firing conditions" interpretation, while also trying to fulfil the condition that state based specifications related by the state based subtyping relation should translate to processes related by an appropriate process algebraic development relation.

2.3 Subtyping

All notions of subtyping are closely linked with ideas of substitutability. This is clear when it is recalled that the reason for having a subtyping relation is to enable a typechecker to permit objects of one type to be used in contexts which expected an object of a second type. If it cannot be shown that that substitution can always be safely made, the first type cannot be considered a subtype of the second. A subtyping relation is thus a formalisation of a notion of substitutability.

The development relations of process algebra and the refinement relations used with state based notations are also concerned with substitutability, but in this case they model the process of developing a system. When a specification is refined or developed, a formal method requires that the new specification behave only in ways the old specification allowed. If the new specification can behave in a manner which was disallowed by the original specification, a system produced from the new specification could not be expected to meet the original specification, and so the new specification cannot be considered a refinement of the old.

It is clear that subtyping and refinement have much in common, but there are many details on which they can differ.

Subtyping is defined as substitutability: type A is a subtype of type B exactly if objects of type A may be used in any situation where an object of type B was expected, without the object's client (i.e. environment) being able to tell the difference. A more general view of type sees each object as having potentially many types, so the definition changes: type A is a subtype of type B exactly if all objects of type A must necessarily also be of type B. This definition emphasises the similarity of (sub)types with (sub)sets in set theory.

Subtyping is naturally a reflexive and transitive relation. Any object is clearly substitutable for itself, and if A is substitutable for B and B is substitutable for C, then it is clear that A is substitutable for C. Subtyping is thus a preorder.

2.3.1 Inheritance is not subtyping

Inheritance is quite a different concept from subtyping[CHC90], although they are often present together. Inheritance is an implementation technique allowing code to be shared between objects (or classes of objects), while subtyping is a relation between types. Since using inheritance to share code often results in behavioural similarities between the objects involved, it is often the case that subtyping holds between the types associated with the objects.

We can see that inheritance and subtyping do not imply one another using counterexamples. Consider an object of type *Point*, which represents a two-dimensional Cartesian point. *Point*'s state will be represented as follows:

Point
$x, y : \mathbb{R}$

We will now define $3dPoint$ as follows:

$3dPoint$
Point
$z : \mathbb{R}$

We have included the definition of *Point*'s state in our definition of $3dPoint$ by referring to it by name. This kind of inclusion by reference of some or all of the implementation of another class is the defining characteristic of inheritance.

Although *Point* and $3dPoint$ are related by inheritance, they are not necessarily related by subtyping. For instance, in a program which calculates the distance of a *Point* from the origin as $\sqrt{x^2 + y^2}$, substituting a $3dPoint$ would result in an incorrect calculation; the value obtained would give the $3dPoint$'s distance from the z-axis, rather than from the origin. Thus, inheritance does not imply subtyping.

We can also define types which *are* related by subtyping but *not* by inheritance. For instance:

ColouredPoint
$x, y : \mathbb{R}$
$c : colours$

ColouredPoint represents a 2-dimensional point with an associated colour. In this case, it is possible to use the *ColouredPoint* as a *Point*, without referring to the extra state c. The colour of the point does not affect our geometric calculations. However, *ColouredPoint* does not inherit from *Point*, despite being a subtype of it. Therefore subtyping does not imply inheritance.

In this thesis, we will not be concerned with inheritance. It may or may not be present in specifications related by subtyping, but this is irrelevant to our purpose, which involves subtyping only.

2.3.2 Signature-based subtyping

There has been much work on type systems for object oriented imperative programming languages, for instance [BCC+95, FM94, FM96]. Most object oriented programming languages include an inheritance mechanism and use this to guide the language's idea of subtyping; the type associated with a subclass is generally considered to be a subtype of the type associated with its superclass. The rules by which the language allows new classes to be defined using inheritance therefore produce that language's subtyping relation.

As described above, an object of any particular type can masquerade as, or stand in for, an object of any of its supertypes. This means that, for type A to be a subtype of type B, A must have at least all the operations that type B has. (We can regard data members simply as pairs of operations, one of which takes no parameters and returns the value of the data member, while the other takes a single parameter and assigns this value to the data member.)

Let us imagine we have a system with three types of object: X, which is not a subtype of either of the other two types; Y, which is a subtype of X; and Z, which is a subtype of Y and therefore also a subtype of X. Consider the following pseudocode interface signatures for A and B:

```
A.hefalump (X, Y) -> Y
A.woozle (Z) -> X

B.hefalump (X, Y) -> Y
```

These definitions say that B has one operation, called *hefalump*, which takes an X and a Y and returns a Y. A has a similar *hefalump* operation, which we will assume is implemented the same way as B's *hefalump* operation, but it also has a *woozle* operation, which takes a Z and returns an X.

It will now be possible to use an object of type A as if it were of type B, invoking any operation available on a type B object (i.e. *hefalump*), confident that the type A object will be able to respond properly. In fact, it is not even necessary to know whether the object we are working with is of type A or B; all type B's operations will work, regardless. This property is named *the rule of subsumption*.

Things work as described above if the type of each operation in A has the same signature as the type of the corresponding operation in B. However, it is also possible to vary the type of operations without upsetting the rule of subsumption. In particular, the types of the operations must be subtypes: the operation type in A must be a subtype of the operation type in B.

Consider what happens if we change A's *hefalump* operation like so:

```
A.hefalump (X, Y) -> Z
```

A now returns a Z from a *hefalump* operation.

Since Z is a subtype of Y, this new return value is now usable in any situation where the old return value was usable. By changing the return value of an operation to a subtype, we get a subtype of the original operation type.

We now change the signature again:

```
A.hefalump (X, X) -> Z
```

A's *hefalump* operation will now expect an X as its second parameter, instead of Y. Since X is a supertype or Y, any Y will be usable as an X, and so this operation will accept a Y parameter. A's *hefalump* operation is still a subtype of B's *hefalump* operation.

In general, an operation which is to be a subtype of a second operation must have a return type which is a *subtype* of the return value of the second operation, and parameter types which are *supertypes* of the corresponding parameters in the second operation.

Because the types of return values change "in the same direction" as the type hierarchy (i.e. return values can be subtypes in subtypes) and this is the way we would expect, the way return types can change is called *covariant*. Parameter types can change only "in the opposite direction" to the type hierarchy, so they are called *contravariant*.

Four rules can be extracted from the above discussion, which together are sufficient to prove that one type can be used in place of another without causing the system to fail a type-check. Firstly, each operation in the supertype must have a corresponding operation in the subtype. Secondly, these corresponding operations must each have the same number of arguments. Thirdly, the argument types must be contravariant. Fourth, the result types must be covariant.

Together, these four rules form a subtyping relation, which we call *signature-based subtyping*. Most languages implementing a subtyping relation will include at least these four basic requirements—corresponding operations, matching arguments, contravariance, and covariance—to ensure type-safety.

The language Eiffel[Mey92] implements a different signature-based subtyping relation. It allows an overriding method to vary the result covariantly, as in the contravariance/covariance method. However, it also allows the argument types to be covariant, but requires that an implementation carry out a complex global analysis of the program to ensure that this does not allow any operation to be invoked on an object which is not defined to be able to handle that operation correctly.

In this thesis, we will not be concerned with signature-based subtyping, except inasmuch as we assume that it is fulfilled. A signature-based subtyping check of one sort or another is necessary to ensure that all the operation invocations in the program are performed on objects which are defined to be able to handle the operation in question, and to ensure that the correct number and type of arguments are supplied. The subtyping relation we will be concerned with is behavioural, and signature-based subtyping is clearly a prerequisite for it.

2.3.3 Behavioural subtyping

In the context of an ODP system, signature-based subtyping alone turns out not to be a strong enough relation; it is not enough merely for the subtype's operations to have the same names and accept appropriately-related argument types. For correct operation, aspects of the type's behaviour must also be taken into account[ST97, LW94].

Behavioural subtyping, as its name suggests, is concerned with the behaviour of objects. Specifically, it requires that the behaviour of instances of a subtype be (in some sense) *compatible with* the behaviour of instances of its supertype.

We can also think of behavioural subtyping as a generalisation of signature-based subtyping. Signature-based subtyping requires that if any method available in the supertype is called on any instance of a subtype, and the arguments passed are of the corresponding types declared for the supertype method's arguments, then the subtype method will run to completion and return a result of the type declared for the supertype method's return value. Behavioural subtyping is less specific: when a subtype method is used in accordance with the supertype's specification, the results must be in accordance with the supertype's specification.

Another way of stating this requirement is that it should not be possible to implement the subtype specification in such a way that the implementation can violate the supertype specification.

In a similar way to the *draw* example we used in §1.5, we can demonstrate the difference between signature-based and behavioural subtyping. Suppose we have two types, A and B, each with a method named m. A's method takes a natural number as its argument and returns double that number. B's method also takes a natural number as argument, but it returns the successor of its argument. Since both methods m take one natural number as argument and return one natural number as result, signature-based subtyping does not distinguish them; A is a signature-based subtype of B and B is a signature-based subtype of A. However, A's m does not meet the specification for B's m ("return the successor"), so A is not a behavioural subtype of B. Similarly, B's m does not meet the specification for A's m ("double your argument"), so B is not a behavioural subtype of A.

State-based

The "behavioural" aspect of behavioural specification is that behavioural specifications deal with the ordering of events. Consistent with this, behavioural subtyping can also deal with the ordering of events. However, it is not necessary to use a behavioural specification technique to apply a behavioural subtyping relation.

State-based specification notations do not provide a way to explicitly specify the ordering of operations, but operations' preconditions can be used to impose constraints on their ordering. Although this indirect approach makes it less obvious how we can apply a behavioural relation to state-based specifications, several such relations have been proposed.

America[AvdL90] defines one of the simplest behavioural subtyping relations on state-based specifications. His relation makes use of declared symbolic *properties* declared in the classes whose types are being compared. These properties need only be named and declared; the specifier is responsible for giving meaning to the properties.

America's behavioural subtyping relation includes the conditions comprising signature-based subtyping, outlined above. It includes one further condition: all the properties possessed by a type must also be possessed by all of its subtypes. This simple rule requires discipline on the part of the specifier: the type system can have no knowledge of the meaning of a property, and therefore cannot check that it has been declared correctly. However, this subtyping relation is capable of checking that any given property of a supertype is preserved by its subtypes, provided that the specifier has named that property in the specification. Similarly, any property can be ignored, and not required in subtypes. This allows the specifier to decide which properties are significant for the purposes of subtyping, and require exactly those properties to be preserved in subtypes.

He, Hoare and Sanders[HHS86, HH90] define *upward* and *downward simulations*, which are closely related to their notion of *data refinement*. We already mentioned downward simulation in §2.2.

A simulation is a relation from the state space of one type to another. The two types are called the *abstract* type and the *concrete* type. A downward simulation relates states in the abstract type to those in the concrete type; upward simulations relate states in the concrete type to those in the abstract type. We can define the *downward simulation relation* to relate a pair of types exactly when there exists a downward simulation between them. The upward simulation relation is defined similarly.

A downward simulation must have certain properties. In He and Hoare's relational notation, they are as follows:

$$\begin{aligned} CI &\subseteq AI;\ R \\ R;\ CF &\subseteq AF \\ R;\ Cop_i &\subseteq Aop_i;\ R \end{aligned}$$

In these definitions, AI is the abstract type's initialisation: it relates states in some global state space to initial states of the abstract type. AF is the abstract type's finalisation: it relates final states in the abstract type to states in the global state space. With operations indexed by i, Aop_i is an operation on the abstract type, relating abstract states to abstract states. Similarly, CI, CF and Cop_i are the concrete initialisation, finalisation and operations. R is the downward simulation, relating abstract states to concrete states. ";" is the relational forward composition operator: $x\ R;\ S\ y$ exactly if $\exists\, z$ such that $x\ R\ z$ and $z\ S\ y$.

Informally, these inequalities read as follows. Any concrete initialisation must have a corresponding abstract initialisation to which its resulting initial state can be related. A concrete finalisation is not permitted if it can be used to finalise the object from a concrete state for which there is a related abstract state from which no abstract finalisation could be used. Given any abstract before-state, any concrete before-state to which it is related, any operation which can be invoked in that concrete before-state and any concrete after-state which can result, it must be possible to invoke the corresponding abstract operation in the abstract before-state such that the abstract after-state is related to the concrete after-state.

It is not necessary to understand the downward simulation relation in detail in order to understand this dissertation; the Z refinement relation is a more easily understandable instantiation of the basic idea behind downward simulation. We will explain the Z refinement relation in detail in §3.2.2.

Liskov and Wing also introduced a behavioural subtyping relation defined over state based specifications[LW94]. Their relation is formed as the intersection of several rules. The "signature rule" requires conditions which enforce signature-base subtyping, as discussed above. The "methods rule" contains conditions on the precondition and postcondition of each method which implement the downward simulation relation, in a very similar manner to Z refinement.

The novel aspect of Liskov and Wing's approach is the additional rule on computation histories, which is intended to ensure safety in the case where the object is being used concurrently by multiple clients. Liskov and Wing's type specification notation includes a notion of *history properties*: properties which hold over *histories*. A computation is a sequence of states through which an object passes, alternating with the operations causing the state transitions. A history is the sequence of states taken from one computation. For example, a monotonically increasing counter whose state is a single integer might have two operations: *inc*, which increments the counter, and *read*, which returns the current counter value. A computation of such a counter might be as follows:

$$3 \xrightarrow{read} 3 \xrightarrow{inc} 4 \xrightarrow{inc} 5 \xrightarrow{read} 5 \xrightarrow{inc} 6$$

The corresponding history is

$$3, 3, 4, 5, 5, 6$$

History properties express relations which must hold between any two states in a history of the object, provided the first occurs before the second in that history. For instance, the property that our counter increases monotonically might be written as follows:

$$c_\rho \geq c_\psi$$

(Here c_σ is the value of the counter in state σ, ρ is some state in the counter's history, and ψ is some state in the same history later than ρ.) History properties can be written into a specification, and are then called *constraints*.

Two alternative formulations of the rule on computation histories are given by Liskov and Wing: the "constraint rule" ensures that any explicitly-specified constraints in the supertype are met by the subtype, while the "extension map" rule requires that the effects of any newly introduced operation in the subtype can be "explained" as a combination of the effects of existing operations. Either of these rules is supposed to ensure that the view a given client has of the object is self-consistent: in our example, a single client should never see the value of the counter decrease.

Liskov and Wing's subtyping relation is thus a combination of downward simulation, which is accepted as a subtyping relation, and an extra condition on the histories of the object, which extends the subtyping to cope with multiple clients.

Process Algebra

Many relations have been defined over process algebras and other kinds of behavioural specification, a few of which have been drawn from the state-based setting. The commonly used relations include the relations based on traces, failures, refusals, and bisimulation. We will examine many of these relations in more detail in §4.3. Unfortunately we cannot easily tell which of these relations most closely correspond to subtyping.

Several workers have addressed the problem of a process algebraic subtyping relation; we now cite the work of a few of them.

Nierstrasz[Nie95] takes the view that objects provide services, and that operation ordering should be viewed as *non-uniform service availability.* He then develops a notion of types based on this idea, and defines a subtyping relation, named *request substitutability*, over these types. Request substitutability turns out to be identical to the LOTOS extension relation[BSS87]. (We discuss extension in §4.3.5.)

Cusack et al.[CL90, CRS90] examine subtyping by imposing requirements on it and then examining several common subtyping relations with respect to these criteria. The requirements are formulated as a series of "partial" subtyping relations, each of which allows a different kind of change to be made in a subtype. The LOTOS extension relation is again taken to be subtyping.

Rudkin[Rud92] examines LOTOS with a view to determining how to implement inheritance between LOTOS specifications. In the process, he examines the issue of subtyping, and determines that the LOTOS extension relation should be used as a subtyping relation.

Bowman et al.[BBSDS97] set out to determine a LOTOS relation to be used as a behavioural subtyping relation, taking the downward simulation relation on state-based specifications as a reference point. Using a technique for representing, in a process algebra, the "unspecified" behaviour of a state based object which occurs when an operation is invoked outside its precondition, they then conclude that the LOTOS reduction relation corresponds to downward simulation.

We can see that many researchers have taken the LOTOS extension relation to be behavioural subtyping. In this dissertation, we will be exploring the work reported in [BBSDS97] in more detail, and developing the theory that the LOTOS reduction relation corresponds more closely to the state-based subtyping relation—downward simulation—than does extension.

2.4 Conclusion

In this chapter, we have described in more depth the concepts of the state-based and process algebraic specification methods. We then investigated subtyping, looking briefly at the purely signature-based subtyping which we must assume is fulfilled to ensure type-safety. We then discussed behavioural subtyping, in both the state-based and process algebraic settings. We covered the downward simulation and Liscov and Wing subtyping relations, which are accepted as being useful state-based instantiations of subtyping.

In the next chapter, we will prepare to construct our translation by defining the languages and notations we will use.

Chapter 3

Specification notations

In the previous chapters, we have seen what subtyping is, and why we are interested in it. [LW94] contains a definition of behavioural subtyping for type specifications written in a state-based notation with preconditions and postconditions. This definition interests us because it has been designed to take a type's behavioural aspects into account.

In the previous chapter, we found that Liskov and Wing's behavioural subtyping relation contains, at its core, conditions very similar or identical to downward simulation and the Z refinement relation. The difference is in the addition of some extra restrictions on newly introduced operations, and on continued computations or histories. Rather than using Liskov and Wing's relation in our work, we will concentrate on the Z refinement relation instead.

Downward simulation is defined on state-based specifications. We would like to use process algebraic specifications for some of our type specifications, because process algebras capture an object's interaction with its environment in a more direct manner than state-based notations do. Therefore, we would like to find a relation on process algebras which corresponds (as defined in §1.5) to Liskov and Wing's behavioural subtyping relation.

Rather than attempting to translate Liskov and Wing's relation into a process algebra directly, our work will concentrate on attempting to find a process algebraic relation corresponding to the Z refinement relation.

Since Z itself is a very complex language, we will use a simplified state-based specification language together with the Z refinement relation as our state-based specification language and associated subtyping relation.

In this chapter, we will introduce the systems of notation we will use in the following chapters, and then present several examples to show how they are used. The examples we present here will also be used in later chapters.

We first present a simplified state-based notation, which emphasises those characteristics of the specification we will need when we come to translate specifications. We also show how we can apply the Z refinement relation to specifications written in these notations.

Next, we introduce briefly the subset of the LOTOS process algebra which we will use as the target language of our translation. (A full description of this subset appears in appendix A, which also describes labelled transition systems, and shows how they are used to define the semantics of LOTOS by means of inference rules which generate labelled transition systems from LOTOS processes.)

Finally, we will examine the notations presented and discuss their similarities and differences. The similarities provide the foundation of the translation we will present in a later chapter, while the major differences will cause the problems which we will overcome in the following chapters.

3.1 A state-based specification notation

We now present a simple notation for state-based specifications, based loosely on Liskov and Wing's notation[LW94], and some example specifications written in that notation.

In our simplified notation, a type consists of a state-space, a set of initial states, and one predicate for each operation. For simplicity, we will only consider cases where all operations take one argument and return one result. All other cases are special cases of this case. If more than one value was to be passed, a tuple of the required values can be passed instead. If no value was to be passed, a dummy value can be passed instead; "no value" can be considered to be the value of the unit type.

To regularize this, we can consider that each operation always takes a tuple as its argument and returns a tuple. If n values are to be passed, an n-tuple is used. Note that, if $n = 0$, a 0-tuple is used—effectively the unit type. We can therefore use a notation for argument and return values superficially very similar to that used for many procedural programming languages. However, to avoid confusion, we will often write the 0-tuple as *nil* instead of (), and for a 1-tuple, we will usually just write the value contained in the tuple.

Our notation calls for specification of the data type of the argument and result. In the examples we present, we will be using types corresponding to the following sets: the set of natural numbers, $\mathbb{N}$; the unit type, which we will call *nil*, but which is actually the set containing just the value *nil*, and other sets such as $\{1, 2, 3\}$ or $\mathbb{N} \cup \{\mathit{VOID}\}$, where *VOID* is a data value distinct from all the members of $\mathbb{N}$.

The concrete syntax we will use looks like this:

> *spec* :
>
> **State space**: $\{val_1, val_2, \ldots, val_n\}$
> **Initial states**: $val_i, val_j, \ldots$
> **Operation** op_1 $(in_1 \to out_1)$: $f_1(\mathit{before}, \mathit{arg}, \mathit{result}, \mathit{after})$
> **Operation** op_2 $(in_2 \to out_2)$: $f_2(\mathit{before}, \mathit{arg}, \mathit{result}, \mathit{after})$
> ...

This specification is named *spec*, and has a state-space consisting of the values $val_1, val_2, \ldots, val_n$. Each of its initial states, $val_i, val_j, \ldots$, must be a member of its state-space. Each operation, $op_1, op_2, \ldots$, has an argument type, $in_1, in_2, \ldots$, a result type, $out_1, out_2, \ldots$, and a predicate, $f_1, f_2, \ldots$ comprising a boolean function of four arguments, as described below.

The data types (argument type and result type) are syntactic sugar. They should be interpreted as causing two terms to be conjoined with the predicate. If the argument and result types are *in* and *out*, and the predicate is given as *pred*, the actual predicate of the operation is effectively $\mathit{pred} \wedge \mathit{arg} \in \mathit{in} \wedge \mathit{result} \in \mathit{out}$. It can be seen that *in* and *out* are, in fact, merely sets of values, and not true types. When we wish to specify that *in* or *out* is the set containing just the value *nil*, we will specify *nil* as the name of the type.

The behaviour of an object implementing one of these state-based specifications is intended to be as follows:

- The state-space is a collection of values used to represent all the possible states that an object can be in.
- When the object is created, it begins in one of the initial states. The choice of which initial state is up to the implementation, and need not be deterministic.
- The predicate of an operation is a Boolean function of the before-state (*before*), after-state (*after*), arguments (*arg*) and results (*result*) of the operation. The operation is defined with the given arguments in a given before-state if the predicate can be made to hold by some assignment of after-state and result value. If the operation is performed with such an argument and in such a before-state, then the operation will complete with a result value and after-state such that the predicate holds. Note that nondeterminism can be

specified by allowing more than one assignment of after-state and result value to satisfy the predicate.

Some notations require the behaviour of operations to be specified by giving two Boolean functions: a precondition and a postcondition. The predicate in our notation takes the place of both pre- and postcondition, with the precondition being calculated from the predicate as described below.

We can determine the precondition from the predicate as follows: the operation's precondition holds for some before-state and argument value exactly if it is possible to find some result value and after-state such that the predicate holds. That is,

$$pre = \exists\, result, after \bullet pred \wedge arg \in in \wedge result \in out$$

If the precondition does not hold with a given before-state and argument, the intent is that the result of invoking the operation is unspecified in that before-state and with that argument.

In a typical object oriented programming language, such as C++, this would mean that the class implementing the type would not define that operation, which in turn would mean that the compiler would not allow that operation to be invoked on any object of that type.

In our case, however, we are dealing with specifications, not concrete implementations. If an operation is unspecified, this means that we place no requirements on its implementation at all: it may be left unimplemented, or it may be implemented with any behaviour whatsoever. As a result of this, it makes sense to add a new operation in a subtype which was left unspecified in the supertype. For that operation, the supertype's behaviour was unspecified, so whatever the subtype's actual behaviour, it will necessarily meet the supertype's (most undemanding) specification.

3.1.1 Examples

We now present a specification of a one-place buffer which can hold any single natural number, in our notation. The state-space is a disjoint union containing the natural numbers and a special value *VOID* to indicate an empty buffer.

*buffer*1 :

State space: $\mathbb{N} \cup \{ VOID \}$
Initial states: *VOID*
Operation receive ($\mathbb{N} \to nil$): $before = VOID \wedge after = arg$
Operation transmit ($nil \to \mathbb{N}$): $before \neq VOID \wedge result = before \wedge after = VOID$

This specification begins in the state *VOID*, and has two operations, *receive* and *transmit*. The predicates are chosen so that the *receive* operation is only defined when the buffer is empty (in state *VOID*), and the *transmit* operation is only defined when the buffer is full (in state 0, 1, 2, ...).

Recall that we use *before* and *after* in the predicates to indicate the state of the object before and after the operation, respectively. *arg* indicates the argument supplied to the operation, and *result* refers to the result value which will be returned from the operation.

As mentioned above, the data types for each operation are interpreted as additional conjuncts to the operation's predicate. This means that the full predicate for the *receive* operation is

$$before = VOID \wedge after = arg \wedge arg \in \mathbb{N} \wedge result = nil$$

Similarly, *transmit*'s full predicate is

$$before \neq VOID \wedge result = before \wedge after = VOID \wedge arg = nil \wedge result \in \mathbb{N}$$

By using these augmented predicates, we will not have to worry further about data types for arguments and results; we will simply treat argument and return values as being drawn from a

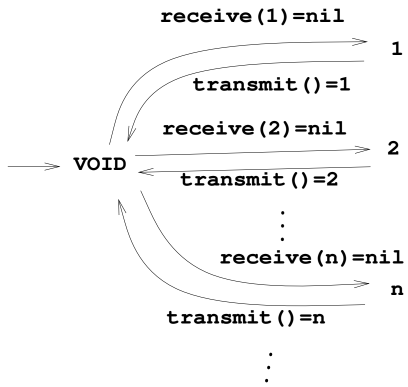

Figure 4: *buffer*1 visualised as a state machine

set of all possible argument and return values, and let the predicate determine which values are allowed for a particular operation. We will revisit this idea in more detail in §6.5.

We can visualise the *buffer*1 specification as a state machine, as shown in figure 4. In this state diagram, each arc is labelled with the name of the operation causing the transition, the argument passed (in brackets), and the result returned (after the equal sign).

There is something of a contradiction in this example. The state-based notation we use here is based on the imperative model, where the object we are specifying is an object, in the "object oriented programming" sense, with a collection of procedures, or methods, which may be called from the surrounding program. The communication here is essentially asymmetric; there is a "call" and a "return". The LOTOS notation views communication as a symmetric phenomenon: when an action occurs, LOTOS notes the fact, but does not attempt to capture which party was responsible for causing the interaction. This will cause some complications when we begin translating state-based specifications into LOTOS—the translation will be slightly complicated by the need to preserve information distinguishing call and return in a context which is not normally concerned with such things.

The specification *buffer*1 is written in the state-based notation, but, in anticipation of the following chapters, it specifies a process-oriented structure: a buffer.

3.1.2 Stacks, bags and queues

This section introduces the examples which we will be considering in depth in the following chapters.

We will consider three types of object: stacks, bags and queues. These are types of "container" objects into which data items may be put, and from which the data items may later be retrieved. The specification of the bag type does not determine the order in which items are retrieved, whereas stacks always return the item most recently inserted, and queues return the item least recently inserted. Intuitively, we would consider stacks and queues to be subtypes of bags, because they are more tightly specified. The only difference between these types is the order in which values are returned from the *get* operation, so this example should test whether our model is sufficiently discriminating to reveal this difference.

We will now specify our bag type. Since we want to be able to put multiple instances of the same value into the bag, our state-space will be defined using multisets: sets which can contain multiple instances of any particular value. We will write multisets using "half-diamond" brackets: for example, $\Lbag\ 0, 3, 2, 7\ \Rbag$. Note that order does not matter—$\Lbag\ 2, 3, 3\ \Rbag$ is the same multiset as $\Lbag\ 3, 2, 3\ \Rbag$. However, the number of times an item is mentioned does matter: $\Lbag\ 0, 1, 2\ \Rbag$ is a different multiset from $\Lbag\ 0, 1, 1, 2\ \Rbag$, and both of those are different from $\Lbag\ 0, 1, 2, 2\ \Rbag$. The union of two multisets, which we write $\cup$, preserves all elements in each set, including all duplicates.

Because we will later be drawing state diagrams of this specification, and translations of it, we would like to keep its state-space small. Therefore, we will restrict the contents of the bag to at most two items, each of which may be 0 or 1. While this would not be useful in a real-world situation, this is enough to exhibit the behavioural differences we want to examine.

A bag is a container object into which values can be placed by *put* operations; they may later be retrieved by *get* operations, but the implementation may return values in any order. The bag specification is as follows:

bag :

State space: multisets of $\mathbb{N}$
Initial states: $\langle\!|\,|\!\rangle$
Operation put $(\{0,1\} \to nil)$: $|\,before\,| < 2 \wedge after = before \cup \langle\!|\ arg\ |\!\rangle$
Operation get $(nil \to \{0,1\})$: $|\,before\,| > 0 \wedge before = after \cup \langle\!|\ result\ |\!\rangle$

We now define a similar stack and queue, using Miranda-style lists for the state-space. These lists are written with square brackets: [...]. List concatenation is written "++". Order and duplication matter in lists, so $[1,0,2] \neq [0,1,2] \neq [0,1,1,2]$.

A stack is like a bag, except that it returns values from the *get* operation in last-in, first-out order: each *get* operation returns the value which was most recently *put* and which was not yet returned by *get*.

$stack$:

State space: lists of $\mathbb{N}$
Initial states: $[\,]$
Operation put $(\{0,1\} \to nil)$: $|\,before\,| < 2 \wedge after = before + +[arg]$
Operation get $(nil \to \{0,1\})$: $|\,before\,| > 0 \wedge before = after + +[result]$

A queue is like a bag, except that it returns values from the *get* operation in first-in, first-out order: each *get* operation returns the first value *put* which has not yet been returned by *get*.

$queue$:

State space: lists of $\mathbb{N}$
Initial states: $[\,]$
Operation put $(\{0,1\} \to nil)$: $|\,before\,| < 2 \wedge after = before + +[arg]$
Operation get $(nil \to \{0,1\})$: $|\,before\,| > 0 \wedge before = [result] + +after$

In the English descriptions of the specifications above, we said that a stack or a queue is "like a bag, except ...". This is the kind of language often used when discussing inheritance in object orient programming languages, and the presence of inheritance often suggests that subtyping is likely to be present.

We now examine the Z refinement relation, and discuss its use as a subtyping relation for state-based specifications in our notation. We will then use it to show that the stack and queue types above are indeed subtypes of the bag type.

3.2 A subtyping relation for state-based specifications

The relation we will use to model subtyping for our state-based specification language, is the Z refinement relation[Spi92]. In this section, we will give a definition of the Z refinement relation, and show how it can be applied to our state-based notation.

3.2.1 Notation

Z refinement relates an abstract specification, *Aspec*, and a concrete specification, *Cspec*. In order to use this relation as a subtyping relation, we must relate the terms used in defining the relation to the terms used in our specification notation.

Aspec corresponds to the supertype, and *Cspec* corresponds to the subtype. In order to show that *Cspec* is a Z refinement of *Aspec*, it is necessary to find an *abstraction relation* which satisfies the conditions of Z refinement. An abstraction relation relates states in *Aspec* to states in *Cspec*. We will show an example of an abstraction relation later in this section.

Astate is a variable referring to a state in the abstract specification, and *Cstate* similarly refers to a state in the concrete specification. *Abs* is a predicate which holds exactly if *Astate* and *Cstate* are related by the abstraction relation. *Astate*, *Cstate* and *Abs* are used in "primed" forms, too, to refer to a second pair of states: Abs' holds if $Astate'$ and $Cstate'$ are related by the abstraction relation.

When we are discussing particular operations in *Aspec* and *Cspec*, we will normally refer to a pair of operations, one in *Aspec* and one in *Cspec*. The operation in *Aspec* is referred to as *AOp*, and the operation in *Cspec* is referred to as *COp*. In some notations, the names of *AOp* and *COp* need not be the same; *AOp* might be an operation named *get*, while *COp* might be named *transmit*. For our purposes, however, we will assume that *AOp* and *COp* always have the same name.

When we are discussing a particular operation invocation, we use the name *arg* to refer to the input value supplied to the operation, and *result* refers to the output value generated by the operation. *arg* and *result* are the same for both *AOp* and *COp*.

We will refer to the predicate of *AOp* as *pred AOp*, and to the predicate of *COp* as *pred COp*. Note that the predicate of an operation is not just what we write down in the specification, but, as described in §3.1, includes the extra terms implied by the argument and result types. Thus, if *AOp* is an operation written

Operation a $(in \rightarrow out)$: p

we have the predicate of *AOp*, *pred AOp*, as follows:

$$pred\ AOp = p \wedge arg \in in \wedge result \in out$$

Once we know the predicate of an operation, we can obtain its precondition. We will refer to the precondition of *AOp* as *pre AOp*, and to the precondition of *COp* as *pre COp*. With *AOp* defined as above, we can obtain *pre AOp* as described in §3.1:

$$pre\ AOp = \exists\, result, after \bullet p \wedge arg \in in \wedge result \in out$$

Lastly, we will be using the predicates *Ainit* and *Cinit*. *Ainit* is true exactly when *Astate* is an initial state of *Aspec*. Likewise, *Cinit* is true exactly when *Cstate* is an initial state of *Cspec*.

3.2.2 The Z refinement relation

We will now examine the Z refinement relation, in three pieces. It is necessary and sufficient for all three conditions to hold simultaneously for refinement to be present.

Since we are using the Z refinement relation as our state-based subtyping relation, we will sometimes refer to it using the symbol for subtyping, $\sqsubseteq$. We write $A \sqsubseteq B$ to mean that A is a subtype of B; that is, that A is a Z refinement of B. (In CSP and Z this notation is often used the other way around; we use it as described here for consistency with our usage of LOTOS relations.)

The first condition necessary for the refinement relation to hold is known as "weakening preconditions", and it states that, wherever the abstract specification can perform an operation, the concrete specification can too. This ensures that refining a specification does not result

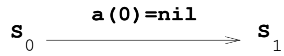

Figure 5: EG_1 visualised as a state machine

in a refined specification which is incapable of performing some operation which the original specification could perform.

The first condition is:

$$\forall\, Astate, Cstate, arg \bullet Abs \land pre\ AOp \Rightarrow pre\ COp$$

The second condition is known as "strengthening postconditions", and states that when an operation is carried out by the concrete specification, which the abstract specification could also have done, the result it produces is one of the results that the abstract specification could have produced. This ensures that, if the caller of an operation can handle only the return values the original specification can produce, this is sufficient to guarantee that it can handle any return value of any refinement of that specification.

The second condition is:

$$\forall\, Astate, Cstate, Cstate', arg, result \bullet \\ Abs \land pre\ AOp \land pred\ COp \Rightarrow (\exists\, Astate' \bullet Abs' \land pred\ AOp)$$

The third rule states that all the initial states in the concrete specification are related by the abstraction relation to initial states in the abstract specification. This ensures that any initial behaviour that the refined specification can exhibit could have been performed by the original specification.

The third and final condition is:

$$\forall\, Cstate \bullet Cinit \Rightarrow (\exists\, Astate \bullet Ainit \land Abs)$$

3.2.3 Simple refinement example

We will now examine an example, to demonstrate how the subtyping relation works. We will use two simple example specifications, EG_1 and EG_2, in the notation described above. EG_1 is specified as follows:

EG_1 :

State space: $\{S_0, S_1\}$
Initial states: S_0
Operation a ($\mathbb{N} \rightarrow nil$): $before = S_0 \land after = S_1 \land arg = 0$

This specification describes a type of objects. Each object can be in one of two states, S_0 or S_1, at any given time. There is a single operation, a, which can be applied to an EG_1 object. When it is applied to an EG_1 object in its initial state, S_0, with argument 0, the object moves to state S_1 and returns nil. All other operations on an EG_1 object are unspecified, so an object with more operations and states may still comply with the specification, provided it fulfils the specification's requirements.

EG_1 can be visualised as a state machine, as shown in figure 5. The notation we use here is the same as that which we used in §3.1.1. The single transition in this state machine therefore represents the operation a being passed the argument 0, and returning nil.

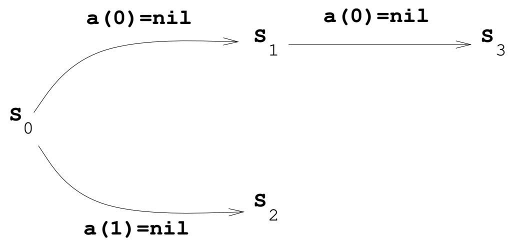

Figure 6: EG_2 visualised as a state machine

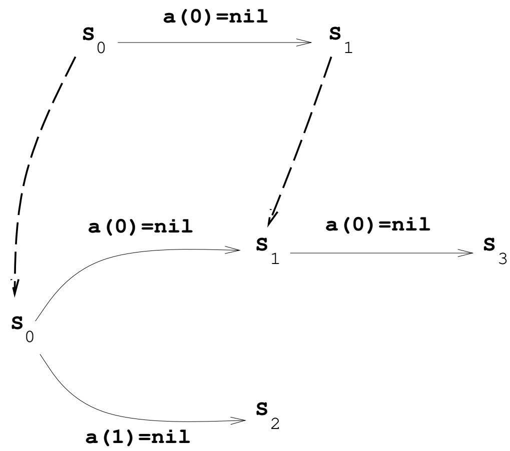

Figure 7: EG_1 and EG_2, linked by an abstraction relation

Here is a second specification, EG_2:

EG_2 :

State space: $\{S_0, S_1, S_2, S_3\}$
Initial states: S_0
Operation a ($\mathbb{N} \to nil$): $(before, arg, after) \in \{(S_0, 0, S_1), (S_0, 1, S_2), (S_1, 0, S_3)\}$

Visualised as a state machine, it is shown in figure 6.

The refinement relation requires an abstraction relation, linking together states in the two specifications. The subtyping relation relates the specifications exactly if a suitable abstraction relation exists. In this case, a suitable abstraction relation would be $\{(S_0, S_0), (S_1, S_1)\}$, which can be visualised as shown in figure 7.

We have now seen the two specifications, EG_1, which will act as *Aspec*, the abstract specification, and EG_2, which will act as *Cspec*, the concrete specification, and we have proposed an abstraction relation. We must now check that each of the three rules comprising the subtyping relation holds. If all of the rules hold, subtyping is present.

Each rule is implicitly universally quantified over all operations, but there is only one operation, a, in our specifications, EG_1 and EG_2. We therefore have to consider only this one operation. *AOp* is the operation named a in *Aspec*, while *COp* is the operation named a in *Cspec*.

Let us first determine *pred AOp*. *AOp* is a in EG_1. a's predicate in EG_1 is given as $before = S_0 \wedge after = S_1 \wedge arg = 0$. Combining this with the argument and result types given, we get

$$pred\ AOp = before = S_0 \land arg = 0 \land result = nil \land after = S_1$$

From this, we can determine $pre\ AOp$:

$$\begin{aligned} pre\ AOp &= \exists\, result, after \bullet pred\ AOp \\ &= \exists\, result, after \bullet before = S_0 \land arg = 0 \land result = nil \land after = S_1 \\ &= before = S_0 \land arg = 0 \end{aligned}$$

Let us now do the same for $Cspec$ (EG_2).

$$\begin{aligned} pred\ COp &= (before, arg, after) \in \{(S_0, 0, S_1), (S_0, 1, S_2), (S_1, 0, S_3)\} \\ &\qquad \land arg \in \mathbb{N} \land result \in \{nil\} \\ &= (before, arg, after) \in \{(S_0, 0, S_1), (S_0, 1, S_2), (S_1, 0, S_3)\} \land result = nil \end{aligned}$$

$$\begin{aligned} pre\ COp &= \exists\, result, after \bullet pred\ COp \\ &= \exists\, result, after \bullet (before, arg, after) \in \{(S_0, 0, S_1), (S_0, 1, S_2), (S_1, 0, S_3)\} \\ &\qquad \land result = nil \\ &= (before, arg) \in \{(S_0, 0), (S_0, 1), (S_1, 0)\} \end{aligned}$$

The first rule requires that:

$$\forall\, Astate, Cstate, arg \bullet Abs \land pre\ AOp \Rightarrow pre\ COp$$

Referring to the preconditions worked out above, we can see that the only way for the left-hand side of the implication to be true is if $Astate = Cstate = S_0$ and $arg = 0$. If this is the case, we can see that the right-hand side is also true, since $Cstate = S_0 \land arg = 0$ is one of its disjuncts. Thus, we can see that this first rule holds.

We will now check the second rule, which requires that:

$$\begin{aligned} &\forall\, Astate, Cstate, Cstate', arg, result \bullet \\ &\qquad Abs \land pre\ AOp \land pred\ COp \Rightarrow (\exists\, Astate' \bullet Abs' \land pred\ AOp) \end{aligned}$$

The left-hand side of this implication is similar to that of the one in the first rule, but with an extra conjunct. In this case, the entire predicate of EG_2's a operation must also apply. If $Astate = Cstate = S_0$ and $arg = 0$, $pred\ COp$ is satisfied only if $result = nil$ and $Cstate' = S_1$. So, if we assume that $Astate = Cstate = S_0$, $arg = 0$, $result = nil$ and $Cstate' = S_1$, we must now check that there exists an $Astate'$ such that Abs' and $pred\ AOp$ hold. And indeed there is such an $Astate'$, S_1. If $Astate' = S_1$ and $Cstate' = S_1$, then Abs' holds. Also, we can see that $pred\ AOp$ holds when $Astate = S_0$, $arg = 0$, $result = nil$ and $Astate' = S_1$. Thus, the whole rule holds, and the second condition is satisfied.

Finally, we must check the last rule:

$$\forall\, Cstate \bullet Cinit \Rightarrow (\exists\, Astate \bullet Ainit \land Abs)$$

There is only one initial state in $Cspec$ (EG_2), namely S_0. The initial state S_0 in $Aspec$ (EG_1) is related to it by the abstraction relation; thus this third rule holds.

Since all three rules hold, we find that EG_2 is a subtype of EG_1.

3.2.4 Refinement example: bag and stack

We now revisit the stack, bag and queue example specifications that we introduced in §3.1.2. We will now show that, using the Z refinement relation as our subtyping relation, the stack type is a subtype of the bag type—that is, that stack is a Z refinement of bag.

To do this, we let $Aspec$ be the bag type, and let $Cspec$ be the stack type, as defined in

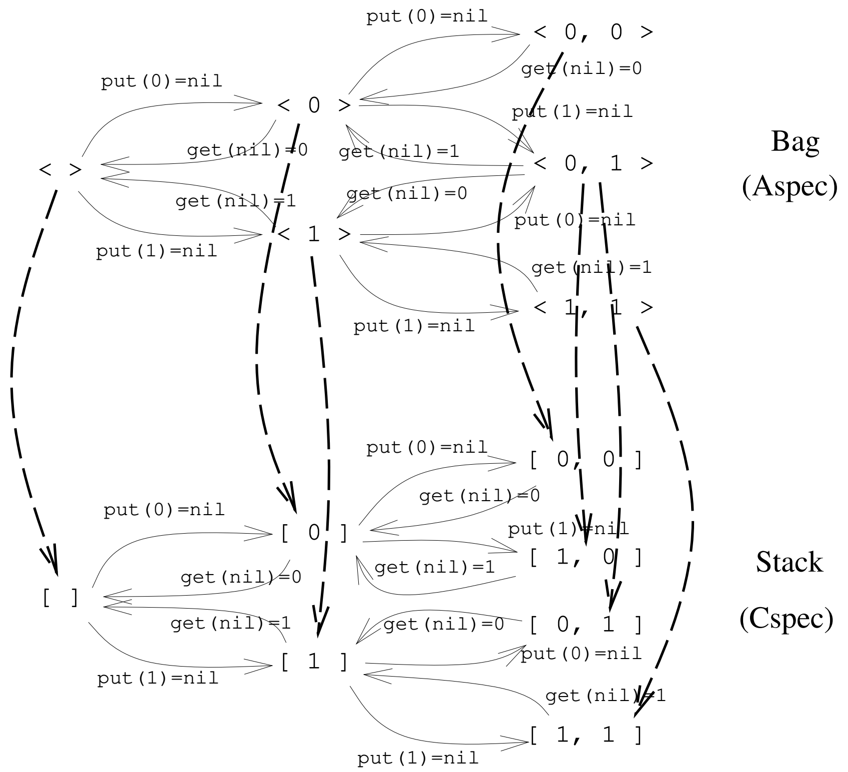

Figure 8: Stack and bag specifications with an abstraction relation

§3.1.2. We will then check that the Z refinement relation holds between these two specifications, and, since we are using the Z refinement relation as our subtyping relation, we will thereby have shown that stack is a subtype of bag.

The definitions of bag and stack were as follows:

bag :

State space: multisets of $\mathbb{N}$
Initial states: $\Lbag\Rbag$
Operation put $(\{0,1\} \rightarrow nil)$: $|\,before\,| < 2 \wedge after = before \cup \Lbag\, arg\, \Rbag$
Operation get $(nil \rightarrow \{0,1\})$: $|\,before\,| > 0 \wedge before = after \cup \Lbag\, result\, \Rbag$

stack :

State space: lists of $\mathbb{N}$
Initial states: $[\,]$
Operation put $(\{0,1\} \rightarrow nil)$: $|\,before\,| < 2 \wedge after = before + +[arg]$
Operation get $(nil \rightarrow \{0,1\})$: $|\,before\,| > 0 \wedge before = after + +[result]$

We can visualise our stack and bag specifications as state diagrams as shown in figure 8. This diagram also shows a proposed abstraction relation, which we can write down more formally as follows:

$$
\begin{aligned}
Abs = \{&(\Lbag\Rbag, [\,]), (\Lbag\, 0\, \Rbag, [0]), (\Lbag\, 1\, \Rbag, [1]), (\Lbag\, 0,0\, \Rbag, [0,0]),\\
&(\Lbag\, 0,1\, \Rbag, [1,0]), (\Lbag\, 0,1\, \Rbag, [0,1]), (\Lbag\, 1,1\, \Rbag, [1,1])\}
\end{aligned}
$$

We must now determine the predicates and preconditions of each operation. We will consider the *put* operation; a full proof would contain a similar proof for the *get* operation.

AOp and COp are the *put* operations of each of $Aspec$ and $Cspec$ respectively. We can calculate AOp and COp's predicates and preconditions as follows:

$$pred\ AOp = |\, before \,| < 2 \land arg \in \{0,1\} \land after = before \cup \langle\!|\ arg\ |\!\rangle \land result = nil$$

$$pred\ COp = |\, before \,| < 2 \land arg \in \{0,1\} \land after = before ++ [arg] \land result = nil$$

$$\begin{aligned} pre\ AOp &= \exists\, result, after \bullet pred\ AOp \\ &= \exists\, result, after \bullet |\, before \,| < 2 \land arg \in \{0,1\} \land after = before \cup \langle\!|\ arg\ |\!\rangle \\ &\qquad \land result = nil \\ &= |\, before \,| < 2 \land arg \in \{0,1\} \end{aligned}$$

$$\begin{aligned} pre\ COp &= \exists\, result, after \bullet pred\ COp \\ &= \exists\, result, after \bullet |\, before \,| < 2 \land arg \in \{0,1\} \land after = before ++ [arg] \\ &\qquad \land result = nil \\ &= |\, before \,| < 2 \land arg \in \{0,1\} \end{aligned}$$

As before, we must show that each of the three refinement conditions holds. The first condition is that:

$$\forall\, Astate, Cstate, arg \bullet Abs \land pre\ AOp \Rightarrow pre\ COp$$

Because $pre\ AOp = pre\ COp$, this condition is clearly true.

The second condition requires that:

$$\begin{aligned} &\forall\, Astate, Cstate, Cstate', arg, result \bullet \\ &\qquad Abs \land pre\ AOp \land pred\ COp \Rightarrow (\exists\, Astate' \bullet Abs' \land pred\ AOp) \end{aligned}$$

$$\begin{aligned} \equiv\ &\forall\, Astate : \text{multisets of } \mathbb{N}, Cstate, Cstate' : \text{lists of } \mathbb{N}, arg, result \\ &\quad \bullet (Astate, Cstate) \in Abs \\ &\qquad\qquad \land |\, Astate \,| < 2 \land arg \in \{0,1\} \\ &\qquad\qquad \land |\, Cstate \,| < 2 \land arg \in \{0,1\} \\ &\qquad\qquad\quad \land Cstate' = Cstate ++ [arg] \land result = nil \\ &\qquad\quad \Rightarrow (\exists\, Astate' \bullet (Astate', Cstate') \in Abs \land |\, Astate \,| < 2 \land \\ &\qquad\qquad arg \in \{0,1\} \land Astate' = Astate \cup \langle\!|\ arg\ |\!\rangle \land result = nil) \end{aligned}$$

$$\begin{aligned} \equiv\ &\forall\, Astate : \{\langle\!|\,|\!\rangle, \langle\!|\ 0\ |\!\rangle, \langle\!|\ 1\ |\!\rangle\}, Cstate, Cstate' : \{[\,], [0], [1]\}, arg : \{0,1\} \\ &\quad \bullet (Astate, Cstate) \in Abs \land Cstate' = Cstate ++ [arg] \\ &\qquad\quad \Rightarrow (\exists\, Astate' \bullet (Astate', Cstate') \in Abs \land Astate' = Astate \cup \langle\!|\ arg\ |\!\rangle) \end{aligned}$$

We must now check that this condition holds for all the combinations of $Astate$, $Cstate$, $Cstate'$ and arg. In fact, some of the combinations are independent of the value of arg, and for most values of $Astate$, only one value of $Cstate$ and $Cstate'$ is possible. We now check some of the possibilities.

If $Astate = \langle\!|\,|\!\rangle$, then we have $Cstate = [\,]$ and $Cstate' = [arg]$. $Astate'$ must then be $\langle\!|\ arg\ |\!\rangle$, and the condition $Astate' = Astate \cup \langle\!|\ arg\ |\!\rangle$ holds.

If $Astate = \langle\!|\ 0\ |\!\rangle$ and $arg = 0$, then $Cstate$ must be $[0]$ and $Cstate'$ must be $[0,0]$. It then follows that the only possible value of $Astate'$ is $\langle\!|\ 0, 0\ |\!\rangle$, and the condition holds.

If $Astate = \langle\!|\ 0\ |\!\rangle$ and $arg = 1$, then $Cstate = [0]$ and $Cstate' = [0,1]$. $Astate'$ must be $\langle\!|\ 0, 1\ |\!\rangle$, and, again, the condition holds.

With all the other combinations, the condition can also be shown to hold.

The third condition requires that:

$$\forall\, Cstate \bullet Cinit \Rightarrow (\exists\, Astate \bullet Ainit \land Abs)$$

There is only one initial state in *Cspec*, namely $[\,]$. *Aspec* also has only one initial state, $\langle\!|\,|\!\rangle$. Since these two states are related by *Abs*, the third condition holds.

For a full proof that stack is a refinement of bag, we would need to repeat the procedure above with the *get* operation, finding the values of *pred AOp*, *pred COp*, *pre AOp* and *pre COp*, then showing that each of the three conditions holds.

By showing that all three conditions hold for each operation, we show that *Cspec* (stack) is a refinement of *Aspec* (bag), and since we are using the refinement relation as our subtyping relation, this shows that stack is a subtype of bag.

More specifically, the preconditions of *get* are the same in both bag and stack, as are the predicates of *put*. However, the predicate of *get* is stronger in stack than bag: after the operations $put(0) = nil$ and $put(1) = nil$, bag is capable of performing either of the operations $get(nil) = 0$ or $get(nil) = 1$, while stack can only perform $get(nil) = 1$. This shows the effect of the "strengthening postconditions" condition in our subtyping relation: an operation which has a nondeterministic outcome in the supertype can be refined such that some or all of the nondeterminism is removed in the subtype.

We can also show that the queue type is a subtype of the bag type using a procedure very similar to that which we used above to show that stack is a subtype of bag.

On the other hand, the bag specification is not a subtype of either the stack or queue specifications. If bag *were* a subtype of stack or queue, we would be able to construct an appropriate abstraction relation. As we described above, though, the *get* operation in the bag specification can, in certain circumstances, return a value nondeterministically, while stack and queue are deterministic in those same circumstances. This prevents the construction of an abstraction relation. By showing that an appropriate abstraction relation cannot be constructed, we can show that bag is not a subtype of either stack or queue.

3.3 A subset of LOTOS

When we construct our translation, we will be translating to a subset of the LOTOS process algebra. We will not describe this subset in detail here; we do that in appendix A. In this section we assume that the reader is familiar with LOTOS, and merely give a brief listing of the constructs included in our LOTOS subset.

The syntax of our LOTOS subset includes the following constructs: process definition (of the form $P(\tilde{x} : \tilde{T}) := B$), deadlock ("*stop*"), action prefix ("$a;\ B$", "$a!e;\ B$" and "$a?x : T;\ B$"), internal actions ("$i;\ B$"), choice ("$B\ [\,]\ C$"), hiding ("$B \backslash G$"), process instantiation ("$P(\tilde{e})$"), parallel composition ("$B\ |[G]|\ C$"), and generalised choice ("$choice\ x : \{x_1, \ldots, x_n\}\ [\,]\ B$").

We represent traces using a CSP-like notation, for example $\langle\,\rangle$, $\langle a \rangle$, $\langle push, pop \rangle$. We also use these notations with their normal LOTOS meanings: "$\xrightarrow{a}$", "$\stackrel{b}{\not\rightarrow}$", "$\stackrel{\sigma}{\Longrightarrow}$", "$\stackrel{\sigma}{\not\Longrightarrow}$", "$P\ after\ \sigma$" "$\mathrm{Tr}(P)$", and "$\mathrm{Ref}(P, \sigma)$".

3.4 Comparison between notations

We have now examined the state-based specification notation we will be using in this dissertation, and outlined the subset of the LOTOS process algebraic specification notation which we will use. We have a subtyping relation for state-based specifications, which consists of the core conditions of Liskov and Wing's behavioural subtyping relation. We saw in chapter 2 that these conditions were the same as the conditions of the downward simulation relation, and of the Z refinement relation. Accordingly, we are using the Z refinement relation as our behavioural subtyping relation on state-based specifications.

We do not, however, have a subtyping relation for LOTOS specifications. In subsequent chapters, we will attempt to find one by constructing a translation from our state-based notation to LOTOS. This translation will allow us to take state-based specifications and convert them into LOTOS. We will then attempt to translate our subtyping relation, too, by finding a relation between LOTOS processes (which will be our LOTOS subtyping relation) such that if two state-based specifications are related by subtyping, their corresponding translated LOTOS counterparts will be related by the LOTOS subtyping relation.

To do this, we will have to do two things. We must construct our translation, and we will need to find some likely candidate relations between LOTOS specifications. Before we start doing these things, we will first examine the two specification notations, state-based and process algebraic, to find any similarities which we can use to build our translation.

3.4.1 Basic correspondences

We will first identify the most obvious and basic similarities. To illustrate these similarities, we will refer to two specifications, both describing a buffer object, but one written in our state-based notation and one written in LOTOS.

The state-based buffer is the same one we introduced in §3.1.1, apart from the changes of name of the specification and the operations:

statebuffer :

State space: $\mathbb{N} \cup \{\, VOID\}$
Initial states: $VOID$
Operation in $(\mathbb{N} \to nil)$: $before = VOID \wedge after = arg$
Operation out $(nil \to \mathbb{N})$: $before \neq VOID \wedge result = before$
$\wedge\ after = VOID$

Here is a LOTOS definition of a buffer. We claim that, at an intuitive level, it is "the same specification" as the state-based buffer, but written in a different notation:

$$lotosbuffer := in?x:\mathbb{N};\ \ out!x;\ \ lotosbuffer$$

We can immediately see that the specification of *lotosbuffer* is much more concise than that of *statebuffer*. This is due to the way the state-based notation requires that the object's state be made explicit in the specification, whereas the LOTOS specification represents state implicitly.

The most basic correspondence between a state-based specification and a LOTOS process specification is simply that the two correspond. A state-based specification describes an object with some internal state, and with a number of operations by means of which it can communicate with its environment. A LOTOS process also has internal state, although in this case it is represented implicitly in the specification, and it communicates with its environment by performing actions at gates. We will therefore take as our first correspondence that a state-based specification corresponds with a LOTOS process. (In Object-Z, we can treat each object definition as a miniature Z specification, so an Object-Z object also corresponds to a process.)

A state-based specification has a fixed number of operations, each of which is described by an "operation" clause in its specification. Each operation takes an argument and returns a result. There may therefore be an unlimited number of different possible interactions which can take place, but the number of operations is fixed when the specification is written. Similarly, a LOTOS process interacts with other processes via gates. Each action which occurs can carry data, giving an unlimited number of possible interactions, but the number of gates is fixed when the specification is written. *statebuffer* has two operations, *in* and *out*. Similarly, *lotosbuffer* can perform operations on two gates, also named *in* and *out*. We will take each state-based operation to correspond with a gate in a LOTOS process.

Now we have to look more closely at operations and gates. State-based operations take one argument and return one result each time they are invoked. LOTOS processes can pass data when they perform actions. LOTOS's "!" output operator and "?" input operator can be used to pass data into and out of LOTOS processes. In Full LOTOS, multiple data values can be passed in a single action prefix. For example, the process $x!3?y:\mathbb{N};\ B$ can evolve by performing an action such as x_3_0 or x_3_1. Could we, perhaps, use this mechanism to pass data, and thus have each state-based operation correspond to a single action? Let us examine operations and actions a little further before we answer that.

Operations are invoked zero or more times during the lifetime of the object. Each time it is invoked, an operation takes an argument, and returns a result. When a specification is implemented, an operation is likely to be implemented using a procedure call of some kind. Procedure calls take time to complete; the argument is definitely passed *before* the result is returned; the output value which is returned may depend on the argument value supplied.

If operations correspond to gates, then we would expect operation invocations to correspond to actions. However, actions are defined to be atomic, and occur at a single instant in time. Consider a state-based specification for an object which squares numbers:

squarer :

State space: *nil*
Initial states: *nil*
Operation square ($\mathbb{N} \to \mathbb{N}$): $result = arg \times arg$

How would we translate this into LOTOS, if we are using one action to correspond to one operation? We would want the translated LOTOS specification to behave similarly to the following:

$$\begin{aligned} squarer := {} & square\_0\_0;\ squarer \ [\,]\ square\_1\_1;\ squarer \\ & [\,]\ square\_2\_4;\ squarer\ [\,]\ square\_3\_9;\ squarer\ [\,]\ \ldots \end{aligned}$$

The obvious LOTOS encoding of this is as follows:

$$squarer := square?x:\mathbb{N}!x^2;\ squarer$$

Unfortunately, it is not valid LOTOS because LOTOS's scoping rules ensure that the two xs cannot refer to the same variable.

The process synchronising with this could "call" the *square* "operation" like this:

$$square!3?x:\mathbb{N};\ B$$

For this to work, the process would have to offer exactly that set of actions which correspond to possible call-and-return sequences of the original operation. This is difficult to arrange, and somewhat counterintuitive, so we will instead use two actions to translate each operation invocation. We will consider this issue in more detail in the next subsection.

Both state-based and process algebraic specifications have a state space. A state-based specification has an explicit statement of which states the object can take. In our state-based notation, this is given explicitly as a set. In other notations like Z, variables are defined; the state space of the process is the cross product of the possible states of each variable. In a process algebraic specification, the states are not made explicit; rather, there is a state corresponding to each possible behaviour which the process can evolve to. Since we are intending to translate state-based specifications into LOTOS, each state in the state space of the original specification will translate into a behaviour in the LOTOS translation. Problems will arise if the original state space is infinite, since this would lead to an infinite LOTOS specification. We note that these problems exist, but we do not address them in this dissertation. Instead, we will avoid using specifications with infinite state spaces.

Both styles of specification can also be thought of as having an interface, in the sense that a class in an object oriented programming language has a defined interface. In our state-based notation, this is made explicit; all the operations the object can respond to are listed in the specification, together with their argument and result types. In a LOTOS behaviour, the gates it responds to are not explicitly listed, but they can be calculated easily; they are denoted $\mathcal{L}$.

Finally, we note that neither state-based nor process algebraic specifications specify what an object *actually does*. Instead, they specify what the object can potentially do. The object needs an environment to be embedded in. State-based specifications describe objects which can respond to operation invocations; some other object must do the invoking. LOTOS specifications describe processes which can interact at gates; but which specific interactions are performed are not known until we know which gates the environment interacts with, and what data values it passes.

3.4.2 Nondeterministic output

One of the most significant differences between notations is the issue of *blocking* versus *non-blocking* value passing. In the non-blocking scheme, the values passed are controlled by exactly one party to the interaction, and are simply passed to the other party, who has no influence over that value. The value passed need not be deterministic: the controlling party may pick a value nondeterministically, but the receiving party has no control over which value it receives. The blocking scheme, however, allows both parties to control the parameter: each party may permit the use of a particular set of values, and the value which is actually passed must be in both sets. If the two sets do not intersect, the interaction blocks indefinitely, and cannot occur, possibly leading to a deadlock. Intuitively, the parties may only interact if they can agree on a value which is acceptable to both.

These two schemes for handling data passing are based on a lower-level notion of blocking and non-blocking behaviour. In a process algebra such as LOTOS, each action which a process performs consists of a name and zero or more data values. When the process interacts with another process with which it is composed in parallel, it can, at every stage, either perform an action which the other process offers (perhaps after performing an internal action), or refuse all the actions offered. By refusing actions, our process can block the progress of the other process or influence the future actions which the other process will perform.

In a state-based language such as Z, under its conventional interpretation, an operation may be invoked with input values, and may produce output values. The inputs are generally understood to be controlled by the environment and the outputs are determined by the operation predicate. There is no notion of the object being able to block its environment by refusing certain inputs; instead, the effect of invoking the operation is only well-defined in the case where the operation's precondition is true.

In both of these cases, we can see that the data value being passed is not the only concern when deciding whether the action is refused, or whether the operation's effect is well-defined. Rather, it is the combination of the action name and all the data values, or the operation name and all its inputs, which is crucial; all these items are considered together. In a situation where refusals are permitted, (that is, in a blocking language) there is no practical difference between an operation which refuses any and all inputs and an operation which is, itself, refused. Thus, we can see that the ability to block data values is merely a special case of the ability to block any interaction.

In this section, we will review how data passing is treated in a few different notations.

LOTOS

Data passing in LOTOS is done by use of the "!" and "?" operators. To receive a data value associated with an action, the receiving process uses a construct of the form "$a?x : T;\ B$", where a is the action name, T is the type of data being received, x is the variable name to which the value will be assigned, and B is the behaviour to which the process will evolve. The semantics associated with this input action is described in full in appendix A, but in short, the behaviour "$a?x : T;\ B$" offers its environment a choice of a number of actions of the form a_v, one for each value v of type T. When one of these actions is performed, the particular v chosen is substituted for the variable name x in B, and the behaviour evolves to the resulting substituted B.

Outputting data in LOTOS is done with an action prefix of the form "$a!x;\ B$". In this case, x is the value to be passed, and the behaviour offers the single action a_x. This can synchronise with the corresponding action a_v offered by "$a?x : T;\ B$", where $v = x$.

The effect of this is that, although the syntax of LOTOS distinguishes between input (using "?") and output (using "!"), the semantics does not. Both input and output are implemented using action names incorporating the appropriate data value in the same way; there is no restriction against two input actions synchronising together, or two output actions.

Thus, at the semantics level, there is no indication of the direction the data is being passed. We can illustrate this using the following example processes:

$P := a!3;\ stop$
$Q := a!3;\ stop\ []\ a!5;\ stop$
$R := a?n : \mathbb{N};\ stop$
$S := a!1;\ stop\ []\ a!3;\ stop$

(The syntax of the LOTOS constructs we use in this dissertation are defined in §3.3.)

We can place P in parallel with R as follows: $P \| R$. This behaviour can perform only the action a_3, as we would expect. $Q \| R$ can perform either a_3 or a_5, which is also as we would expect, since these cases involve placing an input action in parallel with an output action.

It is also permissible to write $P \| P$, a construct which can only perform a_3, or $Q \| Q$, which can perform either a_3 or a_5. Significantly, $Q \| S$ can perform only one action, a_3. Q on its own is capable of performing a_5, and S can perform a_1, but in parallel, each of these actions are blocked by the other process's inability to perform the corresponding action.

This shows that LOTOS uses the blocking model for value passing, and also shows why "blocking" is named so. Although the value output by a process would normally be expected to be determined internally by that process, possibly based on previous inputs, the example of $Q \| S$ shows that, in LOTOS, the choice of output value can be determined from outside the process performing the output. This is a characteristic which we will find to cause us problems later in this dissertation.

Input/output automata

Input/output automata[LT89, Seg92] are a form of behavioural specification which inherently distinguishes between input and output actions.

Each I/O automaton has an interface: a set of actions which it can engage in. The actions in an automaton's interface are classified into three sets: a set of input actions, a set of output actions and a set of internal actions. The automaton can perform any of its internal actions at any time without interacting with its environment; these actions are not externally visible. The automaton can perform any of its output actions at any time; these are visible to the environment, although the environment cannot block these actions. The environment can perform any of the automaton's input actions at any time, and the automaton cannot block these actions.

As an example, let us define a simple I/O automaton with behaviour similar to that of the following LOTOS process:

$P := get?p : bool;\ put!p;\ P$

where *bool* is a type containing two values, *true* and *false*. We use *bool* here to make the sets of actions and transitions finite, for ease of presentation; I/O automata do not have to be finite in this way, however.

Firstly, we will identify the actions our automaton can engage in: $get(true)$, $get(false)$, $put(true)$ and $put(false)$. $get(true)$ and $get(false)$ comprise the automaton's "input signature"; the set of actions it cannot block. $put(true)$ and $put(false)$ will comprise the "output signature"; the set of actions the automaton controls. This automaton has no internal actions.

The specification of an I/O automaton gives a set of transitions defining when particular actions may occur, and what their outcomes are. This set is usually specified by giving preconditions which state when any given internal or output action may be performed, and "effect" clauses specifying the outcome of any action (including input actions). I/O automata cannot have a precondition on their input actions, because they cannot refuse or block an input action.

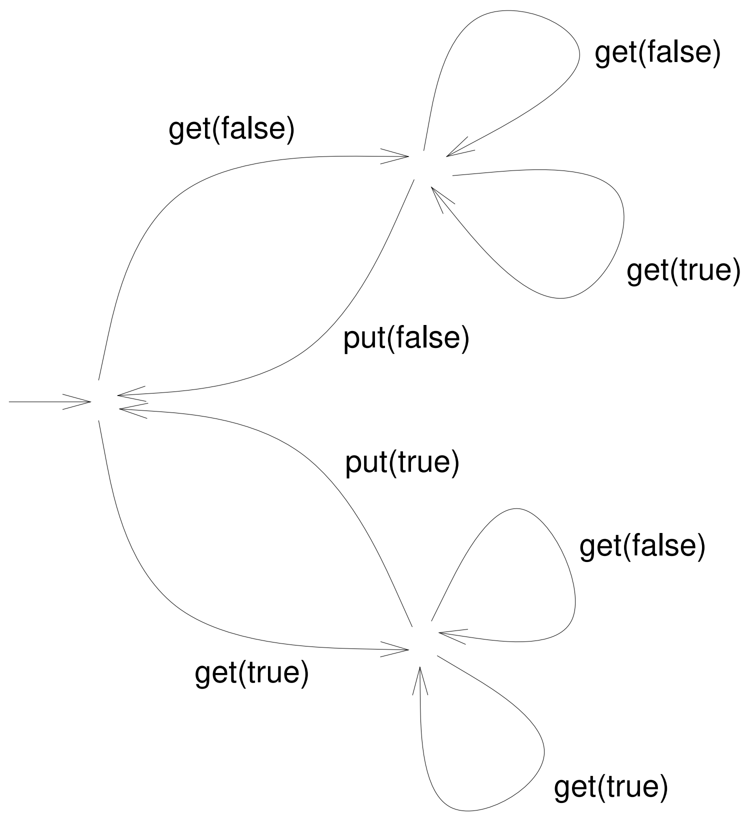

Figure 9: Transition system for an I/O automaton

When represented as a transition system, an I/O automaton must therefore have transitions corresponding to every possible input action leading from every state.

We can complete our example I/O automaton by producing the transition system shown in figure 9. Because the actions *get*(*true*) and *get*(*false*) are in the automaton's input signature, they may occur at any time; thus, we have had to decide what the outcome will be if they occur in situations where the LOTOS behaviour would have blocked them. In this case, we have decided to ignore such actions.

We can see from figure 9 that there are arcs labelled *get*(*true*) and *get*(*false*) leading from every state; this ensures that input actions cannot be blocked, and is required by the definition of I/O automata.

When we define our translation, we will require a similar effect: since the environment may call any operation of a state based specification at any time, and the state based specification cannot block such a call, we must ensure that the behavioural specification which results from our translation should similarly not be able to block any input from its environment. We will devote some effort to this problem later in the thesis.

$\mathbf{CSP}_{78}$

Hoare[Hoa78] describes an early version of the CSP notation, which we will refer to as CSP_{78}. In section 4 of his description, Hoare outlines a method of performing inter-process communication using data-carrying actions to mimic the behaviour of procedure calls. The method simply involves an action at the start of the procedure carrying the arguments passed to the procedure, and an action at the end of the procedure carrying any return values; this is now regarded as a standard method of performing remote procedure calls.

We can write a LOTOS procedure in the style of CSP_{78} as follows:

$$P := x?a : T;\ \ldots;\ x!r;\ P$$

where a is the procedure's argument, which is of type T, r is the return value, and "..." represents the processing which takes place while the procedure executes. The procedure can be called by a process performing the corresponding actions, such as:

$$Q := x!7;\ x?r : \mathbb{N};\ \mathit{stop}$$

In CSP_{78}, input and output commands refer directly to the named process they are to interact with, so $x!7$ sends the value 7 to a process named x and $y?n$ receives a value from the process named y. Recent notations, such as LOTOS and modern CSP, use action names instead of naming the corresponding process; in these notations it becomes natural to use the procedure name as the action name, allowing a single process to provide several different procedures.

What is interesting about the method described by Hoare is that it encodes a normal imperative style procedure-call in a process algebra. We will later use this idea of using a pair of actions to represent a single procedure call when we build our translation; each (state based) operation invocation will, when translated into a process algebraic specification, be represented by two actions, one representing the procedure's start and the other representing its finish.

CSP-OZ

The state based approach to specification is closely related to the imperative programming model, so it is not surprising to find that the non-blocking model is widespread; in particular, it is used by Z, and by the Liskov and Wing notation. Object-Z, by contrast, uses the blocking approach.

However, CSP-OZ[Fis97] is interesting because it allows both approaches. An undecorated parameter name in a CSP-OZ schema is taken to be a blocking parameter in the normal style of Object-Z, which can be controlled by both the object and its environment. A parameter decorated with "?" is an input parameter and is controlled solely by the environment, while a parameter decorated with "!" is an output parameter and is controlled solely by the object in question.

"?" and "!" parameters thus behave in the manner of I/O automaton input and output actions, while undecorated parameters behave like ordinary CSP actions. In this way, CSP-OZ avoids mandating one or other of these approaches.

3.4.3 Two actions per operation; separate call and return

Since operations do not take place instantaneously, but take some time to complete, it seems natural to model them by two actions, one representing the *call*—the transfer of control into the operation's code and the passing of parameters—and one representing the *return*, where the flow of control returns to the caller and the return value is passed back. This was suggested by Hoare in [Hoa78], which we reviewed in §3.4.2. We will now consider in detail how we can apply this technique.

The most obvious method is as follows. Consider again the above example specification, *squarer*. We will now rewrite *squarer* in LOTOS, but using two actions to correspond with each operation:

$$\mathit{squarer} := \mathit{square}?x : \mathbb{N};\ \mathit{square}!x^2;\ \mathit{squarer}$$

This has the obvious advantage over the LOTOS *squarer* given in the previous subsection that it is possible to write down the complete specification. The previous behaviour specification was infinite, and could not be written in standard LOTOS. Of course, the above *squarer* process also has infinite behaviour, but the specification can be written down.

A *squarer* process's *square* operation could be called like this:

$$square!3;\ square?x:\mathbb{N};\ B$$

After the two interactions on the *square* gate, the value 9 would be substituted for x throughout B.

As another example, let us consider a process to take square roots:

rootfinder :

State space: *nil*
Initial states: *nil*
Operation root ($\mathbb{N} \to \mathbb{Z}$): $arg = result^2$

This specification exhibits nondeterminism (since $x^2 = (-x)^2$), and has a precondition which is not always true (since relatively few natural numbers have integer square roots). In LOTOS, we could represent *rootfinder* like this:

$$p = \{n:\mathbb{N} \mid \exists\, m:\mathbb{N} \bullet n = m^2\}$$

$$rootfinder := root?x:p;\ (root!\sqrt{x};\ rootfinder\ []\ root! - \sqrt{x};\ rootfinder)$$

This appears to do what we want, although there are some problems with this method of representing operations.

- Operations are no longer atomic. This is not, in itself, a problem, since this was the reason we wanted to use two actions to represent each operation invocation. However, for this method to work, it is necessary for the caller of an operation to be always at the same stage of the interaction, either before the first action of an operation or between the two actions of an operation, as the process it will call. In particular, it may be possible for the caller's first action of an operation invocation to synchronise with the called process's second action, or vice versa. If all of the processes synchronising on a single gate perform actions in pairs as described above, then they cannot get out of synchronisation. If, however, one or more processes do not follow the protocol, or if the process algebra being used allows some of the processes synchronised on a single gate to interact while not involving other processes on that gate, we may get meaningless results. If this problem arises in practice, it should be possible to solve it by changing one of the actions, so that the set of actions used for the "call" part of the operation is disjoint from the set of actions used for the "return" part. For example, "call" actions could be of the form *name_CALL_arg*, while "return" actions could be of the form *name_RETURN_ret*, where *name* is the operation name, *arg* is the argument value, and *ret* is the return value.

- A process might deadlock after performing the first action of a pair, preventing any processes it is synchronised with from progressing. This is not so much a problem with the formal system as a reflection of a problem that occurs in real systems. If, in a procedural system, a procedure is called which then does not return (for instance, because it has entered an infinite loop), the process as a whole will not progress. Similar problems can occur in most implementation languages. The fact that one process may deadlock and thus prevent other processes from progressing is therefore a realistic feature. If the output of our translation can also exhibit this kind of deadlock, this gives us the possibility of checking for it and attempting to prove its absence using the normal process algebra tools and methods. This kind of deadlock would therefore be a desirable feature for our translation to have; it makes our formal model more true to life.

- Only two process may usefully synchronise on a single gate. Since all processes synchronised in parallel on a single gate must perform actions on that gate together, it is probably not useful to synchronise more than two processes which are following this protocol. All the processes taking the "caller" side of the protocol would have to pass the same argument and receive the same return value, and all the processes taking the other side of the protocol would have to have exactly the same observable behaviour as each other. It would not be possible to have several "client" processes communicate with a single "server" process on a single gate; the server would have to be written with a separate gate for each client. In this dissertation, we will only be considering pairs of processes, so this problem will not arise in our case.

- The process calling an operation may be able to influence the return value it receives. For example, we could call the *root* operation of the *rootfinder* process above as follows:

  $$root!3;\ root?x:\mathbb{N};\ \mathrm{B}$$

  (In fact, LOTOS's type checking prevents us writing this, but if subtyping of data were included in the language a similar situation would arise.) This process, after performing the first action, *root*_3, can perform only an action of the form *root*_*x* where x is a natural number. Since this process is synchronised with *rootfinder*, the value that *rootfinder* returns will be constrained to be positive. This goes against our understanding of real systems, where the operation decides what the return value will be; the caller cannot influence this decision. We can "fix" this in the *rootfinder* specification above by adding some internal actions, like this:

  $$rootfinder := root?x:p;\ (i;\ root!\sqrt{x};\ rootfinder\ []\ i;\ root!-\sqrt{x};\ rootfinder)$$

  In a later chapter, when we define our translation formally, we will use a similar solution for this problem.

- There is no guarantee that the return values the caller is able to accept are sufficient to receive any value the operation is able to return. This could either cause a nondeterministic operation to return a different value, as discussed in the previous item, or it could result in a deadlock if the caller cannot receive any of the values that the operation is prepared to return. We have considered a method of fixing this by encoding the return type in the first action of the operation, thus preventing the operation starting unless the caller can handle the return type. However, this turned out to be inflexible and unnecessary; this situation should be prevented by using subtyping and type checking to ensure that an operation is never called unless the caller can receive any value of its return type. We will not discuss this option further here.

When we define our translation from state-based to LOTOS specifications in a later chapter, we will use a refined representation of operations, based on the method given above. At that time, we will also solve some of the problems listed above.

3.5 Conclusion

In this chapter, we have presented a simple state-based specification notation, based on Liskov and Wing's notation [LW94].

We presented the Z refinement relation, and demonstrated how it can be applied to specifications written in our state-based notation. Since the Z refinement relation is very similar to the downward simulation relation, and to the core conditions of Liskov and Wing's "Behavioural Subtyping" relation [LW94], we believe that it is suitable for use as a subtyping relation over state-based specifications.

We briefly introduced a subset of the LOTOS specification notation which we will use for writing process algebraic specifications.

LOTOS is the language we intend to obtain a subtyping relation for, by means of translating the Z refinement relation to it. We therefore compared the two languages, finding some similarities and correspondences between the state-based and process algebraic notations which will give a basis upon which we will construct our translation.

In this thesis, we will be primarily interested in the challenge of translating from a non-blocking language (our variant of Liskov and Wing's notation[LW94]) to a blocking language (LOTOS). We note that, in situations where the languages to be used are not determined by other factors, it may be possible to ease the task of translation by choosing a state based notation and a process algebra which are either both blocking or both non-blocking. This would result in a much more straightforward translation than the one we will produce. We, however, hope to gain some insight by examining the issues which arise when translating between two languages which are not well-matched.

In the next chapter, we will examine some of the many standard relations between LOTOS processes, and evaluate them as regards their suitability for use as subtyping relations.

Chapter 4

Representing unspecified behaviour in LOTOS

We have defined the notion of subtyping as that of substitutability. Most current object oriented programming languages use a signature-based subtyping check, which compares the names, argument types and return values of operations. We are instead interested in behavioural subtyping, which compares the behaviours of operations: in particular, the order in which operations may be invoked.

In chapter 2, we examined the behavioural subtyping relation proposed by Liskov and Wing[LW94], and noted that the main conditions of their relation are essentially the same as the downward simulation relation and the Z refinement relation. We decided to use the Z refinement relation as an appropriate subtyping relation for state-based specification languages.

Since our goal is to produce a behavioural subtyping relation for specifications written in process algebras, our next task is to examine the existing relations defined on process algebraic specifications to determine whether an appropriate relation already exists.

A wide range of development relations have been defined for the LOTOS specification language. If a process algebraic behavioural subtyping relation already exists, it seems likely that one of LOTOS's development relations will embody its conditions.

In this chapter, we will introduce a number of standard relations defined between LOTOS processes, known as the LOTOS development relations. Of these relations, we will show that one, called reduction, closely matches our intuitive understanding of the meaning of subtyping.

We will examine how we might use the reduction relation as a subtyping relation. We discover that there is a substantial difference between the way our state-based notation and LOTOS each treat unspecified behaviour. Due to this, and the idea that subtyping means substitutability, translating a pair of state-based specifications related by subtypes into LOTOS using the simple translation suggested in §3.4 will not produce LOTOS specifications related by subtyping. Instead, we will find that we have to modify our translation to account for the difference in the treatment of unspecified behaviour.

4.1 Translation

In order to prove that a relation in one formal notation corresponds to a relation in another, it is necessary to use some form of translation[BDLS95, DBBS99]. By translation, we mean a relation (often a function) from one specification notation to another, where related specifications are considered to have the same meaning. (We do not *require* a translation to be a function, although it will almost always be so. It is conceivable that a specification might be validly translated two different ways—the two resulting specifications might be equivalent in the target language, or might differ only in aspects which are not expressible or significant in the original notation.)

Specifications in different notations are not directly comparable, but translating both into a common notation provides a means of comparison. If one of the notations being used can be translated to the second, the second notation can act as the common notation. It may be, however, that neither notation can express all specifications expressible in the other; in these cases, a third notation is needed, to which both of the other notations can be translated.

Translations can be performed on any of several different levels. A syntax-based translation can be used, which translates terms and operators in one notation directly into equivalents in another notation. In some cases, such a direct mapping may not exist, perhaps because the target language is based on a different set of primitives than the source language. In these cases, a more indirect approach must be used. One such approach involves translating indirectly via a third language. A third approach consists of "flattening" (or "expanding out") the source specification into a form which is easily translatable into the destination notation; essentially, any terms which cannot easily be translated are first transformed into terms which can. We will now consider each of these forms of translation in more detail.

4.1.1 Direct syntax-based translation

Syntax-based translation is a very superficial method; it seems likely to give good results only in cases where the source and destination language are based on the same model and concepts, such as would be the case translating, for example, between two process algebras, or between two Algol-based imperative programming languages. Because of its superficial nature, a syntax-based translation can preserve much of the structure of the original specification, carrying it through into the target language.

Derrick et al.[DBBS99] describe a syntax-based translation from LOTOS to Object-Z. This translation can be successful because structures or combinations of structures in LOTOS can be identified with structures in Object-Z. Data types in the full LOTOS language are specified using a separate language called ACT-ONE; these ACT-ONE data types are translated to Object-Z data types. LOTOS behaviours are then translated to Object-Z class schemas. Consider, for example, the following LOTOS behaviour:

$$B := a;\ b;\ stop$$

This translates to the following Object-Z class (after some simplification):

B

$s : s_0 \mid s_1 \mid s_2$

INIT

$s' = s_0$

a

Δs

$s = s_0 \wedge s' = s_1$

b

Δs

$s = s_1 \wedge s' = s_2$

From this very simple example, it can be seen how the sequencing obtained using action prefix in LOTOS is reproduced using an explicit state variable in Object-Z. The translation is obtained by "unravelling" the LOTOS behaviour and translating it from the inside out. First, "*stop*" is translated into an Object-Z class which has no operations and therefore does nothing. Next,

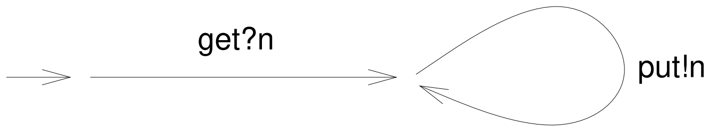

Figure 10: Extended transition system corresponding to a LOTOS process

"b; *stop*" is translated by augmenting the state of the translation of *stop* and adding the operation b. Finally, an operation a is added, together with more state, resulting in the translation of "a; b; *stop*" above.

Because of the directness of syntax-based translation, much of the structure of the original specification can be preserved in the translated specification. For example, [DBBS99] gives a translation of the following LOTOS behaviour:

$$a?x : \mathbb{N};\; b?y : \mathbb{N};\; c!largest(x, y);\; stop$$

In that translation, the variables x and y are translated into Object-Z variables named x and y. The *Max*3 example in [DBBS99] shows that LOTOS process specifications are translated into Object-Z classes, LOTOS process instantiations are translated into Object-Z object instances, and LOTOS data types (specified in ACT-ONE) are translated into Object-Z data types. Thus much of the high-level structure of the original specification is carried forward through the translation. Other, more indirect, styles of translation may not result in structures such as these retaining any recognisable identity in the translated specification.

Because a syntax-based translation must translate identifiable structures in the source notation into similar structures in the destination notation, this method cannot be used in cases where the languages to be translated between are radically different. If appropriate points of correspondence between the two languages cannot be found, a different kind of translation may have to be used.

4.1.2 Indirect translation through a third notation

If it proves difficult to translate directly from one notation to a second, it can be helpful to perform the translation in stages.

Derrick et al.[DBBS96] describe such a translation, which translates LOTOS specifications via extended transition systems[WC90] to ZEST[CR92] and thence Z. We can illustrate this by translating the following LOTOS specification:

$$Box := get?n : \mathbb{N};\; Box'(n)$$

$$Box'(n : \mathbb{N}) := put!n;\; Box'(n)$$

The LOTOS process *Box* receives a single data value via a *get* action, then outputs that value repeatedly via *put* actions.

The first stage of the translation is to create an ETS (extended transition system) corresponding to the original LOTOS process. The method used in [DBBS96] produces an ETS for *Box* similar to that shown in figure 10. We can see that this step has retained some of the structure of the original specification (the fact that the variable is named n, for example), but has lost some information (for example, the distinction between the definitions of *Box* and *Box*$'$. However, the effect is not nearly as pronounced as it would be using a flattening translation (which we will discuss in the next subsection), the result of which can be seen in figure 11.

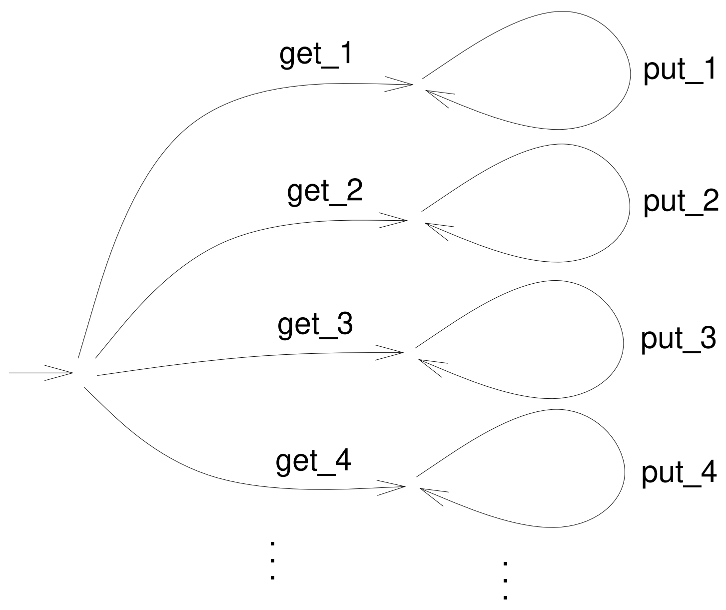

Figure 11: Flattened translation of a Z specification

The next step in our example is to translate the ETS into ZEST. Finally, the ZEST specification can be mapped down to Z to complete the translation. This results in a specification similar to that given in the next subsection.

This translation is not as straightforward as the syntax-based translation we discussed previously. Although each step in the translation looses little information, the cumulative effect of several translation steps results in the translated specification being less syntactically faithful to the original than is the case when performing a direct syntax-based translation. The indirect method, however, has the advantage that it may be used in cases where no syntax-based translation is apparent.

In chapter 5 of this thesis, we use elements of this kind of approach by defining variants of LOTOS with modified output semantics. These modified semantics contain the kind of features which could be used in a translation of a state based specification. The resulting modified LOTOS specifications could then be translated to labelled transition systems, or, with appropriate rules, could be translated to standard LOTOS. However, we do not actually define a translation which makes use of these modified languages; instead, we implement the features we need within our (single-stage) translation.

4.1.3 Flattening translation

In the section entitled “Interpreting state based techniques with behavioural semantics” of [BD99], an example is presented of the translation of a (state based) Object-Z specification into a (behavioural) labelled transition system. This is an example of a flattening translation: the state space of the labelled transition system is obtained by enumerating the state space of the Object-Z specification, and the transitions are generated by considering the pre- and postconditions of each operation in each state and adding transitions depending on their truth values.

For example, a Z specification such as the following might be translated by a flattening translation similar to that given in [BD99] into the labelled transition system shown in figure 11.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Box}} \\ \hline
f : \textit{boolean} \\
n : \mathbb{N} \\ \hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Init}} \\ \hline
\textit{Box} \\ \hline
f = \textit{false} \\ \hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{get}} \\ \hline
\Delta \textit{Box} \\
\textit{num}? : \mathbb{N} \\ \hline
f = \textit{false} \\
f' = \textit{true} \\
n' = \textit{num}? \\ \hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{put}} \\ \hline
\Delta \textit{Box} \\
\textit{num}! : \mathbb{N} \\ \hline
f = f' = \textit{true} \\
\textit{num}! = n = n' \\ \hline
\end{array}
$$

This specification defines the behaviour of a *Box*, an object which stores a single number via a *get* operation, then provides that number in response to any *put* operation. A (hypothetical) non-flattening LOTOS translation of *Box* might be as follows:

$$\textit{Box} := \textit{get}?n : \mathbb{N};\ \textit{Box}'(n)$$

$$\textit{Box}'(n : \mathbb{N}) := \textit{put}!n;\ \textit{Box}'(n)$$

In this non-flattening translation, we can see the original structure of the specification carried through to the LOTOS version; in the flattening translation, the identity of many of the components of the original specification have been lost.

Flattening translation clearly looses a considerable amount of the structure of the original specification, but can bridge a larger gap between the semantics of the original specification and that of the target notation.

The translation we will develop in this thesis is a flattening translation.

4.2 Mapping state-based specifications to process algebra

Before we can decide whether a particular relation corresponds to the state-based subtyping relation, we must decide what "correspondence" means. As we outlined in §1.5, we define it in terms of a translation from state-based specifications to LOTOS. We will consider a refinement relation on LOTOS processes to correspond to the state-based subtyping relation if, exactly when a state-based specification A is a subtype of another state-based specification B, the LOTOS translation of A is a refinement of the LOTOS translation of B, using the refinement relation we are testing.

As such, whenever we want to test the suitability of a LOTOS relation as a subtyping relation, we will have to examine it in conjunction with a translation. In this section, we will define the straightforward translation we will use in this chapter. This translation will be sufficient to see which of the LOTOS development relations would be suitable as a subtyping relation. In later chapters, we will modify the translation until our chosen relation corresponds to the state-based subtyping relation.

We will define our translation rather informally at this stage, and we will translate our state-based specifications into labelled transition systems instead of into LOTOS process definitions. Since the semantics of LOTOS processes are defined in terms of labelled transition systems generated from the process definitions, this will suffice at present.

4.2.1 A simple translation

We now define a very simple translation from our state-based specification notation into labelled transition systems. This translation does not carry over either arguments or return values into the labelled transition system generated, and we will show, with an example, that this will have to be remedied. We will improve this translation in the next chapter.

Let us show how this translation works with an example. We will translate the "stack" specification given in §3.1.2.

Each state in the original specification is mapped to a state in the labelled transition system. The stack specification has the following set of states: $\{[\,], [0], [0, 0], [0, 1], [1, 0], [1], [1, 1]\}$. This means that our labelled transition system will have seven states.

We then examine the operation definitions to find out which operations can be performed in which before-states. For a given operation and a given before-state, we examine the operation's predicate. We find all the after-states for which there is some combination of argument and result value which satisfies the predicate. Then, for each of these after-states, we add a transition to our labelled transition system leading from the before-state to the after-state and label it with the operation name.

Considering the initial state of our stack specification, we can see that *get* cannot be performed, since the condition $before \neq [\,]$ is untrue. *put*, however, can be performed, either with argument 0 and result *nil*, or with argument 1 and result *nil*. If its argument is 0, the after-state will be $[0]$; if the argument is 1, the after-state is $[1]$. We therefore add two transitions labelled *put* to our labelled transition system; one from $[\,]$ to $[0]$, and from $[\,]$ to $[1]$.

The result of doing this for each state in our stack specification is shown in figure 12. The dotted arrows show which states in the labelled transition system correspond to which state in the original state-based stack specification.

We can translate the bag and queue specifications too, and get the labelled transition systems shown in figures 13 and 14 respectively.

Despite that each of these diagrams looks slightly different, our translations of each of the bag, stack and queue specifications are, in fact, bisimilar with one another. (We define bisimulation in §4.3.7.) Since bisimulation is a stronger relation than any other we will use, this means that, for our purposes, these three labelled transition systems are equivalent. They are all also bisimilar to the labelled transition system in figure 15.

From this, we can see that the translation given above is not suitable for our purposes, since it does not translate these specifications into distinguishable labelled transition system. However, this translation is appropriate for the examples we will look at in this chapter. We will present several progressively more refined translations later, as we determine what features our translation will need.

4.3 Initial attempts to model subtyping

We will now examine some of the common LOTOS development relations and analyse them for similarity with our intuitive idea of subtyping as substitutability.

We will first consider the trace-based development relations: trace preorder, trace extension and trace equality. We will then look at some development relations based on traces and refusals:

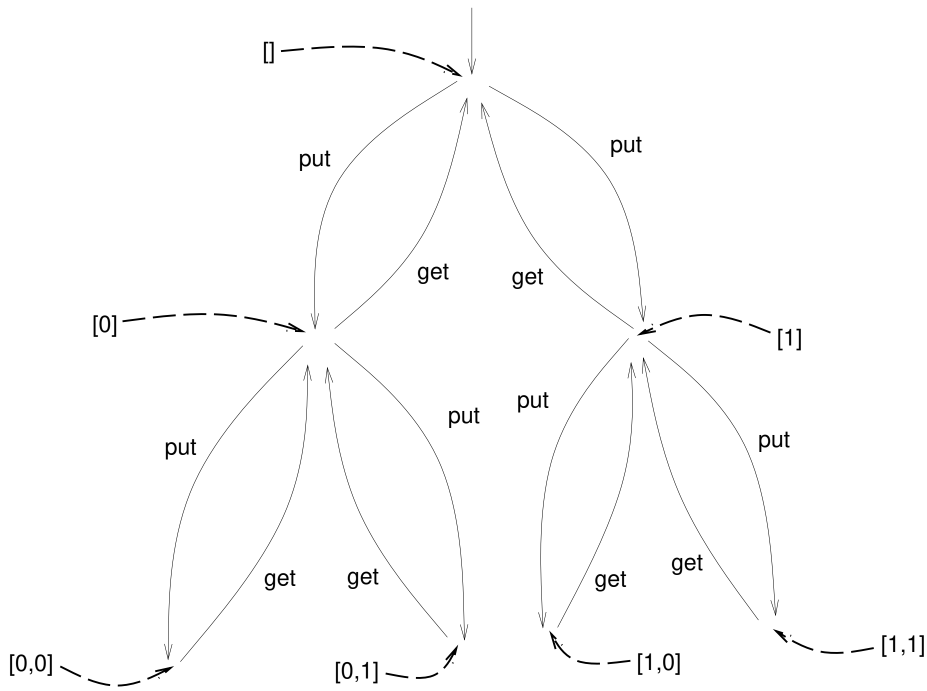

Figure 12: Translation of the stack specification

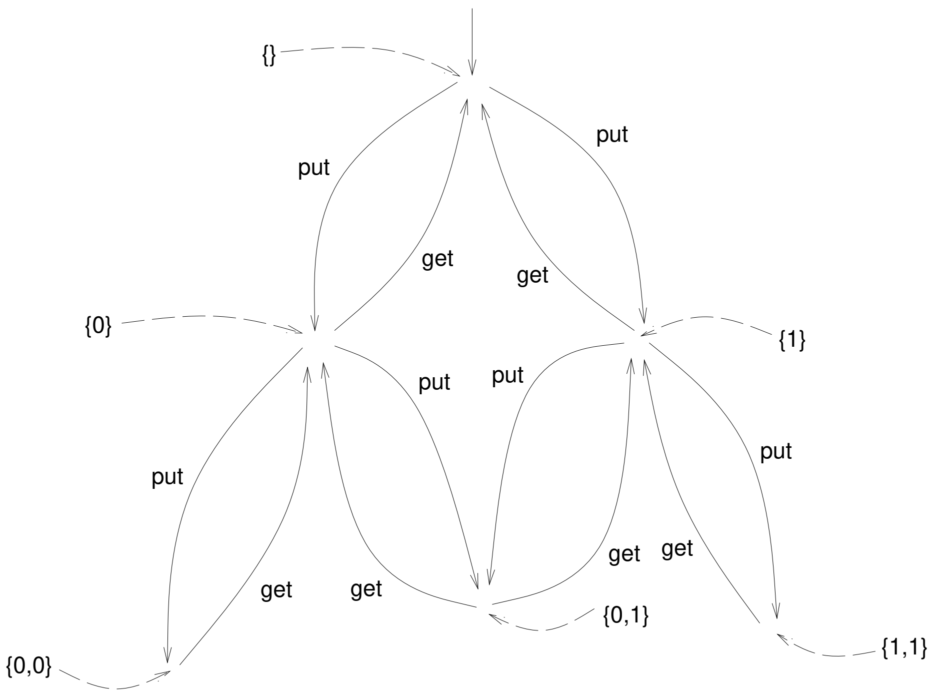

Figure 13: Translation of the bag specification

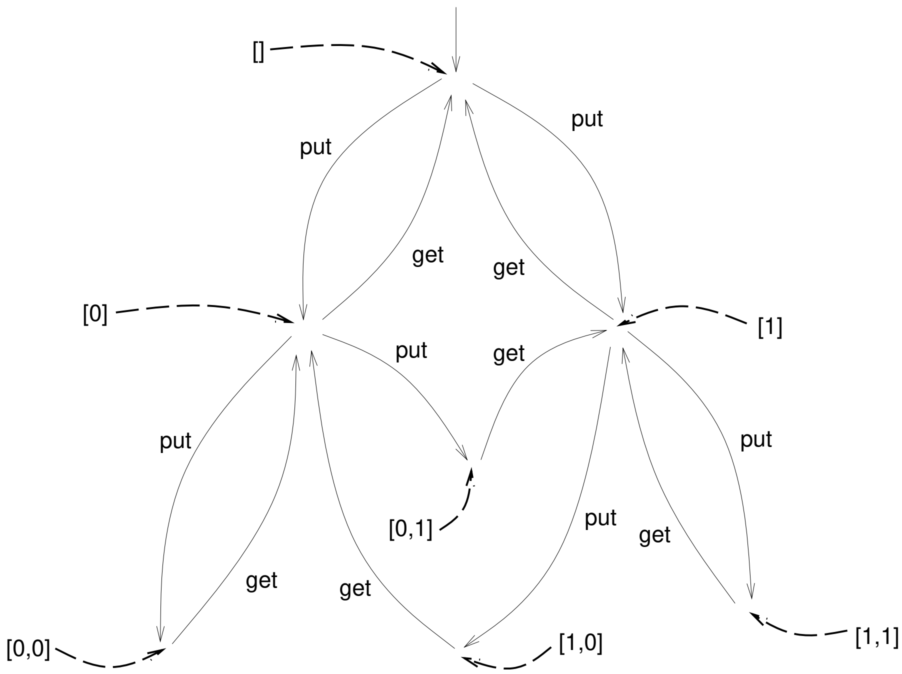

Figure 14: Translation of the queue specification

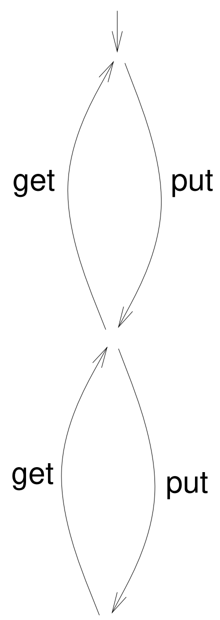

Figure 15: Simplified labelled transition system which is bisimilar to the translations of the bag, stack and queue specifications

testing equivalence, extension and reduction. We will also examine the bisimulation equivalence in two forms, strong bisimulation and weak bisimulation.

This analysis will give us a good idea of which relations might be suitable for use as a subtyping relation.

Recall first the characteristics of the behavioural subtyping relation. If, in a particular state, the original specification is able to perform an operation with a given argument, then any subtype of that specification must be able to perform the same operation with the same argument in the corresponding state. When the subtype has performed the operation, both the result which it returns and the state in which it finishes must correspond to a result and a finishing state which could have resulted from the same operation in the original specification.

4.3.1 Trace preorder

We first consider trace preorder, which is one of the simplest refinement relations defined on LOTOS specifications. It is based purely on trace semantics, where processes are described and identified in terms of the traces they can perform.

Trace preorder is written $\leq_{tr}$. The relation holds between two processes, P and Q, exactly if P has no more traces than Q. In other words, if there is a trace in $\mathrm{Tr}(P)$ which is not in $\mathrm{Tr}(Q)$, then $P \leq_{tr} Q$ does not hold.

Trace preorder is defined as follows:

$$P \leq_{tr} Q \Leftrightarrow \mathrm{Tr}(P) \subseteq \mathrm{Tr}(Q)$$

(Traces and Tr() are defined in §A.12.)

To restate this relation in terms of subtyping, the trace preorder requires that any sequence of operations that the subtype P can perform, its supertype Q must also be able to perform.

This relation does not allow P to have any more traces than Q. This is not what we want, since our intuitive understanding of subtyping suggests that a subtype should be able to define operations which its supertype did not, and so provide more specialised functionality than its supertype.

In addition, and more seriously, the trace preorder does not require that the subtype be able to perform all the traces that its supertype can perform. For example, let us define P and Q as follows:

$$P := a;\ \mathit{stop}$$
$$Q := a;\ \mathit{stop}\ []\ b;\ \mathit{stop}$$

With these definitions, $\mathrm{Tr}(P) = \{\langle\,\rangle, \langle a\rangle\}$ and $\mathrm{Tr}(Q) = \{\langle\,\rangle, \langle a\rangle, \langle b\rangle\}$. It is clearly true that $P \leq_{tr} Q$, but it is also clear that we would not consider P to be a subtype of Q, because P cannot perform, and indeed refuses, the action b which Q can perform.

4.3.2 Trace extension

Since trace preorder is not useful to us because of its failure to require a refined process to exhibit all the traces that the original process did, we will now consider trace extension as a candidate for our subtyping relation.

Trace extension is written $\leq_{tre}$. It holds between processes P and Q exactly if P has no fewer traces than Q. That is, if there is a trace which is in $\mathrm{Tr}(Q)$, but which is not in $\mathrm{Tr}(P)$, then $P \leq_{tre} Q$ does not hold.

The definition of trace extension is

$$P \leq_{tre} Q \Leftrightarrow \mathrm{Tr}(P) \supseteq \mathrm{Tr}(Q)$$

Trace extension satisfies our primary objection to trace preorder, which was that trace preorder did not require that a refined process must be able to perform all the traces that the

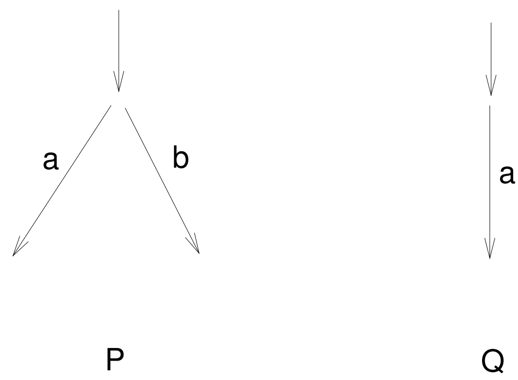

Figure 16: Labelled transition systems for processes P and Q

original process could perform. This is, in fact, exactly what trace extension does require.

As a subtyping relation, trace extension requires that the subtype is able to perform any trace which its supertype can perform. It also allows the subtype to perform traces which the supertype could not perform, which corresponds well with our experience with type checkers in object oriented programming languages, which invariably allow subtypes to define operations which their supertypes did not define.

For example, let us define P and Q as follows:

$P := a;\ \mathit{stop}\ []\ b;\ \mathit{stop}$
$Q := a;\ \mathit{stop}$

Corresponding labelled transition systems for these definitions are shown in figure 16.

Here, $\mathrm{Tr}(P) = \{\langle\,\rangle, \langle a\rangle, \langle b\rangle\}$, and $\mathrm{Tr}(Q) = \{\langle\,\rangle, \langle a\rangle\}$. According to the definition above, this means that $P \leq_{tre} Q$. The situation being modelled here is that Q is a specification for an object with a single operation a, while P is a subtype with the same a operation and an additional operation b. It certainly makes sense that if we take an environment which uses an object of type Q—and so only calls the a operation—and then substitute P for Q, that environment will then only call P's a operation, and will observe no change. P has then been successfully substituted for Q, and, in this case, $\leq_{tre}$ appears to be a suitable subtyping relation.

However, although trace extension seems suitable for the examples above, there are some examples for which it does not work well.

Let us now define processes P' and Q' as follows:

$P' := a;\ b;\ \mathit{stop}\ []\ a;\ \mathit{stop}$
$Q' := a;\ b;\ \mathit{stop}$

Corresponding labelled transition systems for these definitions are shown in figure 17.

We now have $\mathrm{Tr}(P') = \mathrm{Tr}(Q') = \{\langle\,\rangle, \langle a\rangle, \langle a, b\rangle\}$. This means that, like the previous example, $P' \leq_{tre} Q'$. If we place Q' in an environment which attempts to use its operation a, then its operation b, both operations will succeed. For instance, we can place Q' synchronised in parallel with $a;\ b;\ \mathit{stop}$, and the resulting process will always be able to perform $\langle a, b\rangle$. If we place P' in the same environment, the b operation may not succeed; that is, the resulting process is not necessarily able to perform $\langle a, b\rangle$. This is because, when P' performs a, it may choose to perform the a action on the right hand side of its choice construct. After performing this a it evolves to stop, and must then refuse to perform any further action, including b. In other words, trace extension does not prevent us from introducing deadlocks in refined processes. This means that P' is not substitutable for Q', and trace extension cannot be considered suitable as

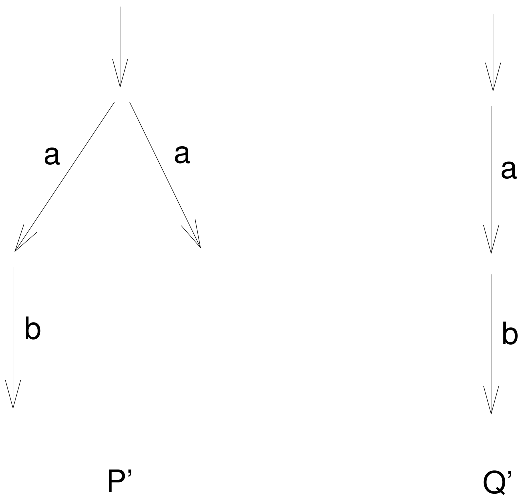

Figure 17: Labelled transition systems for processes P' and Q'

a subtyping relation.

4.3.3 Trace equivalence

There is a third relation based on trace semantics: trace equivalence. For two processes to be trace equivalent, they must have exactly the same set of traces.

The definition of trace equivalence is as follows:

$$P =_{tr} Q \Leftrightarrow \mathrm{Tr}(P) = \mathrm{Tr}(Q)$$

Trace equivalence fails to be a suitable subtyping relation for the same reason as trace extension. With processes P and Q defined as above, $P =_{tr} Q$. Exactly the same argument we used to show that trace extension is not a suitable subtyping relation can then be used against trace equivalence, too; trace equivalence cannot distinguish between processes with different refusal characteristics, and therefore cannot prevent the introduction of deadlocks.

Since trace equivalence is the strictest of the development relations which considers only traces, and it is not strict enough to prevent the introduction of deadlocks, it is now clear that any relation which is able to serve as a subtyping relation must be able to distinguish processes based on something more than just their traces. We will now examine some of LOTOS's development relations which are based on refusals as well as traces.

4.3.4 Testing equivalence

One of the simplest relations based on both traces and refusals is testing equivalence[Led92]. It is also the strictest of the trace/refusals development relations.

Testing equivalence is written $=_{te}$. It holds between two processes, P and Q, exactly when they are trace equivalent (that is, $P =_{tr} Q$) and, after any trace, their refusals are equal.

We can define testing equivalence as follows:

$$P =_{te} Q \Leftrightarrow \mathrm{Tr}(P) = \mathrm{Tr}(Q) \wedge \forall \sigma \in Act^* \bullet \mathrm{Ref}(P, \sigma) = \mathrm{Ref}(Q, \sigma)$$

(We write Act^* to denote the set of all sequences of elements of Act.)

Testing equivalence is a stronger relation than trace equivalence; it implies trace equivalence. Due to the extra conditions on refusals, the processes P and Q from the example in §4.3.2 above are not related by testing equivalence: P can refuse a after the empty trace, while Q cannot.

If we assume that our model of observation of processes is based on trace and refusal semantics, testing equivalence is the strongest relation we can devise[vG90]. It relates only processes which cannot be distinguished by observing which traces and refusals they are capable of.

However, testing equivalence is not suitable for use as a subtyping relation. This is suggested by the fact that it is an equivalence relation. We would like to be able to make such changes to a process as add new operations (that is, new traces), or remove behaviour which no trace is guaranteed to invoke (that is, reduce or remove nondeterminism), and thus create a subtype. However, testing equivalence is too strict a relation for this; it does not permit either of the above, except in certain very limited cases.

For example, let us define:

$$P := a;\ \mathit{stop}\ []\ a;\ \mathit{stop}$$
$$Q := a;\ \mathit{stop}$$

Both P and Q have traces $\{\langle\,\rangle, \langle a\rangle\}$, and both can refuse any set of actions after $\langle a\rangle$, and can refuse any set of actions which does not contain a after $\langle\,\rangle$. They are therefore testing equivalent, and we could refine P to Q, or Q to P, using this relation. However, let us define:

$$P := a;\ b;\ \mathit{stop}$$
$$Q := a;\ b;\ \mathit{stop}\ []\ a;\ c;\ \mathit{stop}$$

In this case, we would like P to be a subtype of Q. The reasoning behind this is as follows. If the environment performs the trace $\langle a, c\rangle$ when synchronised with Q, it would be subject to deadlock: when Q performs $\langle a\rangle$, it may evolve to either $b;\ \mathit{stop}$ or $c;\ \mathit{stop}$, and the environment would then offer only c. If Q evolves to $b;\ \mathit{stop}$, the process would then deadlock; exactly the same result would occur if we used P instead of Q. Thus, we believe that P should be an acceptable subtype of Q. However, testing equivalence does not allow this kind of refinement.

While the testing equivalence relation ensures that the two processes it relates are very similar, and can be substituted for one another, it does not allow the flexibility in refining behaviour that we would like. For example, testing equivalence also does not allow extra traces to be added, which we want to allow extra operations to be added. In fact, it requires the subtype to have exactly the same traces as the supertype.

Testing equivalence appears to be a sufficient but not necessary condition for a subtyping relation.

4.3.5 Extension

We will now consider the LOTOS extension relation[BSS87] as a candidate for a LOTOS subtyping relation.

Extension is written "ext". It is a combination of trace extension and an additional condition which prevents the introduction of deadlocks. P is an extension of Q exactly if P is a trace extension of Q and, after any trace that Q can perform, P does not refuse more than Q (that is, for any trace which Q can perform, if P can perform the trace and refuse a set of actions, Q can also perform the trace and refuse the same set of actions).

In other words, any trace the original process can perform, an extension must also be able to perform, and after that trace, any set of actions the original process cannot refuse, the extension must also be unable to refuse.

Since extension cannot introduce deadlocks, and it allows new traces to be added, it appears, at first, to be an ideal candidate for the subtyping relation. Its definition is

$$P \text{ ext } Q \Leftrightarrow \mathrm{Tr}(P) \supseteq \mathrm{Tr}(Q) \wedge \forall\, \sigma \in \mathrm{Tr}(Q) \bullet \mathrm{Ref}(P, \sigma) \subseteq \mathrm{Ref}(Q, \sigma)$$

Consider two LOTOS processes, P and Q:

$P := a;\ stop\ []\ b;\ stop$
$Q := a;\ stop$

Corresponding labelled transition systems for these definitions appear in figure 16.

P's traces are $\{\langle\rangle, \langle a\rangle, \langle b\rangle\}$. Q's traces are $\{\langle\rangle, \langle a\rangle\}$. Since any trace which P can perform is also a trace of Q, P must be a trace extension of Q. Taking $Act = \{a, b\}$, let us now examine P and Q's refusals.

$\mathrm{Ref}(P, \langle\rangle) = \{\{\}\}$
$\mathrm{Ref}(Q, \langle\rangle) = \{\{\}, \{b\}\}$
$\mathrm{Ref}(P, \langle a\rangle) = \mathbb{P}\, Act$
$\mathrm{Ref}(Q, \langle a\rangle) = \mathbb{P}\, Act$
$\mathrm{Ref}(P, \langle b\rangle) = \mathbb{P}\, Act$
$\mathrm{Ref}(Q, \langle b\rangle) = \{\}$

We can see that, after $\langle\rangle$ or $\langle a\rangle$, P can refuse less than Q. $\langle b\rangle$ is not a trace which Q can perform, so it is not required that P can refuse less than Q after that trace for extension to hold.

Since P can perform every trace that Q can perform (and more), and, after any trace that Q can perform, Q refuses at least everything that P refuses after the same trace, we can see that $P\ \mathrm{ext}\ Q$.

At first glance, it would appear that P should be a subtype of Q. Our definition of behavioural subtyping is based on the principle that a subtype must be usable in any situation where the supertype could be used, and not behave in a way that the supertype could not. If we have a process which may be an instantiation of either P or Q, we can detect which it is by offering the action b to the process. If the b is performed, we know that the process is P, but if b is refused, we must have Q. Since this observation tells us that we are observing an instance of P, our definition of behavioural subtyping tells us that P is not a subtype of Q, and if $P\ \mathrm{ext}\ Q$, but at the same time P is not a subtype of Q, then we obviously cannot use extension as our behavioural subtyping relation.

The corresponding situation in object oriented systems is that of adding a method in a subtype which is not defined in the supertype. However, in strongly typed object oriented systems such as Java or C++, the type checker ensures that it is not possible to call an operation in a situation which would reveal this difference between the supertype and subtype. Other systems, such as Smalltalk, produce error messages when a program calls an undefined operation, or result in undefined behaviour such as the program crashing or giving incorrect results. We will consider the implications of this "undefined behaviour" characteristic in §4.4.

4.3.6 Reduction

We now consider the reduction relation[BSS87], which is like a combination of trace preorder and a property similar to that of extension for preventing the introduction of deadlocks.

Reduction is written "red". P is a reduction of Q exactly if P has no traces which Q cannot perform, and, after any trace, P cannot refuse any set of actions which Q cannot refuse after the same trace.

The definition of reduction is:

$$P\ \mathrm{red}\ Q \Leftrightarrow \mathrm{Tr}(P) \subseteq \mathrm{Tr}(Q) \wedge \forall \sigma \in Act^* \bullet \mathrm{Ref}(P, \sigma) \subseteq \mathrm{Ref}(Q, \sigma)$$

This relation works very much as we would expect to allow a specification to be refined. Those traces which the original specification can perform are those allowed in a reduced specification; however, a reduced process cannot simply remove behaviour at will, because it must not refuse anything which the original specification could not refuse.

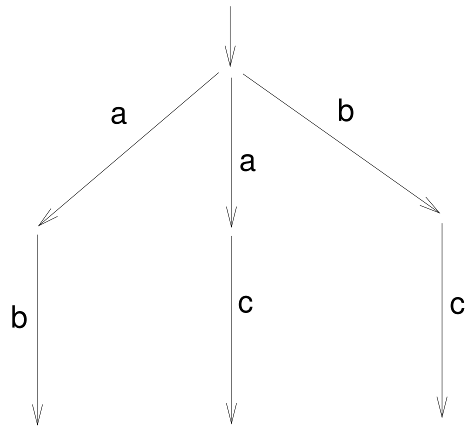

Figure 18: Labelled transition system for Q

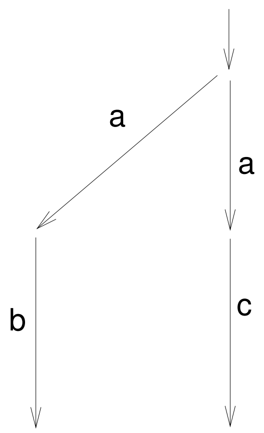

Figure 19: Labelled transition system for P_1; not a reduction of Q

For example, let us define Q as follows:

$Q := a;\ b;\ \mathit{stop}\ []\ a;\ c;\ \mathit{stop}\ []\ b;\ c;\ \mathit{stop}$

Figure 18 shows the labelled transition system of this definition.

Q's traces are $\{\langle\,\rangle, \langle a\rangle, \langle a,b\rangle, \langle a,c\rangle, \langle b\rangle, \langle b,c\rangle\}$. Any reduction of Q must contain no traces other than those in $\mathrm{Tr}(Q)$. Let us now define P_1 as follows:

$P_1 := a;\ b;\ \mathit{stop}\ []\ a;\ c;\ \mathit{stop}$

P_1's labelled transition system is shown in figure 19.

We will define $\mathit{Act} = \{a,b,c\}$ for this example. The traces of P_1 are $\{\langle\,\rangle, \langle a\rangle, \langle a,b\rangle, \langle a,c\rangle\}$, which are a subset of those of Q. However, after the trace $\langle\,\rangle$, P_1 refuses the action b, while Q cannot refuse b. This violates the deadlock-freeness condition of reduction, and thus P_1 is not a reduction of Q.

However, let us define P_2 as follows:

$P_2 := a;\ b;\ \mathit{stop}\ []\ b;\ c;\ \mathit{stop}$

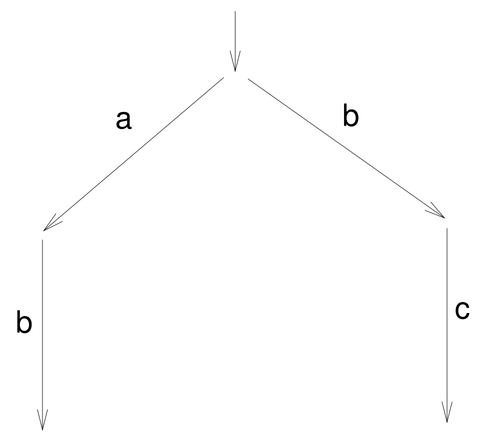

Figure 20: Labelled transition system for P_2; a reduction of Q

P_2's labelled transition system is shown in figure 20.

In this case, P_2's traces are $\{\langle\rangle, \langle a\rangle, \langle a, b\rangle, \langle b\rangle, \langle b, c\rangle\}$, which are a subset of those of Q. Let us examine the refusals of P_2 and Q:

$$\begin{aligned}
&\mathrm{Ref}(P_2, \langle\rangle) = \{\{\}, \{c\}\} \\
&\mathrm{Ref}(Q, \langle\rangle) = \{\{\}, \{c\}\}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Ref}(P_2, \langle a\rangle) = \{\{\}, \{a\}, \{c\}, \{a, c\}\} \\
&\mathrm{Ref}(Q, \langle a\rangle) = \{\{\}, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}\}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Ref}(P_2, \langle b\rangle) = \{\{\}, \{a\}, \{b\}, \{a, b\}\} \\
&\mathrm{Ref}(Q, \langle b\rangle) = \{\{\}, \{a\}, \{b\}, \{a, b\}\}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Ref}(P_2, \langle a, b\rangle) = \mathbb{P}\,Act \\
&\mathrm{Ref}(Q, \langle a, b\rangle) = \mathbb{P}\,Act
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Ref}(P_2, \langle a, c\rangle) = \{\} \\
&\mathrm{Ref}(Q, \langle a, c\rangle) = \mathbb{P}\,Act
\end{aligned}$$

$$\begin{aligned}
&\mathrm{Ref}(P_2, \langle b, c\rangle) = \mathbb{P}\,Act \\
&\mathrm{Ref}(Q, \langle b, c\rangle) = \mathbb{P}\,Act
\end{aligned}$$

Neither process has any refusals after any other traces.

We can see from this list that, after any trace, P_2 cannot refuse any set of actions which Q cannot refuse. As such, P_2 is a reduction of Q.

If we have two processes, one of which is a reduction of the other, the reduced process may not exhibit any behaviour that the original process could not exhibit, either by performing a trace that the original process could not perform or by refusing actions which the original process could not refuse. The only changes that can be made in a reduced process, in fact, are that nondeterminism may be reduced or removed. If a process can make nondeterministic choices between branches of its behaviour, a reduced process need not necessarily include all of those branches.

This appears to be enough to satisfy our requirements that a subtyping relation should ensure that a subtype can be used in place of its supertype in any environment without the environment being able to tell the difference.

However, reduction suffers the same problem as trace preorder; it does not allow new traces to be added to the process, and so does not allow new operations to be added. At first glance, reduction therefore seems not to be an appropriate relation for use as a subtyping relation.

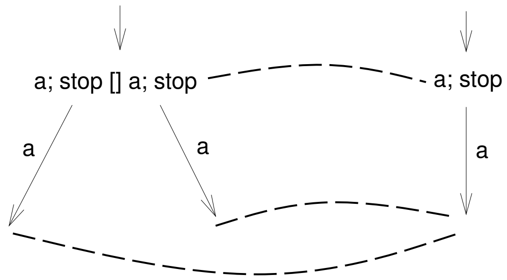

Figure 21: $a;\ stop\ []\ a;\ stop \sim a;\ stop$

However, in §4.4 we will see how a difference between the state-based and process algebraic specification settings indicates that reduction should be used, despite this drawback.

4.3.7 Bisimulation

One of the strongest relations defined over process algebras is bisimulation equivalence[Mil89]. Two processes are considered to be bisimilar exactly if there is a bisimulation relation by which they are related. There are two kinds of bisimulation relation: strong bisimulations and weak bisimulations. These give rise to, respectively, the strong bisimulation equivalence and the weak bisimulation equivalence. We will define these formally below.

For processes P and Q to be strongly bisimilar (related by the strong bisimulation equivalence), denoted $P \sim Q$, the following condition must hold: if P can perform any given action a (including the internal action), and thus evolve to behaviour P', Q must be able to perform the same action a and evolve to some behaviour Q' such that P' and Q' are strongly bisimilar. The converse condition must also hold.

To illustrate this, consider the following examples:

$$(a;\ stop\ |[Act]|\ b;\ stop) \sim stop$$
$$a;\ stop \not\sim b;\ stop$$
$$a;\ stop \not\sim i;\ a;\ stop$$

$$a;\ stop\ []\ a;\ stop \sim a;\ stop$$

$$P := a;\ a;\ P$$
$$Q := a;\ Q$$
$$P \sim Q$$

We have illustrated the last two of these examples in figures 21 and 22. The strong bisimulations are shown with dotted lines.

The weak bisimulation equivalence, denoted $\approx$, is very similar to strong bisimulation, except that it treats internal actions as unobservable. Strong bisimulation considers the internal action, i, to be an ordinary action, so $a;\ i;\ B \not\sim a;\ B$. However, $a;\ i;\ B \approx a;\ B$.

A bisimulation between processes P and Q is a relation from behaviours to behaviours; we denote the set of all behaviours Beh. If a pair of behaviours, B_1 and B_2, are to be related by a strong bisimulation, $\mathcal{R}$, the following conditions must hold, for all $a \in Act \cup \{i\}$:

$$\forall B_1' \in Beh \bullet B_1 \xrightarrow{a} B_1' \Rightarrow \exists B_2' \in Beh \bullet B_2 \xrightarrow{a} B_2' \wedge B_1'\ \mathcal{R}\ B_2'$$
$$\forall B_2' \in Beh \bullet B_2 \xrightarrow{a} B_2' \Rightarrow \exists B_1' \in Beh \bullet B_1 \xrightarrow{a} B_1' \wedge B_1'\ \mathcal{R}\ B_2'$$

Before we can define weak bisimulation, we must introduce a new piece of notation: If a is

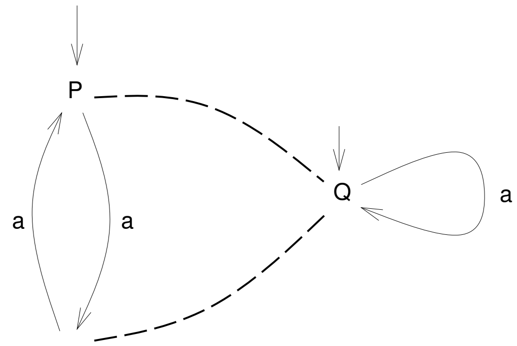

Figure 22: $P \sim Q$

a trace, $\widehat{a}$ is the trace obtained by removing any and all internal actions i from a. For example, $\widehat{\langle a, b, i, c\rangle} = \langle a, b, c\rangle$ and $\widehat{\langle i, a, i\rangle} = \langle a\rangle$. In particular, note that $\widehat{\langle i\rangle} = \langle\,\rangle$.

If a pair of behaviours, B_1 and B_2, are to be related by a weak bisimulation, $\mathcal{R}$, the following conditions must hold, for all $a \in Act \cup \{i\}$:

$$\forall B_1' \in Beh \bullet B_1 \xrightarrow{a} B_1' \Rightarrow \exists B_2' \in Beh \bullet B_2 \xRightarrow{\widehat{\langle a\rangle}} B_2' \wedge B_1' \mathcal{R} B_2'$$

$$\forall B_2' \in Beh \bullet B_2 \xrightarrow{a} B_2' \Rightarrow \exists B_1' \in Beh \bullet B_1 \xRightarrow{\widehat{\langle a\rangle}} B_1' \wedge B_1' \mathcal{R} B_2'$$

($\Rightarrow$ is defined in §A.12.)

Bisimulation is a very strong relation and, even in its weak form, it is not suitable for use as a subtyping relation. This is mainly because bisimulation is an equivalence—and we have already established that equivalences are not suitable as subtyping relations—but also due to the extreme nature of the observational model bisimulation characterises.

While relations such as the trace preorder are based only on an observer capable of observing traces when they occur, and the testing equivalence is based on an observer capable of offering a process an action after a trace and detecting whether the process can accept or refuse it, bisimulation characterises an observer capable of "cloning" the process under test at any time and testing the resulting processes separately. This is a much stronger notion of observation than is likely to be necessary for our subtyping relation.

4.4 Unspecified behaviour

In the previous section, we examined a number of standard LOTOS development relations and found none of them directly applicable to the problem of subtyping in process algebra.

We found that the LOTOS reduction relation came closest, but that it could not permit all the kinds of subtyping possible with our state-based relation. We will now analyse why this is, shedding some light on one major difference between the two specification settings.

4.4.1 Why does reduction not work as a subtyping relation?

Any LOTOS process must refuse any operation which its specification says it refuses. When it does so, the process's environment may be affected by that refusal. For example, if we consider a process offering only the action a, synchronised in parallel with a process offering both a and b, the only action which both processes can perform is a. For the processes to progress, the next action performed must be a. We can see that the process offering only a, and refusing b, has influenced the behaviour of the process offering both actions.

We can thus see that since the refusals of a process affect its environment, those refusals are part of the specified behaviour of a LOTOS process.

In a state-based specification, each operation specified has defined limits to its applicability: its precondition. However, there is no universal agreement on the exact meaning of the precondition, except that if an operation is invoked when its precondition holds, its behaviour will conform with the specified predicate.

In [Str95], for example, operations are classified as either *passive* or *active*. An active operation is considered to be invoked automatically whenever its precondition is fulfilled, while passive operations are invoked only by a specific request from the environment. This "firing conditions" approach involves placing two different interpretations on whether an operation's precondition is fulfilled in a single specification. If that is a useful thing to do, then it is unsurprising that there is no general agreement on the meaning of an operation when its precondition is not fulfilled!

According to [Str95], the most common interpretation of an unfulfilled precondition, and perhaps the "lowest common denominator" interpretation, is that the operation's behaviour is not defined by the specification. The specification does not require anything of the implementation of the operation outside of the specified precondition. The operation may cause the entire system to halt, it may give an incorrect result, or it may even do something useful if its precondition is not fulfilled; we say its behaviour is *unspecified*. This contrasts strongly with the behaviour of a LOTOS process which refuses an action: refusals are well-defined, and do not cause unspecified behaviour.

The state-based subtyping relation allows us to add new operations in a subtype, or to relax the operation's precondition so that it may be applied in cases where it originally could not. It is able to allow this, without producing a subtype which can behave in a way which breaks the supertype's specification, because our state-based specification language treats an operation's undefined behaviour as unspecified.

The LOTOS specification language, and other process algebras like CSP, treat undefined behaviour as "negatively specified". If the specification does not allow the behaviour, the process is guaranteed to refuse it; when offered an action it is not specified to be able to accept, a LOTOS process is effectively specified to refuse the action. As such, we cannot add arbitrary new behaviour to a LOTOS specification and call the result a subtype of the original, even if we could add behaviour analogously to a state-based specification, and thus create a subtype.

We can now see that the simple form of translation from state-based specifications to LOTOS which was suggested in §3.4 cannot work; the "impedance mismatch" between the state-based notion of unspecified behaviour and LOTOS's notion of refusals is too great to allow our subtyping relation to translate meaningfully.

We will approach this problem as follows. We will attempt to change our translation in such a way that unspecified behaviour in the original state-based specification translates to unspecified behaviour in the derived LOTOS specification. Before we can do this, we must find a way to produce a LOTOS process containing unspecified behaviour.

4.4.2 Representing unspecified behaviour in LOTOS

What do we mean by unspecified behaviour? Let us consider an example. We define two state-based specifications as follows:

P :

State space: $\{S_0, S_1\}$
Initial states: S_0
Operation a ($nil \rightarrow nil$): $before = S_0 \wedge after = S_1$
Operation b ($nil \rightarrow nil$): $false$

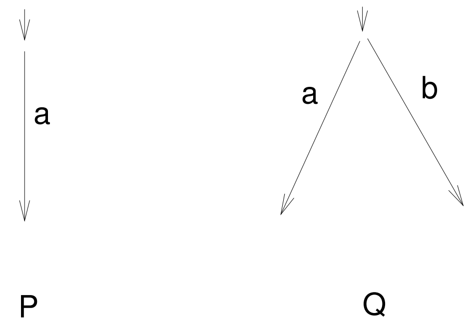

Figure 23: Labelled transition systems for P and Q

Q :

State space: $\{S_0, S_1, S_2\}$
Initial states: S_0
Operation a ($nil \to nil$): $before = S_0 \wedge after = S_1$
Operation b ($nil \to nil$): $before = S_0 \wedge after = S_2$

We can show that Q is a subtype of P according to our state-based subtyping relation. Operation a is the same in both P and Q, while operation b is newly introduced; this satisfies the three conditions given in §3.2.2.

If we now translate (intuitively) P and Q into LOTOS, using §3.4 as a guide, we get the following processes:

$P := a;\ stop$
$Q := a;\ stop\ []\ b;\ stop$

Figure 23 shows the labelled transition systems corresponding to these two processes.

Since none of our operations take any arguments or produce any return values, we have simply translated the invocation of operation a into the action a, and the invocation of operation b into the action b. Each state in the state-based specification has become a state in the LOTOS behaviour, and each such state comprises a choice between all the operation invocations possible in that state. Recalling the above discussion, we note that we are considering an operation "possible" only if its precondition holds.

What if an operation is not "possible"—if its precondition does not hold? So far, we have assumed that the state in which the precondition fails to hold should simply not contain a choice corresponding to that operation. In those states where no operation's precondition holds, we made that state's behaviour *stop*, which is effectively the same as an empty choice: a "choice" between no courses of action. This is what results in our LOTOS processes refusing actions in situations where the corresponding state-based object performs unspecified behaviour.

Let us imagine that we have a LOTOS process definition called Ω, which exhibits "unspecified behaviour" in the sense that state-based specifications exhibit unspecified behaviour when one of their operations is invoked outside of its precondition. Let us now rewrite our above definitions of P and Q, as follows:

$P' := a;\ (a;\ \Omega\ []\ b;\ \Omega)\ []\ b;\ \Omega$
$Q' := a;\ (a;\ \Omega\ []\ b;\ \Omega)\ []\ b;\ (a;\ \Omega\ []\ b;\ \Omega)$

Figure 24 shows the labelled transition systems corresponding to these two processes. What we have done is more obvious from the labelled transition system than from the LOTOS specification. For reference, the transitions which were present in P and Q are marked with heavy lines on the

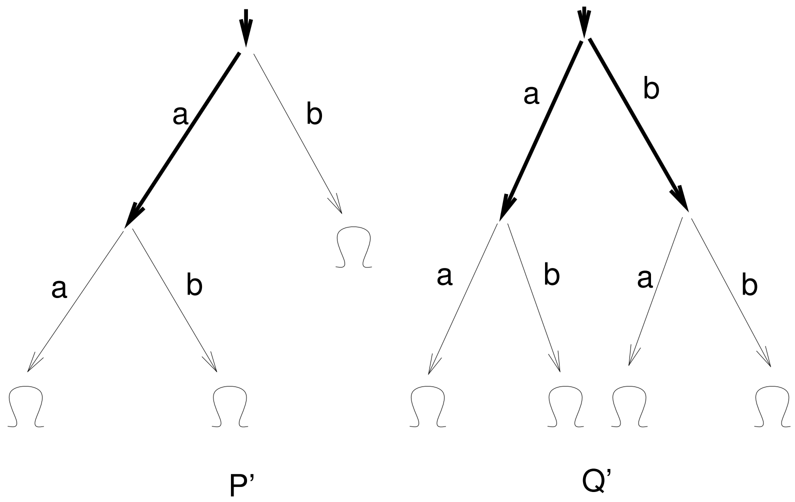

Figure 24: Labelled transition systems for P' and Q'

labelled transition systems of P' and Q'. At each state, we added a new choice of behaviour for each action which the process did not offer in that state. So, the initial state of P offered a, but not b. We therefore added a choice allowing b. Taking this new transition, however, results in the process evolving to Ω, our "unspecified" behaviour. We will discuss this procedure in more detail in §4.5.

4.4.3 Defining Ω

Now that we can see what will be done with Ω, we can see more easily how to define it. The purpose of Ω is to exhibit "unspecified" behaviour. More accurately, after reaching Ω by some trace, it should be possible to refine Ω's behaviour into any other behaviour using the subtyping relation. If we are to use reduction as our subtyping relation, then, we need to define Ω as the *least reduced* process possible.

According to the definition of "red", we see that a more reduced process has fewer traces and fewer refusals than a process which is less reduced. To produce a process, Ω, which is least reduced, then, we should be trying to maximize both the traces and the refusals of Ω. Let us now examine some possible definitions for Ω.

There are several different definitions of Ω which we could use, some of which we give here. Since we believe that red should be used as our LOTOS subtyping relation, our main criterion in evaluating these definitions will be that the definition of Ω should exhibit all traces and all refusals.

We first consider this simple definition:

$$\Omega' := i;\ \Omega'$$

This process, in LOTOS, has no traces but the empty trace, $\langle\,\rangle$. To have a trace, a process must be able to perform the trace and thereby evolve to some state; Ω' can evolve only to itself, and in so doing performs the empty trace.

While Ω' is clearly useless for our purposes, it is the least refined process for some other refinement models, such as the failures/divergences model of CSP[Hoa85]. In CSP, this process's behaviour is undefined because of CSP's notion of divergence: if a process is defined in terms of itself its behaviour cannot be calculated.

However, trace/refusals semantics does not distinguish divergence; Ω' is testing equivalent to stop. This does not fulfil our requirements for a process which exhibits undefined behaviour.

There are several other candidate definitions:

$$
\begin{aligned}
\Omega := &\ (\mathit{choice}\ a : \mathit{Act}\ []\ i;\ a;\ \Omega)\\
&\ []\\
&\ i;\ \mathit{stop}
\end{aligned}
$$

$$
\begin{aligned}
\Omega'' := &\ (\mathit{choice}\ a : \mathit{Act}\ []\ i;\ a;\ \Omega'')\\
&\ []\\
&\ i;\ \mathit{stop}\\
&\ []\\
&\ i;\ \Omega''
\end{aligned}
$$

$$
\begin{aligned}
\beta := &\ (\mathit{choice}\ a : \mathit{Act}\ []\ a;\ \beta)\\
&\ []\\
&\ i;\ \mathit{stop}
\end{aligned}
$$

The first line of each of these process definitions ensures that they each can perform any trace. Due to the internal action in the first line of Ω and Ω'', those two processes can "commit" to any particular action, and accept only that action next; this introduces a large number of refusals, but does not enable the process to refuse *everything at once*. For this reason, each process has another choice, $i;\ \mathit{stop}$, which enables the process to, at any stage, refuse any and every set of actions. Because of this, β also has all possible refusals, even though it does not "commit" to actions by using an internal action. Ω'' also has a further choice, $i;\ \Omega''$, which makes it divergent when observed using a semantics which distinguishes divergent processes.

Because all of Ω, Ω'' and β can perform any trace, and can refuse any set of actions after any trace, they are all testing equivalent; that is, they all reduce one another. Since $\mathrm{Tr}(\Omega) = \mathit{Act}^*$ and $\forall \sigma \in \mathit{Act}^* \bullet \mathrm{Ref}(\Omega, \sigma) = \mathbb{P}\,\mathit{Act}$, they occur at the top of the reduction preorder; every behaviour is a reduction of them.

Ω' and Ω'' are divergent, whereas the other two are not. This makes no difference in LOTOS, but Ω' and Ω'' may be useful when applying our technique to other process algebras.

For our purposes, which involve the use of reduction, they are all alike. We will use the definition labelled Ω above.

4.5 Reduction as subtyping

Now that we have a definition for Ω, we will clarify the way in which we will translate state-based specifications into LOTOS, combining Ω into the resulting process definition.

In our state-based subtyping relation, it is possible to create a subtype not only by adding new operations, but also by allowing existing operations to be invoked in new orderings, or to allow them to be invoked with arguments which were not valid in the supertype.

We can treat new orderings of operations in a similar way to new operations; it is just another way to add new functionality in a subtype which will not have any effect when an object is used according to the supertype's specification.

Since adding new orderings of operations seems simpler than adding new operations, we will consider that first, using buffer objects as an example. We will consider the addition of new operations in §4.5.2. We will defer any examination of the effects of data-passing to the next chapter.

4.5.1 Example: Buffers

A buffer performs a similar function in process-oriented software as a variable does in structured programming: a value may be stored into it (using a "*receive*" operation) and may later be read back (with a "*transmit*" operation). There are two significant differences. With a buffer, only one *transmit* operation may be made for each *receive*, and some buffers can hold more than one value at a time, with each *transmit* retrieving the oldest un*transmit*ed value.

We will use the same state-based specification of a one-place buffer as we did in chapter 3:

> $buffer1$:
>
> **State space**: $\mathbb{N} \cup \{VOID\}$
> **Initial states**: $VOID$
> **Operation** receive ($\mathbb{N} \to nil$): $before = VOID \wedge after = arg$
> **Operation** transmit ($nil \to \mathbb{N}$): $result = before \wedge after = VOID$

(We can omit the term $before \neq VOID$ from *transmit*'s predicate, since the implicit $result \in \mathbb{N}$ combined with the explicit $result = before$ effectively subsumes it.)

We will also define a two-place buffer as follows:

> $buffer2$:
>
> **State space**: $(\mathbb{N} \times \mathbb{N}) \cup \mathbb{N} \cup \{VOID\}$
> **Initial states**: $VOID$
> **Operation** receive ($\mathbb{N} \to nil$): $(before = VOID \wedge after = arg))$
> $\quad\quad \vee (before \in \mathbb{N} \wedge after = (before, arg))$
> **Operation** transmit ($nil \to \mathbb{N}$): $before = (result, after)$
> $\quad\quad \vee (result = before \wedge after = VOID)$

Let us ignore, for the moment, that the *transmit* and *receive* operations of these specifications take arguments and return results, and translate them into LOTOS as follows:

$$buffer1 := receive;\ transmit;\ buffer1$$

Our two-place buffer requires an auxiliary process definition to translate, because of the way its behaviour branches. It translates as follows:

$$buffer2 := receive;\ buffer2a$$

$$buffer2a := transmit;\ buffer2\ []\ receive;\ transmit;\ buffer2a$$

These LOTOS processes may also be represented by labelled transition systems. The labelled transition systems corresponding to *buffer1* and *buffer2* are given in figure 25.

For each of these definitions, we can easily see that the only actions the processes can engage in are $\mathcal{L} = \{receive, transmit\}$. We will assume, for the moment, that *receive* and *transmit* are the only actions it is possible to perform; that is, we will take $Act = \{receive, transmit\}$.

The state-based specifications $buffer1$ and $buffer2$ are related by subtyping. This is the case because a two-place buffer can be used where a one-place buffer is required—in such a situation, its extra functionality of accepting two un*transmit*ed values would not be used. However, the LOTOS definition $buffer2$ is not a reduction of $buffer1$. If we are to use reduction as our subtyping relation, we will have to translate the processes differently, so that $buffer2$ becomes a reduction of $buffer1$. However, we should be able to justify our translation as still being intuitively correct. Let us modify the two processes to obtain $buffer1'$ and $buffer2'$, as shown in figure 26.

We have added transitions such that every node has at least one transition leading away from it for every possible action (for this example, since $Act = \{receive, transmit\}$, that means each of *receive* and *transmit*.) By following any of the transitions we have added, the process evolves to Ω.

Because Ω is at the top of the reduction preorder—that is, because every other process is a reduction of Ω—we can see that $buffer2'$ is now a reduction of $buffer1'$. Wherever $buffer2$ has a trace which $buffer1$ could originally not perform, our modification has ensured that $buffer1'$ can now perform that trace. Since Ω can refuse any set of actions after any trace, we can see that, after any trace added by $buffer2$ over $buffer1$, $buffer2'$ cannot refuse more than $buffer1'$.

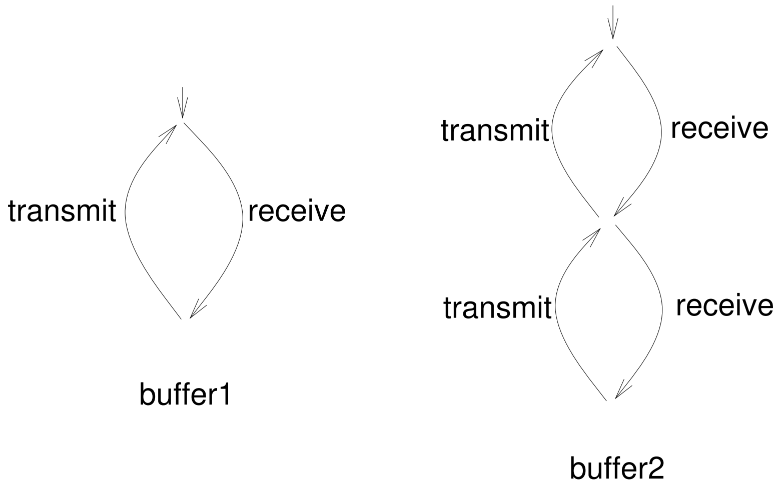

Figure 25: *buffer1* and *buffer2*

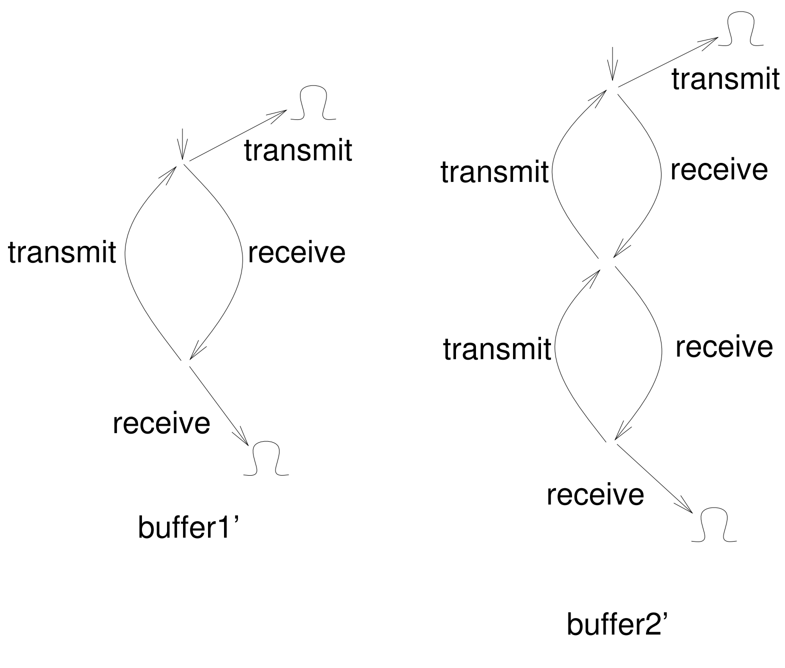

Figure 26: $buffer1'$ and $buffer2'$ with transitions leading to Ω

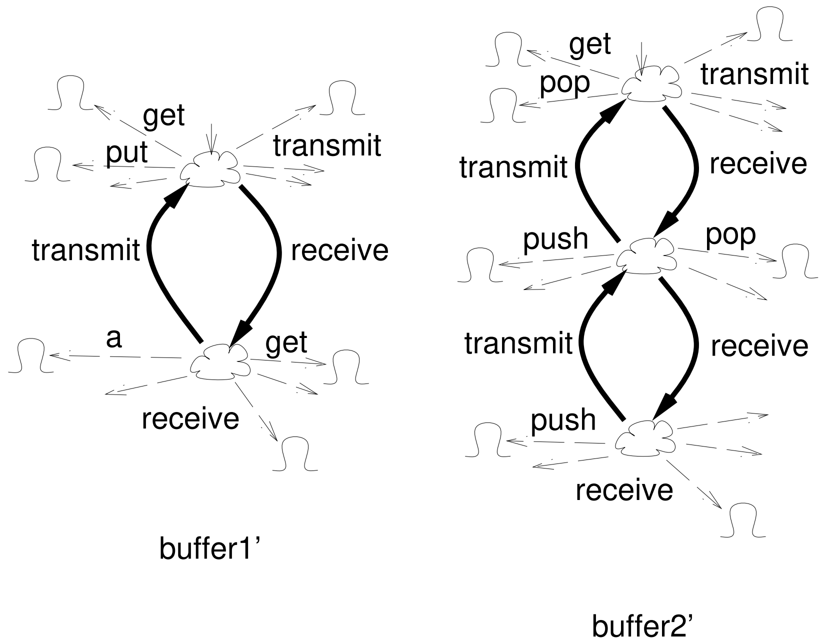

Figure 27: *buffer1'* and *buffer2'* with "cloudy nodes"

Thus, we can see that, with our modified translation, *buffer*$2'$ is a reduction of *buffer*$1'$.

As a result of our interpretation of state-based subtyping—that an attempt to perform an undefined operation on an object has unspecified results—we will now incorporate the above technique into our translation from our state-based specification language into LOTOS.

We now have specifications of a one-place and a two-place buffer which are related by the subtyping relation. As we have presented it, this method only works if we consider the alphabet of possible actions to be just *receive* and *transmit*. We will now make this method more general.

In figure 26, we assumed that the only actions available were *receive* and *transmit*. To generalise our method to cope with any definition of Act, we must simply add appropriate transitions, as we did before, for *all* the actions which the state does not already offer. We will first change our diagramming notation to reflect this. Any node which we are translating according to our new method—that is, for which we are adding transitions leading to Ω—we will depict as a "cloud" symbol. This is intended to symbolise the "unspecified" nature of the extra transitions. We have shown this in figure 27, which also shows a few of the transitions implied by these clouds as dotted arrows.

The intended meaning of these "cloudy nodes" is that, for every node represented as a cloud, and for every action in Act, if there is no transition labelled with that action leading away from that node, an implicit transition labelled with that action leads from that node to Ω.

To relate these "cloudy nodes" to their LOTOS equivalents, we will use the following semi-formal definition. This definition is not valid LOTOS, because LOTOS is not able to express what we want, so we present the following as a kind of shorthand.

$$\text{otheractions}(\{a_1, \ldots, a_n\}, B) \equiv (\text{choice } c : (Act \backslash \{a_1, \ldots, a_n\}) \; [] \; c; \; B)$$

With this definition, *otheractions* takes a set containing those actions which are considered to be already catered for by the other behaviour of the process in which it is used, and a behaviour, which will be Ω or a similar behaviour.

Using this definition of *otheractions*, we can now write new definitions for one-place and

$$
\begin{aligned}
\mathit{buffer}1'' &:= \mathrm{otheractions}(\{\mathit{receive}\}, \Omega) \\
&\ [] \\
&\ \mathit{receive};\ \mathit{buffer}1''_1 \\
\mathit{buffer}1''_1 &:= \mathit{transmit};\ \mathit{buffer}1'' \\
&\ [] \\
&\ \mathrm{otheractions}(\{\mathit{transmit}\}, \Omega) \\
\mathit{buffer}2'' &:= \mathrm{otheractions}(\{\mathit{receive}\}, \Omega) \\
&\ [] \\
&\ \mathit{receive};\ \mathit{buffer}2''_1 \\
\mathit{buffer}2''_1 &:= \mathit{transmit};\ \mathit{buffer}2'' \\
&\ [] \\
&\ \mathit{receive};\ \mathit{buffer}2''_2 \\
&\ [] \\
&\ \mathrm{otheractions}(\{\mathit{receive}, \mathit{transmit}\}, \Omega) \\
\mathit{buffer}2''_2 &:= \mathit{transmit};\ \mathit{buffer}2''_1 \\
&\ [] \\
&\ \mathrm{otheractions}(\{\mathit{transmit}\}, \Omega)
\end{aligned}
$$

Figure 28: LOTOS definition of one- and two-place buffers with clouds

two-place buffers, related by reduction as we wish. These definitions are shown in figure 28. For the same reasons that $\mathit{buffer}2'$ is a reduction of $\mathit{buffer}1'$, $\mathit{buffer}2''$ is a reduction of $\mathit{buffer}1''$.

We now have a method (albeit a very informal one) for translating state-based specifications into LOTOS in such a way that any action which was unspecified in the original specification is now accepted, but the process's behaviour after that action is unpredictable, where our notion of "unpredictable" or "unspecified" behaviour is given by our definition of Ω. In any subtype in which that action *is* specified, the subtype's behaviour after accepting the action must be a reduction of the corresponding behaviour in the supertype, because *any* behaviour is a reduction of Ω. In effect, we have explicitly specified that the translated LOTOS process's behaviour is unspecified after performing any action which, in the state-based specification, was unspecified. This corresponds to our intuition about the unspecified behaviour of state-based specification languages.

These two processes are now related in the way we wish. $\mathit{buffer}2''$ is both a reduction and a subtype of $\mathit{buffer}1''$. This is consistent with our notion that reduction is a reasonable process algebraic instantiation of the subtyping relation.

4.5.2 Example: Conference binding objects

We will now consider some more examples. The "buffer" examples in the previous subsection required only that the subtype should be able to accept operations in different orderings than that which its supertype could accept. We now consider some examples which include the addition of new operations to an object's interface. We attempt to model binding objects which could be used in distributed multimedia conferences based on ODP-style distributed systems.

A conference binding object is designed to regulate the multimedia data streams in an on-line conference. The model used is that a binding object is created for the conference, then each party connects to the binding object. We assume that each user's data stream can be split up into a sequence of frames: small data units which the protocol will not subdivide. For example, a frame might carry one hundredth of a second of speech data, or a portion of video data sufficient to perform a single screen refresh. We do not model frames as such, but instead we model the act of accepting a frame by the conference binding object for retransmission to all the conference participants as a single operation in the state-based world, or as a LOTOS action.

We will concentrate on binding objects which allow only one participant to speak at any one time. A variety of binding types may be formed by using different methods for choosing who is allowed to speak when.

To simplify the presentation, we will specify conference binding objects which allow exactly three participants to communicate through them. We will not model the process of setting up or shutting down the binding, or the process of connecting the participants to the binding object.

We will present two different types of binding object, implementing two different styles of conference, and then investigate the subtyping relationship between them.

Round-robin

Each user connected to the conference binding object is capable of generating a stream of frames. The binding object may receive frames continuously from each participant, or it may only receive frames when the user is allowed to speak. We do not model this; rather, we are only interested in the retransmission of frames from one participant to the other participants. When the first user is allowed to speak, the conference binding object will retransmit the frames she generates to all the other conference participants. We will represent the retransmission of each such frame by an operation or action named $speak1$. Similarly, the retransmission of the second user's frames is represented as $speak2$, and the third user's frames as $speak3$. Each user also has a "pass" button, which she presses when she has finished speaking. The binding object's reaction to the pressing of these buttons is modelled as $pass1$, $pass2$ and $pass3$.

Our first binding object, the round-robin binding, allows the first user to speak continuously until she presses her pass button. The second user may then speak, on the same terms, and so on. After the third speaker has finished, the first speaker is again allowed to speak. Let us first give a state-based specification showing how this is to work:

RR :

State space: $\{S1, S2, S3\}$
Initial states: $S1$
Operation speak1 $(nil \rightarrow nil)$: $before = after = S1$
Operation speak2 $(nil \rightarrow nil)$: $before = after = S2$
Operation speak3 $(nil \rightarrow nil)$: $before = after = S3$
Operation pass1 $(nil \rightarrow nil)$: $before = S1 \wedge after = S2$
Operation pass2 $(nil \rightarrow nil)$: $before = S2 \wedge after = S3$
Operation pass3 $(nil \rightarrow nil)$: $before = S3 \wedge after = S1$

We can now translate this into LOTOS using the method described in §4.5.1. The result is as follows:

$$
\begin{aligned}
RR := {} & speak1;\ RR \\
& [] \\
& pass1;\ RR_2 \\
& [] \\
& \text{otheractions}(\{speak1, pass1\}, \Omega)
\end{aligned}
$$

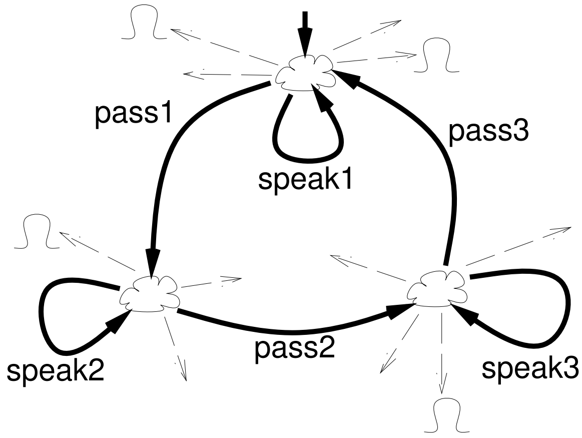

Figure 29: Round-robin conference binding object, RR

$$\begin{aligned}
RR_2 := {} & \mathit{speak}2;\ RR_2 \\
& [] \\
& \mathit{pass}2;\ RR_3 \\
& [] \\
& \mathrm{otheractions}(\{\mathit{speak}2, \mathit{pass}2\}, \Omega) \\
RR_3 := {} & \mathit{speak}3;\ RR_3 \\
& [] \\
& \mathit{pass}3;\ RR \\
& [] \\
& \mathrm{otheractions}(\{\mathit{speak}3, \mathit{pass}3\}, \Omega)
\end{aligned}$$

According to these definitions, the round-robin binding object's behaviour is unspecified after an inappropriate action, such as *pass*2 being performed while the object's behaviour is RR_3. This would not affect the system we have described, since the actions we are using to describe the system denote the activity of the binding object when it is informed that a "pass" button has been pushed, or that a frame has arrived from a user. If the binding object does not respond to a button push, or does not retransmit the frame, the action does not occur.

We can now draw the labelled transition system for RR, using our "cloud" notation. The diagram is shown in figure 29.

Round-robin with executive interrupt

Round-robin with executive interrupt is similar to round-robin, except that there is an additional participant called the chair. The chair is capable of generating a stream of frames in the same way as the users, but the action associated with the frames is *shout*. The chair has two buttons which generate actions *interrupt* and *back*.

The round-robin with executive interrupt works just like the normal round-robin, except that if the chair presses his "interrupt" button, he can interrupt any other speaker, speak for as long as he wishes, then press the "back" button to return the floor to the speaker who was interrupted.

Round-robin with executive interrupt is specified as follows:

RREI :

State space: $\{S1, S2, S3, C1, C2, C3\}$
Initial states: $S1$
Operation speak1 $(nil \to nil)$: $before = after = S1$
Operation speak2 $(nil \to nil)$: $before = after = S2$
Operation speak3 $(nil \to nil)$: $before = after = S3$
Operation pass1 $(nil \to nil)$: $before = S1 \wedge after = S2$
Operation pass2 $(nil \to nil)$: $before = S2 \wedge after = S3$
Operation pass3 $(nil \to nil)$: $before = S3 \wedge after = S1$
Operation shout $(nil \to nil)$: $before = after \wedge before \in \{C1, C2, C3\}$
Operation interrupt $(nil \to nil)$: $(before, after) \in \{(S1, C1), (S2, C2), (S3, C3)\}$
Operation back $(nil \to nil)$: $(before, after) \in \{(C1, S1), (C2, S2), (C3, S3)\}$

We can translate this into LOTOS as shown in figure 30. A labelled transition system of this process is shown in figure 31.

Analysis

If we examine our state-based specifications for *RR* and *RREI*, we can see that they are quite similar. *RREI* has three more states, and three new operations, *interrupt*, *shout* and *back*. Applying the definition of subtyping from §3.2.2, we can see that *RREI* is a subtype of *RR*: the only changes from *RR* to *RREI* are allowed by the "weakening preconditions" rule. We can conclude that *RREI* is a subtype of *RR*.

If the LOTOS translation of *RREI* never performs its *interrupt* action, it will behave in exactly the same way as the LOTOS translation of *RR*. We can see that this is true by examining the labelled transition systems for each of *RR* and *RREI*. If we changed each transition labelled *interrupt* in *RREI* to lead to Ω, we would obtain a labelled transition system equivalent to that of *RR*.

If *RR* ever performs the action *interrupt*, it will evolve to Ω and its behaviour will become unpredictable. If *RREI* performs *interrupt*, it will behave as defined in its specification. Because Ω has the property of being a least reduced process, we can be sure that *RREI*'s behaviour after performing *interrupt* will be a reduction of Ω. We can see that, as a result, *RREI* is a reduction of *RR*.

In §4.5.1, we saw that we can weaken a state-based operation's preconditions to form a subtype, and the corresponding LOTOS translations would be related by reduction. The example in §4.5.2 shows that we can add entirely new operations in a state-based subtype, and still get LOTOS translations which are related by reduction. This seems to be consistent with our idea that reduction can act as a process algebraic subtyping relation.

4.6 Conclusion

In this chapter, we have examined several of the better-known LOTOS development relations, evaluating how nearly they correspond to our state-based subtyping relation. In particular, we wanted to be able to take a pair of state-based specifications, one of which was a subtype of the other, translate both specifications into LOTOS, and have the resulting specifications related in the same way by our LOTOS subtyping relation.

We decided that none of the relations we examined worked perfectly with the translation we outlined in §3.4, but that reduction has properties which corresponded well with our intuitive understanding of subtyping. We then examined the way that each of our specification notations treats unspecified behaviour, and discovered that while state-based systems typically guarantee nothing about unspecified behaviour, LOTOS processes refuse any actions they are not specified to be able to accept.

$$
\begin{aligned}
RREI :=\ & speak1;\ RREI \\
& [] \\
& pass1;\ RREI_2 \\
& [] \\
& interrupt;\ CHAIR_1 \\
& [] \\
& \mathrm{otheractions}(\{speak1, pass1, interrupt\}, \Omega)
\end{aligned}
$$

$$
\begin{aligned}
RREI_2 :=\ & speak2;\ RREI_2 \\
& [] \\
& pass2;\ RREI_3 \\
& [] \\
& interrupt;\ CHAIR_2 \\
& [] \\
& \mathrm{otheractions}(\{speak2, pass2, interrupt\}, \Omega)
\end{aligned}
$$

$$
\begin{aligned}
RREI_3 :=\ & speak3;\ RREI_3 \\
& [] \\
& pass3;\ RREI \\
& [] \\
& interrupt;\ CHAIR_3 \\
& [] \\
& \mathrm{otheractions}(\{speak3, pass3, interrupt\}, \Omega)
\end{aligned}
$$

$$
\begin{aligned}
CHAIR_1 :=\ & shout;\ CHAIR_1 \\
& [] \\
& back;\ RREI \\
& [] \\
& \mathrm{otheractions}(\{shout, back\}, \Omega)
\end{aligned}
$$

$$
\begin{aligned}
CHAIR_2 :=\ & shout;\ CHAIR_2 \\
& [] \\
& back;\ RREI_2 \\
& [] \\
& \mathrm{otheractions}(\{shout, back\}, \Omega)
\end{aligned}
$$

$$
\begin{aligned}
CHAIR_3 :=\ & shout;\ CHAIR_3 \\
& [] \\
& back;\ RREI_3 \\
& [] \\
& \mathrm{otheractions}(\{shout, back\}, \Omega)
\end{aligned}
$$

Figure 30: Round-robin with executive interrupt in LOTOS

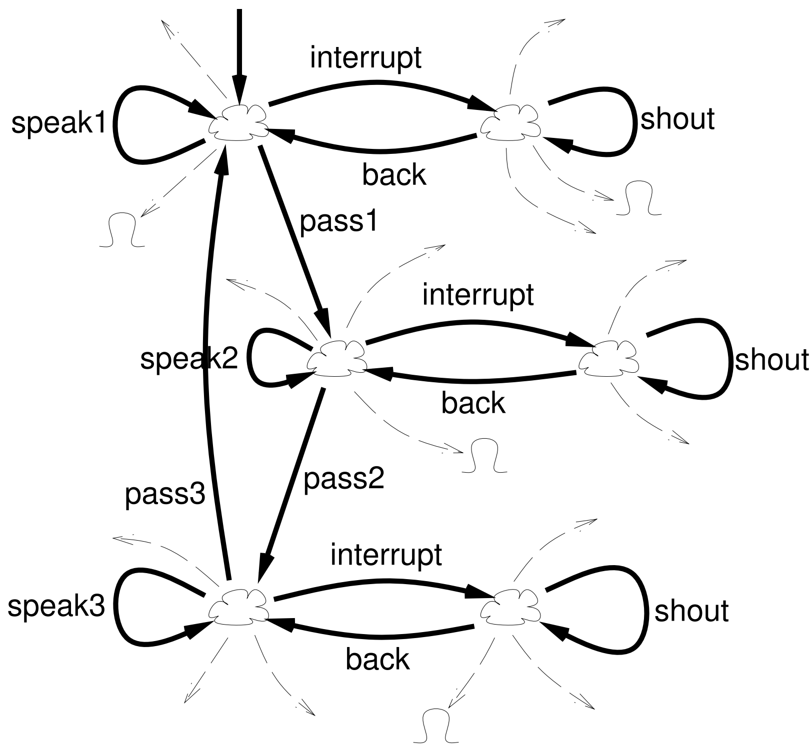

Figure 31: Labelled transition system for round-robin with executive interrupt

From this difference, we formulated Ω, a LOTOS process exhibiting “unspecified” behaviour, and used it to modify our translation to better reflect the meaning of the original state-based specification. In particular, wherever the state-based specification was not specified to be able to perform an operation, the translated LOTOS process would be able to perform the corresponding action and thus evolve to Ω, so emulating the implicit behaviour of the original state-based specification. We examined some examples, and saw that with this approach we could take some state-based specifications related by subtyping, translate them informally into process algebra using Ω, and the translated specifications were related by reduction.

The technique discussed in this chapter of adding “undefined” behaviour to a process has a serious problem for use in practical situations, which is that it is not compositional. That is, if two specifications have been translated using our Ω technique, it is not clear whether it will be possible to meaningfully compose them together in parallel; the extra branches of “undefined” behaviour are likely to result in unwanted deadlocks or other unusual behaviour in the resulting process.

In the next chapter, we will examine a particular problem which arises if we consider the data passed to and from operations, particularly the result values returned from an operation. We will see some possible ways of overcoming this problem, and how this, too, affects our translation. We will then have enough experience with the problems involved in translating state based specifications to LOTOS to be able to define a formal translation.

Chapter 5

Representing nondeterministic output in LOTOS

In the previous chapter, we examined the LOTOS development relations and decided that the reduction relation seemed appropriate for use as a subtyping relation. In the same way as the state-based subtyping relation ensures that an object of a subtype cannot do anything which conflicts with the supertype specification, reduction ensures that a LOTOS process cannot do anything which a less-reduced process definition would prevent.

We saw that, in combination with our simple translation of §3.4, reduction did not relate specifications which had been translated from state-based specifications related by subtyping. We addressed this problem by identifying a difference in interpretation between state-based specifications and process algebraic specifications. We then adjusted our translation to account for the difference, and saw that, for several pairs of example state-based specifications related by subtyping, their new translations were related by reduction, as we wanted.

In this chapter we will examine another problem, and will adjust our translation again to cope with it. We will see that the way we are encoding the return value from operations into the actions used in our LOTOS translations can, in some situations, enable the LOTOS process's environment to select the operation's return value. As well as being counter-intuitive, this effect prevents some examples of subtyping from translating into processes related by reduction.

In this chapter, we will examine several possible ways to prevent a process's environment determining the output of a LOTOS process. We suggest how we can change the semantics of the LOTOS "!" output construct, and how we can introduce a CSP-like nondeterministic choice operator into LOTOS. While we will not use any of our modified LOTOS semantics in our translated processes, they will inspire the changes we need to solve the problem with our translation.

5.1 Translating with data

In §4.2.1, we defined a translation from state-based specifications to process algebra which did not include a translation of data (that is, argument and return values). Instead, the translation simply included a transition in the translated specification when some argument and return values would satisfy the operation predicate. Then, in §4.5.1, we modified our translation to incorporate Ω, our representation of "unspecified" behaviour.

However, with neither of these versions of our translation did the stack, bag and queue specifications from §3.1.2 translate to distinguishable labelled transition systems. In §4.2.1, we showed that all three translated into labelled transition systems bisimilar to one another.

In this section, we will modify our translation to incorporate a representation of data into the translated process. We discussed methods of doing this in §3.4. We will use the simple method of combining the operation name with the argument and result to obtain the action name used in the labelled transition system.

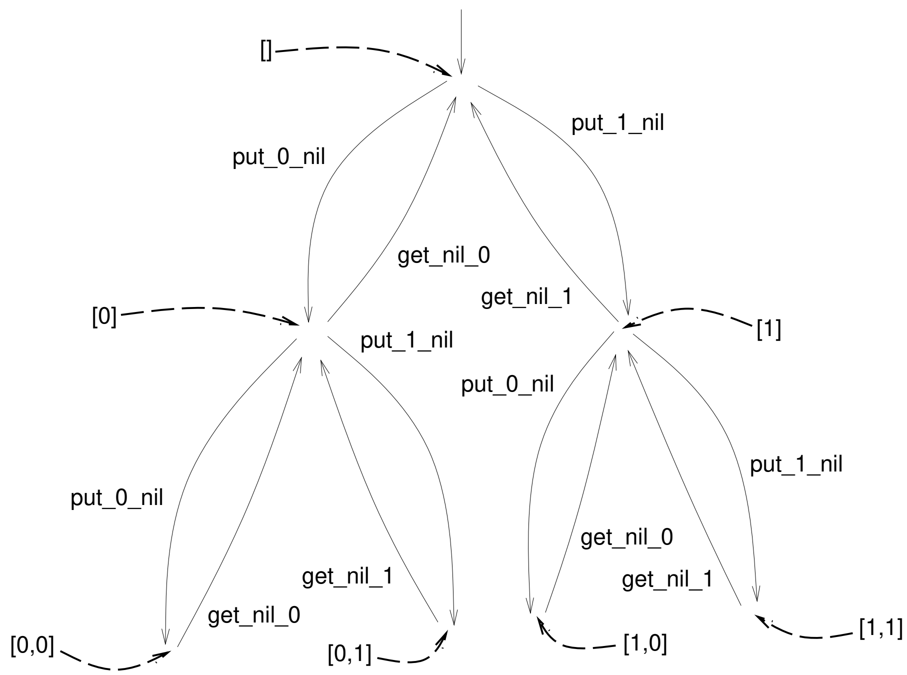

Figure 32: Translation of the stack specification

For simplicity, we will ignore the developments of §4.5.1, and base our translation on the simple translation of §4.2.1. We will again suggest an informal translation, with a view to combining our "unspecified data" technique with the developments in this chapter to create a formal translation in the next chapter. As before, we will show how this translation works by translating the "stack" specification from §3.1.2.

We translate each state in the state-based specification into a state in the labelled transition system in exactly the same way we did before. Our translated stack specification therefore has the same seven states it did when we translated it before: $\{[\,], [0], [0,0], [0,1], [1,0], [1], [1,1]\}$.

We then determine under what circumstances each operation can be performed: in which before-states, with which arguments and result values, and resulting in which after-states. We then generate a transition for each possibility from the state corresponding to the before-state to the state corresponding to the after-state. This time, however, we retain the information about which argument and result value we can perform each operation with: we do this by forming the label on the transition by concatenating the operation name, argument value and result value, separated by underscores.

Considering the initial state of stack, we can see that *get* cannot be performed, since the condition $before \neq [\,]$ is untrue. We can, however, perform *put*, either with argument 0, result *nil*, and after-state [0], or with argument 1, result *nil*, and after-state [1]. We therefore add two transitions to our labelled transition system; one labelled *put_0_nil* leading from [ ] to [0], and one labelled *put_1_nil* leading from [ ] to [1]. Using these new rules, we get the labelled transition system shown in figure 32.

We can now also incorporate the changes we made in §4.5 to incorporate Ω into our labelled transition system to represent the unspecified behaviour in our state-based specification. For every action in *Act*, and for each state in the labelled transition system which does not already have a transition leading away from it labelled with that action, we must add a transition from that state labelled with that action and leading to Ω. As before, we will represent this in our labelled transition system diagrams with a cloud at the node.

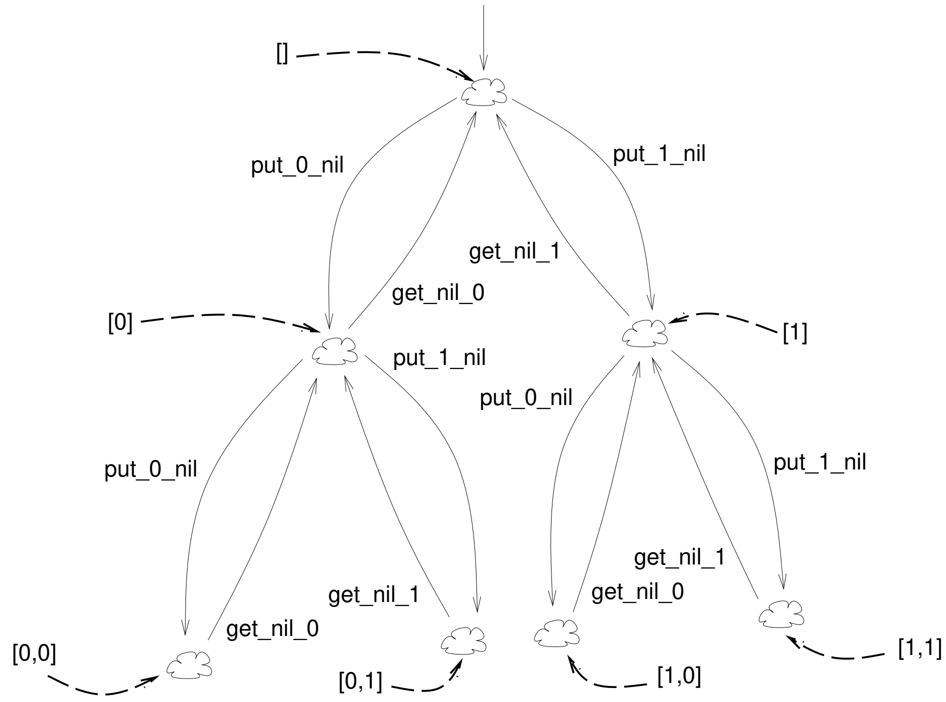

Figure 33: Translation of the stack specification

If we now translate our stack specification using both our "data" and "clouds" ideas together, we obtain the labelled transition system shown in figure 33. We have shown the labelled transition systems for bag and queue, translated in the same way, in figures 34 and 35 respectively.

5.1.1 Inadequacies in the translation

Let us examine the labelled transition systems in figures 33, 34 and 35. As we showed in §3.2.4, the stack and queue state-based specifications are each a subtype of the bag specification. We therefore want the translated stack and queue specifications each to be a reduction of the translated bag specification.

Unfortunately, it is easy to show that this is not the case. While the translated stack and queue each have less traces than the translated bag, they both have more refusals. For instance, stack can perform the trace $\langle put\_1\_nil, put\_0\_nil, get\_nil\_1\rangle$ and then refuse the action *put_0_nil*, while bag cannot. Queue can perform $\langle put\_1\_nil, put\_0\_nil, get\_nil\_0\rangle$ then refuse *put_0_nil*, something which bag cannot do. We will therefore need to change our translation again if we are to use reduction as our subtyping relation.

The reason this happens is because of a difference in interpretation between the state-based world and the translated LOTOS world. The difference between the two actions we considered, one of which can be performed by each of stack and queue, and both of which can be performed by bag, is in the return value portion of the action name. In the state reached after the trace $\langle put\_1\_nil, put\_0\_nil\rangle$, stack can perform *get_nil_0*, queue can perform *get_nil_1*, and bag can perform either action. Because bag offers both actions to its environment, it can refuse neither.

Let us consider the state-based specifications in analogous situations. If stack were in the state $[0, 1]$ when the *get* operation was invoked with argument *nil*, the only return value possible would be 1. Consider, however, bag in state $\{0, 1\}$. If we then invoke the *get* operation, passing *nil* as the argument, we might receive a return value of either 0 or 1, depending on how the

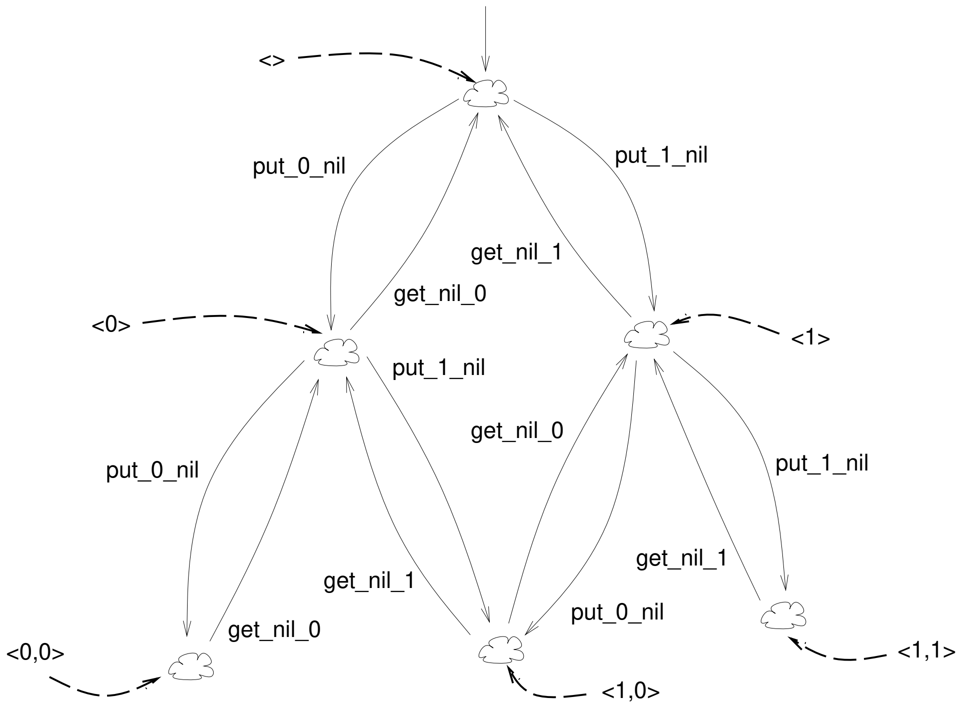

Figure 34: Translation of the bag specification

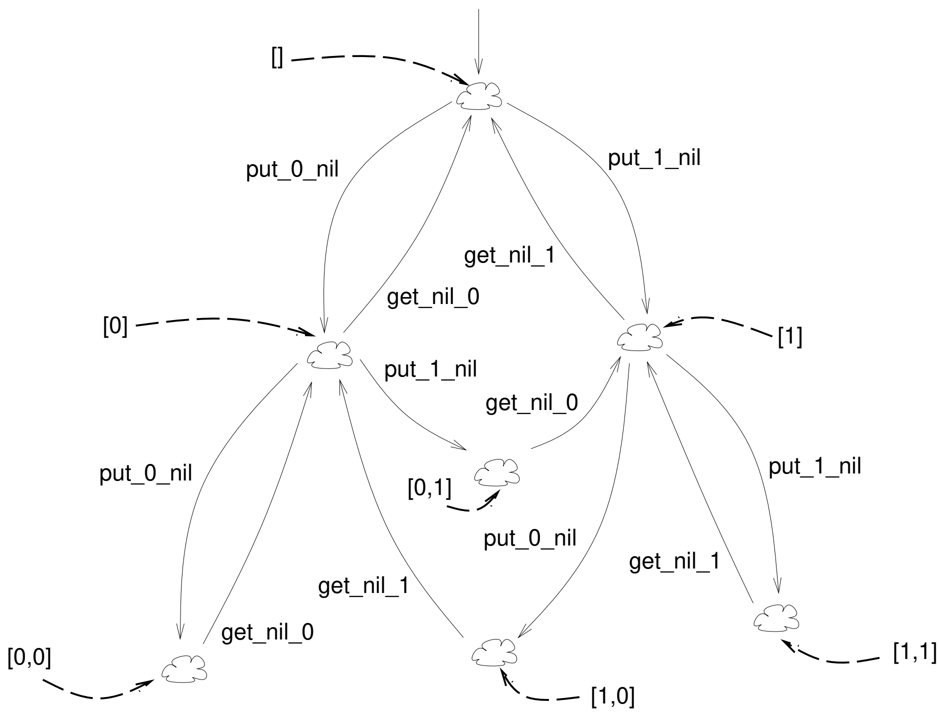

Figure 35: Translation of the queue specification

specification had been implemented. Either would comply with the specification, and the object might consistently return one or the other of the possible return values, or it might even pick one at random. The significant observation here is that the environment of the state-based bag specification is not able to influence which of the two possible return values, 0 or 1, is returned. That is, in a typical implementation, the choice of return value is made by the bag implementation, and cannot be influenced by the caller of the *get* method.

So, we have two different interpretations of the nondeterminism in the *get* method's return value. The state-based specification can clearly be implemented such that the return value is not subject to influence by the environment. In our LOTOS translation, however, the return value may be entirely determined by the environment; the bag process is required to offer an action corresponding to each possible return value.

This difference in interpretation indicates how we must change our translation. We must ensure that the choice of return value is made nondeterministically within the bag process, so that it cannot be influenced by the environment. However, we must ensure that we do not make the choice of operation or argument value nondeterministic; these are determined by the environment, which must be able to choose which operation it invokes and what argument it passes.

Over the course of this chapter, we will examine several attempts to solve this problem. In the next section, we will present a modified semantics for the subset of LOTOS which we have been using, and will thereby attempt to make LOTOS's "!" output operator produce *nondeterministic* output. We will then try a different alternative; we will introduce a nondeterministic choice operator, $\sqcap$, which will act like CSP's nondeterministic choice operator.

We show these modified versions of LOTOS not because we will use them to translate our state-based specifications into—after all, we are trying to find a subtyping relation for LOTOS, not for a variant of it—but because they provided the inspiration for the modification we will next make to our translation. In other words, the labelled transition systems generated by these variants of LOTOS are what we will be (loosely) attempting to emulate when we modify our translation for the last time in the next chapter.

5.2 Non-deterministic output

We will now attempt to produce a variant of LOTOS where all output is nondeterministic. That is, if a process has a choice between outputting two alternative values on the same gate, the process will not, as it would in standard LOTOS, offer both of the actions corresponding to the two outputs; rather, it will nondeterministically choose one or other of the output values and offer only that one to the environment. We should ensure that the choice will be nondeterministic, whatever any synchronised processes may do to try to influence the choice.

Consider these two process definitions:

$$P := a!1;\ B\ []\ a!3;\ C$$
$$Q := a?x : \{0, 1, 2\};\ D$$

(We should note that this definition of Q is not valid LOTOS, since LOTOS does not allow subtyping of data types. We could construct a valid, functionally identical example using LOTOS selection predicates, but that would obscure the intent of the example.)

P consists of a choice between two output actions; Q is a choice between three input actions. However, only one value, 1, "matches up" between the two; thus, $P\ |[a]|\ Q$ can only perform the action a_1 and evolve to $B\ |[a]|\ D$. P cannot choose to perform a_3 because there is no matching action in Q for it to synchronise with. This means that while P corresponds to an object whose method is being called, and Q corresponds to an object calling P's method, Q is determining which return value P returns. As described above, this can cause reduction not to hold between processes which are translations of state-based specifications which are related by subtyping.

The modified LOTOS semantics below takes the following approach to this problem. Firstly, we change the actions associated with data output, so that they are different from those used for input. We then change the semantics of parallel composition, such that, firstly, input actions and output actions can synchronise together in a parallel construct, if they have the same gate and data value, and secondly, any output action can always be performed, even if there is no corresponding input action for it to synchronise with. Intuitively, we can see that the proper consequences of an output action being performed when there is no corresponding input action for it to synchronise with, is deadlock, so this is what we will use.

Under these rules, the construct $P \mid[a]\mid Q$, with the definitions of P and Q above, would behave similarly to $(a!1;\ B\ []\ i;\ stop) \mid[a]\mid Q$. The output of 3 on gate a in the second branch of P's behaviour has no corresponding input in Q, so, effectively, P deadlocks. Any behaviour in Q which can still proceed with P deadlocked is not blocked by this.

To accomplish this, we will have to change the semantics of LOTOS. We now give inference rules for a modified subset of LOTOS implementing the above semantics for nondeterministic output.

We will give new inference rules for the following parts of the language's syntax: process definition of the form

$$P(\tilde{x} : \tilde{T}) := B$$

and the behaviours "$stop$", "$a!e;\ B$", "$a?x : T;\ B$", "$i;\ B$", "$B\ []\ C$", "$B \backslash G$", "$P(\tilde{e})$" and "$B \mid[G]\mid C$".

In the semantics presented here, we use three different forms of action name: $\xrightarrow{a\_n}$ is used to represent an input action, generated from an action prefix construct of the form $a?x : T;\ B$, $\xrightarrow{a\langle n\rangle}$ represents an output action, and is generated from action prefix constructs of the form $a!e;\ B$, and $\xrightarrow{i}$ is the internal action, as generated by hiding or an action prefix of the form $i;\ B$.

o is used to indicate any operation name. a denotes any observable action, that is, any action of the form o_n or $o\langle n\rangle$. α represents any action, including the internal action i.

$[\![e]\!]$ indicates the value obtained by evaluating the expression e. There is no need for an environment, because any variables in the expression will have had values substituted for them before the expression is evaluated. The syntax $B[x/n]$ denotes the behaviour B, but with the value n substituted for the variable name x.

We will also use a helper function, *opname*, which determines the operation name associated with an action. *opname* is defined as follows:

$$opname(o\langle n\rangle) = o$$
$$opname(o\_n) = o$$

We now present the inference rules for this LOTOS variant.

The first rule generates transitions with labels of the form $a\langle n\rangle$ from output action prefix constructs:

$$\frac{-}{o!e;\ B \xrightarrow{o\langle [\![e]\!]\rangle} B}$$

The next rule generates transitions with labels of the form a_n from input action prefix constructs:

$$\frac{-}{o?x : T;\ B \xrightarrow{o\_n} B[x/n]} \quad [\ n \in T\ ]$$

Internal actions, choice, hiding and process instantiation are handled as in normal LOTOS:

$$\frac{-}{i;\ B \xrightarrow{i} B} \qquad \frac{P \xrightarrow{\alpha} P'}{P\ []\ Q \xrightarrow{\alpha} P'} \qquad \frac{Q \xrightarrow{\alpha} Q'}{P\ []\ Q \xrightarrow{\alpha} Q'}$$

$$\frac{P \xrightarrow{a} P'}{P \backslash G \xrightarrow{i} P' \backslash G} \quad [\ opname(a) \in G\ ]$$

$$\frac{P \xrightarrow{\alpha} P'}{P \backslash G \xrightarrow{\alpha} P' \backslash G} \quad [\ opname(a) \notin G\ ]$$

$$\frac{B[\tilde{x}/[\![\tilde{e}]\!]] \xrightarrow{\alpha} B'}{P(\tilde{e}) \xrightarrow{\alpha} B'} \quad [\ P(\tilde{x} : \tilde{T}) := B\ ]$$

The interesting rules are the following, which all have to do with the parallel construct. The first two rules allow either of the two synchronised processes to perform internal actions independently of the other:

$$\frac{P \xrightarrow{i} P'}{P\ |[G]|\ Q \xrightarrow{i} P'\ |[G]|\ Q}$$

$$\frac{Q \xrightarrow{i} Q'}{P\ |[G]|\ Q \xrightarrow{i} P\ |[G]|\ Q'}$$

The next two rules allow either of the two synchronised processes to independently perform any action which is not governed by the parallel construct's gate set:

$$\frac{P \xrightarrow{a} P'}{P\ |[G]|\ Q \xrightarrow{a} P'\ |[G]|\ Q} \quad [\ opname(a) \notin G\ ]$$

$$\frac{Q \xrightarrow{a} Q'}{P\ |[G]|\ Q \xrightarrow{a} P\ |[G]|\ Q'} \quad [\ opname(a) \notin G\ ]$$

The next four rules determine how actions governed by the parallel construct's gate set can synchronise with one another. If the two parallel processes can both perform the same action, they may both perform that action simultaneously, as usual. However, it is also possible for an action of the form $a\langle n\rangle$ to synchronise with an action of the form a_n; this is what allows inputs to synchronise with outputs. When one of the parallel processes performs an input action and the other performs a corresponding output action, the combined process performs the output action. This is done because of the intuitive idea that output is a more "active" action that input; output actions should therefore propagate through parallel constructs preferentially to input actions.

$$\frac{P \xrightarrow{o\_n} P',\ Q \xrightarrow{o\_n} Q'}{P\ |[G]|\ Q \xrightarrow{o\_n} P'\ |[G]|\ Q'} \quad [\ o \in G\ ] \qquad r_1(o, n)$$

$$\frac{P \xrightarrow{o\_n} P',\ Q \xrightarrow{o\langle n\rangle} Q'}{P\ |[G]|\ Q \xrightarrow{o\langle n\rangle} P'\ |[G]|\ Q'}\ [\ o \in G\ ] \qquad r_2(o,n)$$

$$\frac{P \xrightarrow{o\langle n\rangle} P',\ Q \xrightarrow{o\_n} Q'}{P\ |[G]|\ Q \xrightarrow{o\langle n\rangle} P'\ |[G]|\ Q'}\ [\ o \in G\ ] \qquad r_3(o,n)$$

$$\frac{P \xrightarrow{o\langle n\rangle} P',\ Q \xrightarrow{o\langle n\rangle} Q'}{P\ |[G]|\ Q \xrightarrow{o\langle n\rangle} P'\ |[G]|\ Q'}\ [\ o \in G\ ] \qquad r_4(o,n)$$

The final two rules are concerned with creating internal actions in the cases where an output action cannot synchronise with a corresponding input action:

$$\frac{P \xrightarrow{o\langle n\rangle}}{P\ |[G]|\ Q \xrightarrow{i} \mathit{stop}\ |[G]|\ Q}\ [\ o \in G\ ] \qquad i_1(o,n)$$

$$\frac{Q \xrightarrow{o\langle n\rangle}}{P\ |[G]|\ Q \xrightarrow{i} P\ |[G]|\ \mathit{stop}}\ [\ o \in G\ ] \qquad i_2(o,n)$$

Notice that, on their own, these rules apply in many places where we would not want them to apply. To avoid this, the last six rules given are ordered, using the system given by Ulidowski and Phillips in [UP97]:

$$\begin{aligned}\forall\, o, n \bullet (&r_1(o,n) > i_1(o,n), i_2(o,n) \wedge r_2(o,n) > i_1(o,n), i_2(o,n)\\ &\wedge r_3(o,n) > i_1(o,n), i_2(o,n) \wedge r_4(o,n) > i_1(o,n), i_2(o,n))\end{aligned}$$

The ordering ensures that, for each possible transition label, rules i_1 and i_2 only apply if none of the rules r_1, r_2, r_3 or r_4 are applicable. Thus, if the left- and right-hand sides of a parallel composition can perform transitions with labels o_n and $o\langle n\rangle$ respectively for some particular o and n, then rule r_2 applies and their composition can perform the action $o\langle n\rangle$. Since rule r_2 applies, rules i_1 and i_2 are inhibited for the given o and n. If only *one* side of the parallel composition can perform an $o\langle n\rangle$ transition, and the other side cannot perform either $o\langle n\rangle$ or o_n, then rules r_1, r_2, r_3 and r_4 cannot apply, and rule i_1 or i_2 will generate a transition labelled i corresponding to the possibility that the process with the unmatched output transition could "commit" to performing the output and so, due to the lack of an appropriate action to synchronise with, evolve to stop. Notice that the other side of the parallel construct may continue to evolve if it has any alternative transitions to make.

5.2.1 Examples

In this subsection, we will present some small example process definitions to show how the above semantics work in practice.

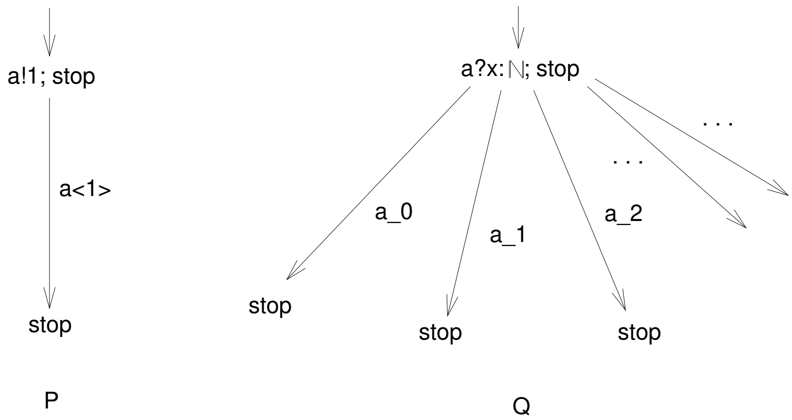

Figure 36: Labelled transition systems for processes P and Q

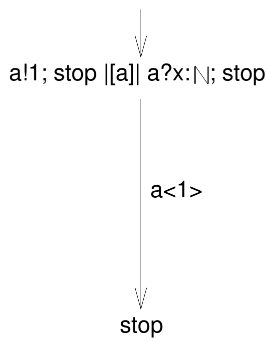

Figure 37: Labelled transition system for process $P\ |[a]|\ Q$

First, let us examine the following process definitions:

$P := a!1;\ \mathit{stop}$
$Q := a?x : \mathbb{N};\ \mathit{stop}$

We have generated labelled transition systems for each of these processes using the rules given above. These labelled transition systems are shown in figure 36. Both of these processes' labelled transition systems are very similar to their labelled transition systems in the normal LOTOS semantics, with the exception that P's transition is labelled $a\langle 1\rangle$ instead of a_1.

When we examine the labelled transition system for the process $P\ |[a]|\ Q$ in figure 37, we see that placing P and Q in parallel has results similar to those in standard LOTOS. While in standard LOTOS, the two processes placed in parallel must both be able to perform exactly the same action in order to progress, our variant of LOTOS with nondeterministic output allows an action of the form a_n to synchronise with an action of the form $a\langle n\rangle$. The input actions of Q which do not correspond to an action P can perform cannot be performed in the combined process, as in standard LOTOS.

Let us now consider process R:

$R := a!3;\ \mathit{stop}\ []\ a!(-2);\ \mathit{stop}$

The labelled transition system for R is shown in figure 38. One of the values that R can output

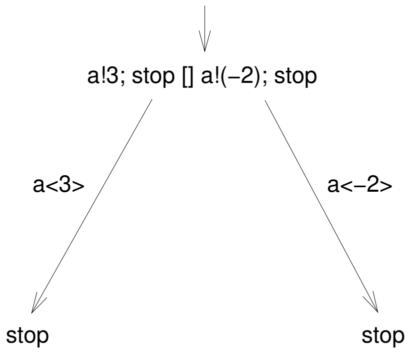

Figure 38: Labelled transition system for process R

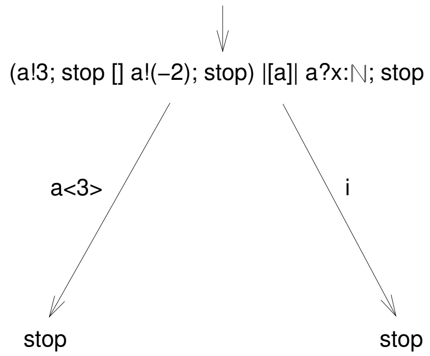

Figure 39: Labelled transition system for $R\ |[a]|\ Q$

on gate a is 3, a member of $\mathbb{N}$; the other is -2, which is not.

(We should note at this point that the process $R\ |[a]|\ Q$ is not valid LOTOS, since the type of the data passed on a in Q is $\mathbb{N}$, whilst in R the value -2, which is not a member of $\mathbb{N}$, is passed. For this reason, the example does not type check. In examining this example, however, we will ignore this problem, and imagine that the example is valid LOTOS. Similar, but valid, examples can be constructed by using LOTOS selection predicates; such examples do not, however, so effectively convey the notion that we are dealing with overlapping data types.)

If we place R in parallel with Q, we can see the intended effect of our modified semantics. Figure 39 shows the labelled transition system of $R\ |[a]|\ Q$. We can see that our variant of LOTOS behaves rather differently than normal LOTOS. In normal LOTOS, $R\ |[a]|\ Q$ would be able to perform exactly one action, a_3, and then deadlock. With our modified semantics, however, $R\ |[a]|\ Q$ is strongly bisimilar to $a!3;\ \mathit{stop}\ []\ i;\ \mathit{stop}$.

Why is this? Referring to the inference rules above, we can see that there are two groups of rules, r_n and i_n, for the parallel composition construct. If, for some values of o and n, any rule in the first group applies, the rules in the second group are prevented, by rule ordering, from being applied. With $o = a$ and $n = 3$, the conditions of rule r_3 hold, and a transition labelled $a\langle 3\rangle$ is generated. For $n = -2$, however, none of the rules r_1, r_2, r_3 or r_4 holds, so rules i_1 and i_2 can be tried. Rule i_1's condition holds, so a transition labelled i is generated. For any other value of n, neither set of rules generates any transitions, so we get exactly these two transitions in our final labelled transition system.

Let us now return to our example of the bag specification. The standard LOTOS development relations, including reduction, characterise various notions of testing[vG90]. The tests they

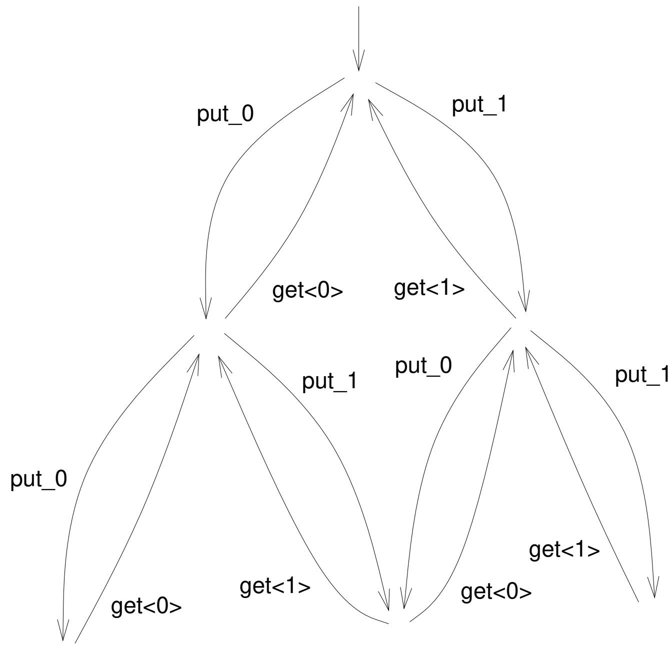

Figure 40: Bag specification with data (modified)

characterise depend on a model of observation, which in turn depends on the semantics of the parallel composition construct. Because we have changed the semantics of parallel composition, reduction is unlikely to be meaningful when used with our modified semantics. For that reason, we will use a tester process to examine the behaviour of the bag specification in this semantics, rather than using the LOTOS definition of reduction.

Figure 40 shows our bag specification, written as a labelled transition system with data. However, for clarity, we have removed the *nil* values, so that each operation has only either an argument or a return value. Where an action's data item is an argument, we have used the input form: o_n. Where an action's data item is a result value, we have used the output form: $o\langle n\rangle$. We have also shown, in figure 41, a process which we will use to test the bag process.

Figure 42 shows the resulting labelled transition system when the labelled transition systems for the bag and the tester are placed in parallel, fully synchronised. Transition A is generated because after the bag process has performed the action put_0, synchronised with the tester's $put\langle 0\rangle$, the bag can then perform $get\langle 0\rangle$. Since there is no corresponding action for it to synchronise with, an internal action is generated. Transition B is generated similarly; after performing the trace $\langle put\_0, put\_1\rangle$, the bag can perform either $get\langle 0\rangle$ or $get\langle 1\rangle$. While $get\langle 0\rangle$ can synchronise with the corresponding get_0 action in the tester, the $get\langle 1\rangle$ action cannot, and so an internal action is generated.

In this case, transition B is what we want, but transition A is not. We want the choice between the two output actions $get\langle 0\rangle$ and $get\langle 1\rangle$ to be made nondeterministically within the bag process; this is how we interpret transition B. Transition A, however, is interpreted as the bag process nondeterministically "committing" to the get_0 action; but the tester did not attempt to "invoke the *get* operation", so this behaviour is inappropriate.

5.2.2 Conclusions

We have presented a semantics for a modified version of the LOTOS specification language, in which choices between output actions are made nondeterministically, and have shown with some

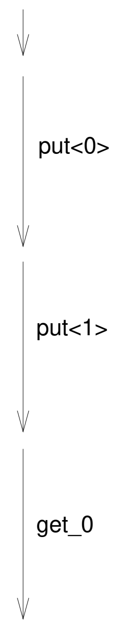

Figure 41: Tester process

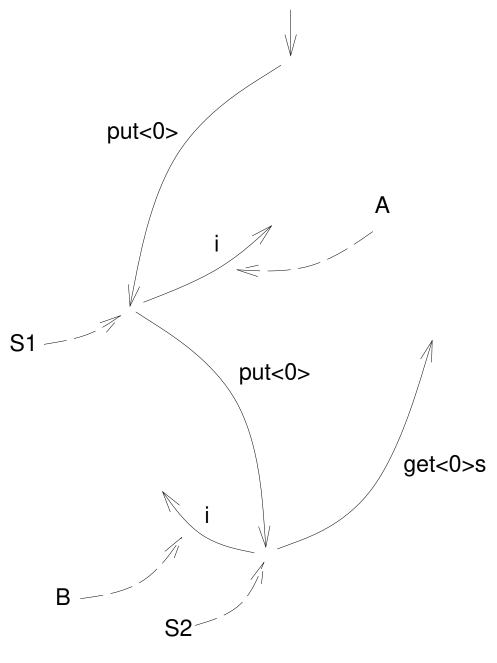

Figure 42: Result of placing bag specification and tester in parallel, fully synchronised

examples how the resulting language works. As a target language for translating state-based specifications into process algebra, however, our language is lacking.

Although it has the nondeterministic output we needed, we were looking for evidence to suggest that reduction would be suitable as a process algebraic subtyping relation. Because reduction is defined in terms of refusals, and refusals are a characterisation of the way a process can observe another process with which it is composed in parallel, any change to the semantics of parallel composition can significantly affect the meaning of reduction. In this case, we have changed the semantics of parallel composition to such an extent that reduction no longer has much meaning. We therefore find that this variant of LOTOS is not of use to us.

We will next look at another modified version of LOTOS; one which might be more generally useful.

5.3 Non-deterministic choice operator

In the previous section, we considered an alternative semantics for LOTOS which incorporated nondeterministic output; that is, whenever a process was able to perform an output action, it could nondeterministically "commit" to that output. As a result, placing processes in parallel could result in deadlocks being introduced where one process could perform an output action, but the other could not perform a matching input or output action.

We found a number of disadvantages to the approach given above. We found that, while the modified semantics introduced deadlocks we wanted, it also introduced unwanted deadlocks. More seriously, due to our modification of the semantics of the parallel composition construct, the standard LOTOS reduction relations could no longer reliably be used with our LOTOS variant.

Since the effect we want is that of a nondeterministic choice between a number of behaviours whose initials are output actions, it would seem appropriate to simply implement a nondeterministic choice operator in LOTOS. The operator should be similar to the normal LOTOS choice operator, but should make its choice entirely locally to the process. That is, the fact that the choice operator has been used should not be observable outside the process; the behaviour should simply be that of one of the two choices. CSP has such a choice operator: $\sqcap$, or "internal choice". We will now attempt to implement $\sqcap$ in LOTOS.

We will now examine some examples of the use of the $\sqcap$ operator and show behaviours in standard LOTOS which we would like them to be equivalent to. In this way, we will attempt to determine how our nondeterministic choice operator should behave and how to implement it.

Firstly, consider this behaviour:

$$a;\ A \sqcap b;\ B$$

This simple nondeterministic choice between two behaviours should obviously be equivalent to the following standard LOTOS behaviour:

$$i;\ a;\ A\ [\,]\ i;\ b;\ B$$

Next, consider what happens if we use the $\sqcap$ operator next to the normal choice operator, as follows:

$$a;\ A\ [\,]\ (b;\ B \sqcap c;\ C)$$

If we follow the pattern of our first example, this would be equivalent to the following standard LOTOS behaviour:

$$a;\ A\ [\,]\ (i;\ b;\ B\ [\,]\ i;\ c;\ C)$$

This is incorrect, because this behaviour offers a initially, but then, at any time, may perform an internal action and begin offering either b or c. The intent of the original behaviour was that

a would be offered together with either b or c; and the choice between b and c would be made nondeterministically.

To get a LOTOS process which is correctly equivalent to the version with nondeterministic choice, we first distribute the deterministic choice over the nondeterministic choice, as follows:

$$(a;\ A \ [\!]\ b;\ B) \sqcap (a;\ A \ [\!]\ c;\ C)$$

We can then use internal actions to convert the $\sqcap$ operator into a standard LOTOS-style nondeterministic choice.

$$i;\ (a;\ A \ [\!]\ b;\ B) \ [\!]\ i;\ (a;\ A \ [\!]\ c;\ C)$$

This behaviour is equivalent to the intended meaning of $a;\ A \ [\!]\ (b;\ B \sqcap c;\ C)$. It offers the action a, together with either b or c.

5.3.1 Inference rules for nondeterministic choice

Now that we have seen some examples of the way we would like our nondeterministic choice operator to behave, we will attempt to formalise our ideas and present inference rules for a variant of the LOTOS language incorporating such a nondeterministic choice operator.

In order to code up our nondeterministic choice operator, we will make use of a new kind of transition. Apart from the observable actions, LOTOS has a single kind of internal action, i. We will introduce a second kind of internal action, distinct from i. We will call this new kind of internal action τ, because it behaves similarly to the τ action in CSP.

In our example above, we saw that, in isolation, the behaviour $a;\ A \sqcap b;\ B$ should behave like the standard LOTOS behaviour $i;\ a;\ A \ [\!]\ i;\ b;\ B$. Our inference rules handling $\sqcap$, then, will be much like the rules for standard LOTOS's choice operator, but will generate our new τ internal action instead of i.

$$\frac{-}{P \sqcap Q \xrightarrow{\tau} P}$$

$$\frac{-}{P \sqcap Q \xrightarrow{\tau} Q}$$

The τ transitions generated by the above rules behave as the normal LOTOS internal actions, except that they "push through" the LOTOS choice operator, as specified by the next two rules:

$$\frac{P \xrightarrow{\tau} P'}{P \ [\!]\ Q \xrightarrow{\tau} P' \ [\!]\ Q}$$

$$\frac{Q \xrightarrow{\tau} Q'}{P \ [\!]\ Q \xrightarrow{\tau} P \ [\!]\ Q'}$$

The following two rules are similar to the normal LOTOS rules for choice, and allow any action other than τ to resolve the choice.

$$\frac{P \xrightarrow{b} P'}{P \ [\!]\ Q \xrightarrow{b} P'} \quad [\ b \in Act \cup \{i\}\ ]$$

$$\frac{Q \xrightarrow{b} Q'}{P \; [] \; Q \xrightarrow{b} Q'} \quad [\; b \in Act \cup \{i\} \;]$$

Other than the above rules for $\sqcap$ and $[]$, τ is treated the same way as i. The rest of the semantics are therefore very similar to the standard LOTOS semantics. In the following, a is drawn from the set Act and α is drawn from $Act \cup \{i, \tau\}$.

Action prefix operates as it does in standard LOTOS. We do not generate the actions of the form $o\langle n \rangle$ which we used in the section above; this is a different approach, not a development of the previous one. Note that, although it is possible to write i explicitly in standard LOTOS, we have not given an inference rule to allow τ to be written explicitly. This is consistent with τ's treatment in CSP, where τ cannot be written explicitly.

$$\frac{-}{o!e;\; B \xrightarrow{o\text{—}[\![e]\!]} B}$$

$$\frac{-}{o?x : T;\; B \xrightarrow{o\text{—}n} B[x/n]} \quad [\; n \in T \;]$$

$$\frac{-}{i;\; B \xrightarrow{i} B}$$

A set of gates may be hidden; doing so transforms actions on any of those gates into a LOTOS-style internal action, i.

$$\frac{P \xrightarrow{o\text{—}n} P'}{P \backslash G \xrightarrow{i} P' \backslash G} \quad [\; o \in G \;]$$

$$\frac{P \xrightarrow{o\text{—}n} P'}{P \backslash G \xrightarrow{o\text{—}n} P' \backslash G} \quad [\; o \notin G \;]$$

Process instantiation is unaffected by our changes; its rule is just as for LOTOS:

$$\frac{B[\tilde{x}/[\![\tilde{e}]\!]] \xrightarrow{\alpha} B'}{P(\tilde{e}) \xrightarrow{\alpha} B'} \quad [\; P(\tilde{x} : \tilde{T}) := B \;]$$

Parallel composition is also not affected by our changes. τ is treated the same as i, in that i and τ both "push through" parallel composition.

$$\frac{P \xrightarrow{x} P'}{P \;|[G]|\; Q \xrightarrow{x} P' \;|[G]|\; Q} \quad [\; x \in \{i, \tau\} \;]$$

$$\frac{Q \xrightarrow{x} Q'}{P \;|[G]|\; Q \xrightarrow{x} P \;|[G]|\; Q'} \quad [\; x \in \{i, \tau\} \;]$$

$$\frac{P \xrightarrow{o\text{—}n} P'}{P \;|[G]|\; Q \xrightarrow{o\text{—}n} P' \;|[G]|\; Q} \quad [\; o \notin G \;]$$

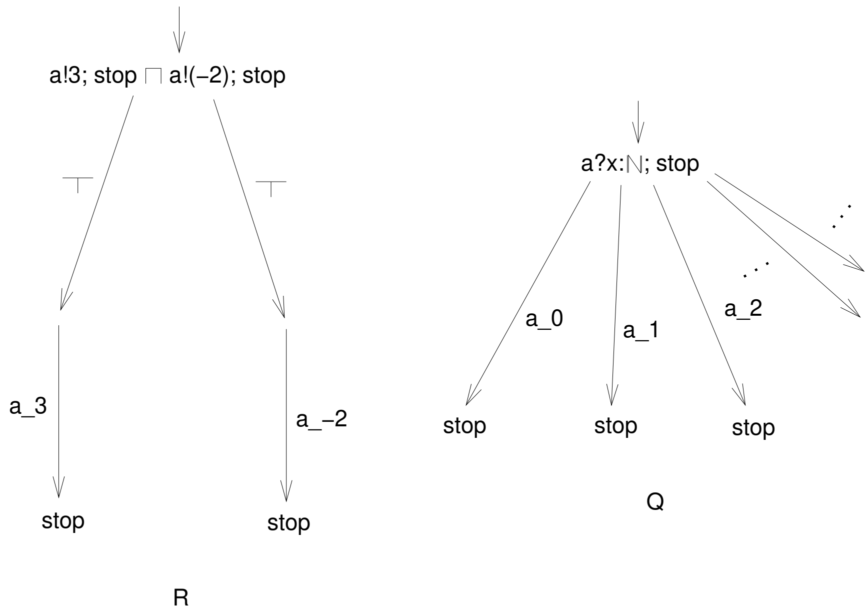

Figure 43: Labelled transition systems for R and Q

$$\frac{Q \xrightarrow{o\_n} Q'}{P \;|[G]|\; Q \xrightarrow{o\_n} P \;|[G]|\; Q'} \quad [\, o \notin G \,]$$

$$\frac{P \xrightarrow{o\_n} P',\, Q \xrightarrow{o\_n} Q'}{P \;|[G]|\; Q \xrightarrow{o\_n} P' \;|[G]|\; Q'} \quad [\, o \in G \,]$$

5.3.2 Examples

By way of some examples, we will now show how the nondeterministic choice operator works.

When we discussed our nondeterministic output semantics above, we gave an example involving processes R and Q. We will now rewrite R to use our nondeterministic choice operator, giving the following definitions:

$$R := a!3;\ stop \sqcap a!(-2);\ stop$$
$$Q := a?x : \mathbb{N};\ stop$$

(This example comes with the same caveat as in the previous section: it fails LOTOS's static type check.) The labelled transition systems for R and Q are shown in figure 43. Notice that R now has two extra τ transitions, which together form the nondeterministic choice. R cannot offer both a_3 and a_-2 at once; rather, it must first choose which τ transition to take. Only then can it offer one or the other action.

We can now combine R and Q together with a parallel construct. Figure 44 shows the labelled transition system of the resulting process, $R\;|[a]|\;Q$. Notice that $R\;|[a]|\;Q$ is structured as a choice between two alternatives, and that the process must choose one of those alternatives before it can offer any actions. The left-hand branch corresponds to the case where R outputs 3 on gate a; the right-hand branch corresponds to the case where R outputs -2 on a. In standard LOTOS, the right-hand branch would disappear when we placed R and Q in parallel, because the action a_-2 has no matching action in Q to synchronise with; in our variant of LOTOS, however,

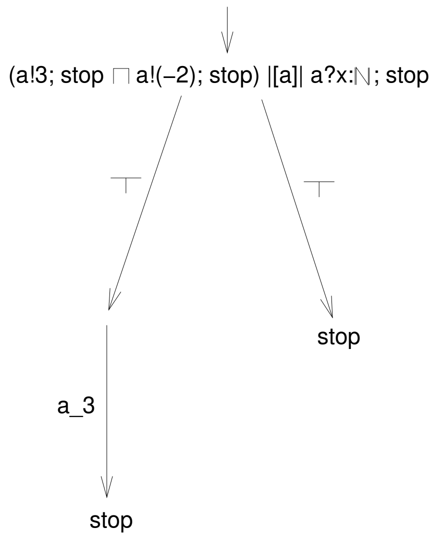

Figure 44: Labelled transition system for $R \; |[a]| \; Q$

the nondeterministic choice forces the branch to remain, even though the behaviour which occurs is equivalent to *stop*. This is the effect we want in our translation: a nondeterministic choice between output actions.

We can see that nondeterministic choice works as we want when it is placed on its own on one side of a parallel construct. However, we could have obtained the same effect using a traditional LOTOS-style nondeterministic choice behaviour such as $i;\ A\ [\,]\ i;\ B$. We will now examine an example which shows how our nondeterministic choice operator can produce effects which are more difficult with the traditional LOTOS encoding of nondeterministic choice.

Let us suppose that a process, P, wishes to interact on one of three gates, a, b and c. It wishes to leave the choice of gate up to its environment, but if the environment chooses to interact on gate a, our process wishes to output either the value 1 or the value 2, and wishes to choose the value nondeterministically, so that the environment cannot influence the choice by restricting the range of actions it offers. Similarly, our process can output 2 or 3 on gate b, and will choose which value nondeterministically. On gate c, our process is only willing to output the value 4.

In general, a specification such as the one in the previous paragraph requires that our process should first pick one value each out of the set of values it can output on each gate—in this case, either 1 or 2 for gate a, either 2 or 3 for gate b and 4 for gate c—and then offer the corresponding set of actions. For example, our process could pick 1 for gate a, 3 for gate b and 4 for gate c; this would result in its offering the actions a_1, b_3 and c_4 to its environment.

In standard LOTOS, we would have to construct a behaviour such as the following to represent our process. P_1's labelled transition system is shown in figure 45.

$$\begin{aligned} P_1 := {} & i;\ (a!1;\ \mathit{stop}\ [\,]\ b!2;\ \mathit{stop}\ [\,]\ c!4;\ \mathit{stop}) \\ & [\,]\ i;\ (a!1;\ \mathit{stop}\ [\,]\ b!3;\ \mathit{stop}\ [\,]\ c!4;\ \mathit{stop}) \\ & [\,]\ i;\ (a!2;\ \mathit{stop}\ [\,]\ b!2;\ \mathit{stop}\ [\,]\ c!4;\ \mathit{stop}) \\ & [\,]\ i;\ (a!2;\ \mathit{stop}\ [\,]\ b!3;\ \mathit{stop}\ [\,]\ c!4;\ \mathit{stop}) \end{aligned}$$

Using our nondeterministic choice operator, however, we can write the behaviour as follows:

$$P_2 := (a!1;\ \mathit{stop} \sqcap a!2;\ \mathit{stop})\ [\,]\ (b!2;\ \mathit{stop} \sqcap b!3;\ \mathit{stop})\ [\,]\ c!4;\ \mathit{stop}$$

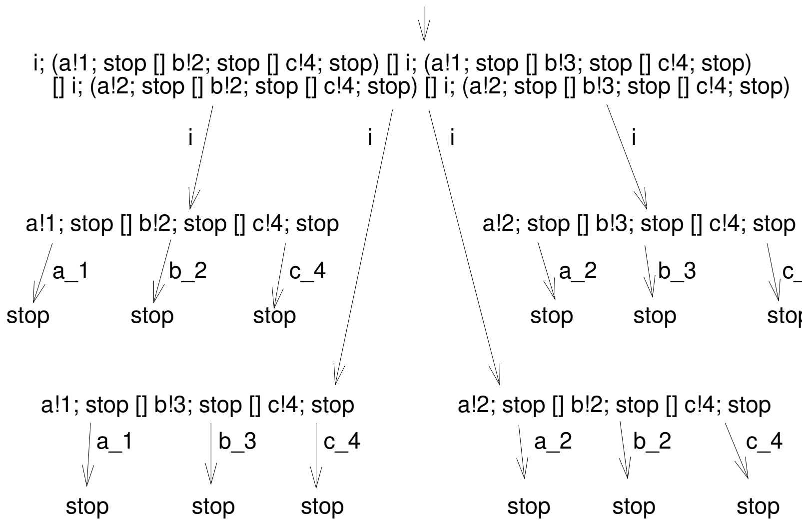

Figure 45: Labelled transition system for P_1

This behaviour produces the (rather large) labelled transition system shown in figure 46.

We can see from this labelled transition system that the nondeterministic choice operator works the way we want it to. When the process is started, it can perform τ actions (which, being a kind of internal action, will "push through" the parallel composition operator) to reach a number of states. In any one of those states, it is possible to perform an action on any one of the three gates a, b or c which the process can interact on, or to perform more τ actions to reach such a state. When an externally observable action occurs, the environment can decide which gate the action occurs on, but does not get to influence the choice of value to be passed: in each state where the process cannot perform τ, it can perform exactly one action on each of the three gates.

P_1 and P_2 are not strongly bisimilar (see §4.3.7); P_2 has several states which can perform either τ or some observable action, while P_1 has no such state.

However, P_1 and P_2 are testing equivalent, if we change the definition of $\Rightarrow$ such that τ is treated as an internal action, like i, and redefine Tr(), *after*, Ref() and $=_{te}$ according to this new definition of $\Rightarrow$. Both P_1 and P_2 can perform the following traces:

$$\langle\,\rangle, \langle a\_1\rangle, \langle a\_2\rangle, \langle b\_2\rangle, \langle b\_3\rangle, \langle c\_4\rangle$$

Taking $Act = \{a\_1, a\_2, b\_2, b\_3, c\_4\}$, the refusals of P_1 and P_2 after the empty trace are:

$$\{\}, \{a\_1\}, \{a\_2\}, \{b\_2\}, \{b\_3\},$$
$$\{a\_1, b\_2\}, \{a\_1, b\_3\}, \{a\_2, b\_2\}, \{a\_2, b\_3\}$$

The refusals of P_1 and P_2 after any nonempty trace are $\mathbb{P}\,Act$. Since the traces and refusals of P_1 and P_2 are the same, the two are testing equivalent. This gives some extra evidence that our definition of $\sqcap$ makes sense.

5.3.3 Comparing the two varieties of internal action

The major difference between i and τ is that τ will propagate through an external choice, whereas i cannot: it will, instead, resolve the choice.

The distinction between i and τ only manifests itself when one of these actions is an initial action of a process (as represented as a labelled transition system). When a process containing

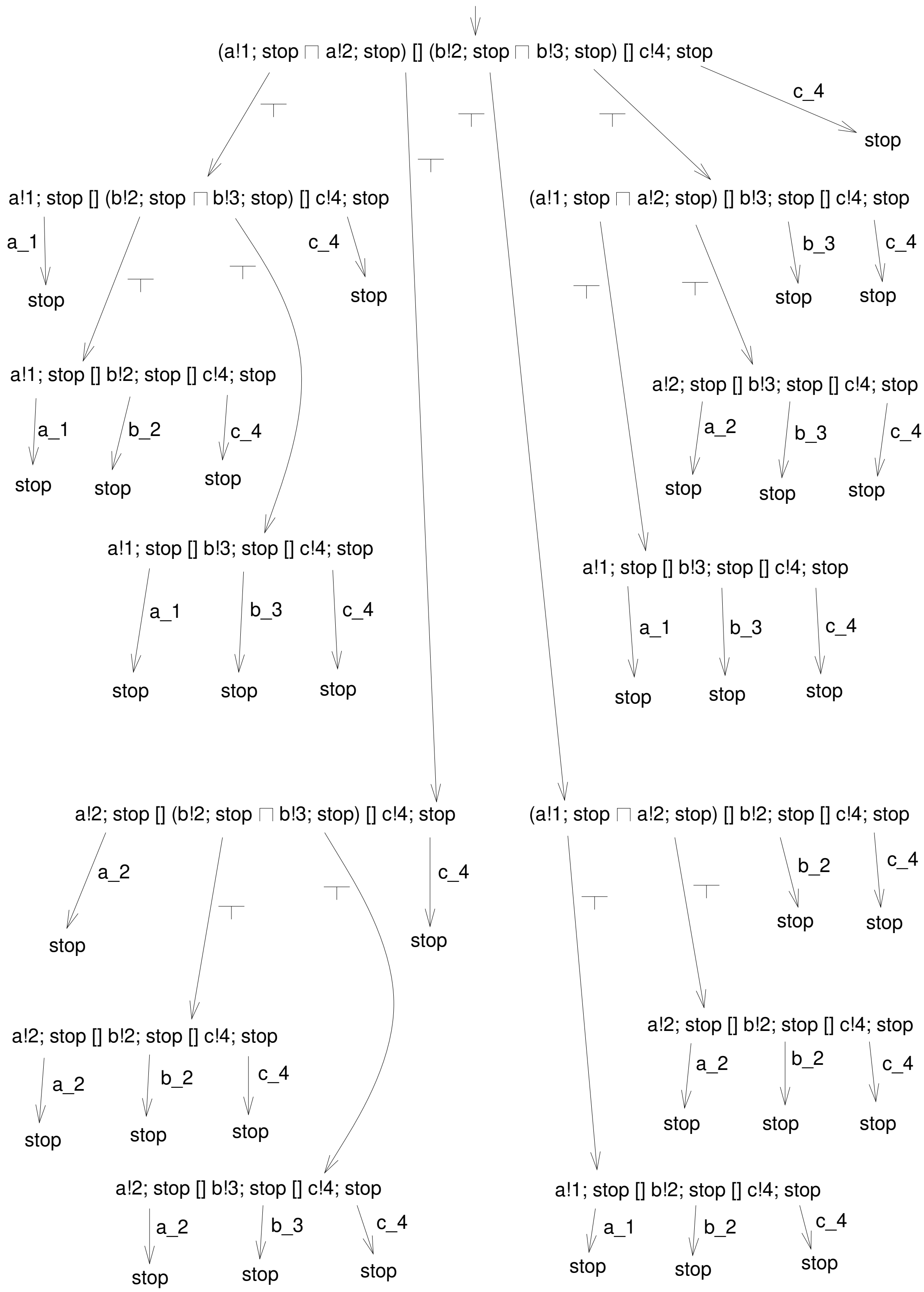

Figure 46: Labelled transition system for P_2

a τ as an initial action is combined with another process using the choice operator, the τ may push through the choice and become an initial of the combined process. However, if a process does not have a τ as one of its initials, any τs within the process will act in precisely the same way as is. This applies to any τs which are no longer the initials of a process; effectively, when a process containing an initial τ is combined in such a way that the τ does not become an initial of the new process, the τ may thereafter be treated as if it were an i.

These different behaviours are the manifestation of two different underlying ideas about what an internal action is. CSP takes the view that an internal action is entirely internal to the process, and its presence cannot be determined by an external observer in any single run of the process. The kind of choice represented by τ can be made at the time a system is implemented.

LOTOS internal actions, however, normally must be present in the system at execution time. An internal action may represent a time-out, which can be influenced by other parts of the system, or an environmental factor which cannot be known at implementation time. The kind of choice represented by i is more general, and may in some cases be made at implementation time, and in others must be held over until execution time.

Given the state of a CSP process and the next action the environment wishes to perform, the process will then move to one of a number of possible stable states (a stable state is one in which no internal action may be performed) and offer the environment choices for its next action according to that state. In LOTOS, it is possible for the environment to interrupt the process as it passes through unstable states and take up any offer of an action which the process is able to perform in one of those unstable states.

The only induction rule in CSP to generate τs is the rule for internal choice. In parallel with CSP's definition, this is also the case in the LOTOS variant of §5.3.1.

5.3.4 Conclusion

We have introduced a CSP-like nondeterministic choice operator, $\sqcap$, into LOTOS, producing a modified language, for which we have given semantics in the form of inference rules. We have seen how the nondeterministic choice operator works, and have seen that it closely models our intuitive understanding of the notion of nondeterministic choice.

However, we will not use this language directly in our translation of state-based specifications into process algebra, because the target language of our translation is LOTOS, and including our nondeterministic choice operator forms a variant of LOTOS.

5.4 Conclusion

In this chapter, we have examined the role of nondeterminism in the semantics of state-based specifications, and have investigated ways to duplicate similar behaviour in LOTOS behaviour definitions. Starting with a translation from the state-based world to LOTOS which did not force return values to be chosen nondeterministically, we examined two possibilities for modifying the LOTOS language to incorporate two notions of nondeterminism.

First, we considered changing the semantics of LOTOS to force choices between output operations to be resolved nondeterministically, in such a way that the environment could not affect the outcome. This change introduced into the language a distinction between input and output actions, which LOTOS normally does not distinguish between. We gave behavioural semantics, in terms of labelled transition systems, for the variant of LOTOS which this change produced, and saw how it worked with some examples.

We noted that the major differences from the standard LOTOS semantics included the generation of a different kind of action label for actions generated by output action prefix constructs, and a change in the semantics of the parallel composition operator to distinguish the two kinds of action. We noted that, because the meaning of refusals, and thence the meaning of most of the LOTOS development relations including reduction, arise from the semantics of parallel composition, our changes rendered reduction meaningless in the LOTOS variant we created.

Secondly, we introduced a nondeterministic choice operator, similar to CSP's $\sqcap$ operator, into LOTOS. Again, we gave a behavioural semantics for the variant language we produced. In this case, we introduced only one new action label, τ, and the only rules introducing the new action are associated with the new $\sqcap$ operator. The semantics of the choice operator, [], were changed to allow the nondeterministic choice operator to "distribute itself" over ordinary choice. The other operators were changed only to treat τ the same way that they treat the internal action i.

Because we made no significant changes to the semantics of parallel composition, the usual development relations should still apply in this variant of LOTOS, provided they are adapted to treat τ similarly to the way they treat i. We have not verified this, however.

This concludes our excursion into variant LOTOS-like languages. In the next chapter, we will draw inspiration from these modified languages and bring together the other ideas presented so far to produce the final version of our translation from our state-based language to LOTOS which meets our requirements.

Chapter 6

Translating state-based specifications into LOTOS

In the previous chapter, we examined how the LOTOS formal specification language could be changed to introduce new ways of obtaining nondeterminism. We were interested in ensuring that the choice between a number of output actions is nondeterministic, such that the environment cannot influence the output of a translated operation.

However, we want our translation to translate state-based specifications into LOTOS, a language which is already well-understood. We will therefore only be using the modified LOTOS languages given in the previous chapter as a source of ideas for the translation we will give below; we will not define our translation to produce specifications in either of our modified languages.

Before we define our translation, we will discuss the features we require of it and show how we can achieve them in our translation.

6.1 Nondeterminism

In LOTOS, there is one way to obtain an entirely nondeterministic choice: present a choice where each branch of the choice has, as its initial action, the same action as each of the other branches. In many circumstances, the internal action, i, is used as the initial action, so that the inclusion of the nondeterministic choice produces no spurious observable behaviour. However, if we use our idea of using two actions to represent each operation, as discussed in §3.4.3, we can avoid the need for extra internal actions, by using the first of the two actions for each operation—the "call" action—to produce the nondeterministic choice. (Doing this simplifies the proof in chapter 7.) We shall now discuss how we will do this.

Let us imagine that we are translating a state-based specification, and are determining the LOTOS behaviour corresponding to state S_1. The predicate $pred_a$ of operation a satisfies the following:

$$before = S_1 \Rightarrow (pred_a \Leftrightarrow arg \in \{1, 2\} \wedge result = arg + 1 \wedge after = S_2)$$

When we model this operation using two actions per invocation, the split is (roughly) as follows: each initial action corresponds to a particular instantiation of the precondition, and each following action corresponds to a particular instantiation of the postcondition. In this case, a's precondition, pre_a, satisfies $before = S_1 \Rightarrow (pre_a \Leftrightarrow arg \in \{1, 2\})$, so the initial actions of the translation will be a_1 and a_2.

If the process performs a_1, this corresponds to the argument 1 being passed to the operation a. The predicate satisfies $before = S_1 \wedge arg = 1 \Rightarrow (pred_a \Leftrightarrow result = 2 \wedge after = S_2)$. Therefore, the process will then offer the action a_2 (corresponding to the act of returning the result 2 from the operation) and evolve to the behaviour corresponding to state S_2.

$pred_a$ does not, however, exhibit any nondeterminism. Let us examine $pred_b$, the predicate for an operation called b, which does exhibit nondeterminism:

$$before = S_1 \Rightarrow (pred_b \Leftrightarrow arg \in \{1, 2\} \wedge result \in \{3, 4\} \wedge after \in \{S_2, S_3\})$$

The precondition for $pred_b$ is $arg \in \{1, 2\}$, the same as the precondition of $pred_a$. However, nondeterminism is present in $pred_b$'s postcondition. After operation b is called with argument 1, corresponding to the translation performing the action b_1, operation b can return either 3 or 4, and can leave the object in state S_2 or S_3. This gives four possibilities for the second action in the translation, one of which must be chosen nondeterministically. We can achieve this by having the translation initially offer four actions labelled b_1, each of which leads to a behaviour corresponding to one of the four possibilities. Thus, we will get a nondeterministic choice between the possibilities for the second action. If we assume that b is the only operation defined, the resulting translated behaviour can be written as follows:

$$\begin{array}{ll} P := & b\_1;\ b\_3;\ Q \\ & [\,]\ b\_1;\ b\_3;\ R \\ & [\,]\ b\_1;\ b\_4;\ Q \\ & [\,]\ b\_1;\ b\_4;\ R \\ & [\,]\ b\_2;\ b\_3;\ Q \\ & [\,]\ b\_2;\ b\_3;\ R \\ & [\,]\ b\_2;\ b\_4;\ Q \\ & [\,]\ b\_2;\ b\_4;\ R \end{array}$$

Here, P is the translated behaviour of the state we are considering, Q is the translated behaviour of S_2 and R is the translation of state S_3.

We can see that the above definition has the properties we want. The choice between the arguments, 1 and 2, is deterministic, and can be decided entirely by the environment. That is, P initially offers both actions b_1 and b_2, and cannot refuse either of them. After the first action has been performed, for example b_1, P evolves to one of four different states, each of which offers one of the two actions b_3 and b_4, and where performing that action results in P evolving to one of the two behaviours Q or R. This situation corresponds to the operation b being called with argument 1 and then being able to return either 3 or 4 and evolving to after-state S_2 or S_3.

6.2 Initial state

Apart from nondeterminism occurring in an operation's postcondition, we also have to model the nondeterminism due to the possibility that a state-based specification can have multiple starting states. A state-based specification starts in one of these states, the particular state being chosen nondeterministically.

This behaviour is easy enough to arrange in our LOTOS translation. If we are translating a specification which has three initial states, and those states translate to the LOTOS behaviours P, Q and R, the behaviour corresponding to the whole specification is $i;\ P\ [\,]\ i;\ Q\ [\,]\ i;\ R$.

Thus, the first stage of our translation will consist of a choice between a number of alternatives, each of which consists of an internal action prefixed onto the behaviour for the translation of one of the states of the original state-based specification.

6.3 Example

Let us now look at a concrete example of the translation we have so far. We will translate the following state-based specification into LOTOS:

shuffler :

State space: S_1, S_2, S_3
Initial states: S_1, S_2
Operation move ($boolean \to \mathbb{N}$): $(before, arg, result, after)$
$\in \{(S_1, true, 2, S_2), (S_2, true, 3, S_3),$
$(S_3, true, 1, S_1), (S_1, false, 3, S_3),$
$(S_3, false, 2, S_2), (S_2, false, 1, S_1)\}$
Operation random ($nil \to \mathbb{N}$): $(result, after) \in \{(1, S_1), (2, S_2), (3, S_3)\}$
Operation lie ($nil \to \mathbb{N}$): $result \in \{1, 2, 3\} \wedge after \in \{S_1, S_2, S_3\}$

We have used a new data type here: *boolean*. As expected, this type consists of two values, *true* and *false*.

The *shuffler* specification describes an object which has three states. It rotates between them when its *move* operation is invoked. The argument passed determines which state the object moves to (specifically, *true* causes the object to move forward one state, *false* causes it to move back one state), and the return value tells the caller which state has been reached. When the *random* operation is invoked, the object moves to a random state, and returns the appropriate number. The *lie* operation has even less deterministic results; the state it reaches is not related to the value it returns. This will demonstrate our approach to nondeterminism.

We will denote the LOTOS behaviour corresponding to state S in the original specification as $[\![S]\!]$, and to the behaviour consisting of the translation of an entire specification *spec* as $[\![spec]\!]$.

Referring to the discussion in §6.2, we can write down a definition of $[\![shuffler]\!]$, in terms of the translations of its states, as follows:

$$[\![shuffler]\!] := i;\ [\![S_1]\!]\ []\ i;\ [\![S_2]\!]$$

Since the three states S_1, S_2 and S_3 are very similar to one another, we will only translate S_1 here.

Let us now translate the operation *move* in state S_1. *move*'s predicate, with S_1 substituted for *before*, simplifies to

$$(arg, result, after) \in \{(true, 2, S_2), (false, 3, S_3)\}$$

We can then extract *move*'s precondition for state S_1, which simplifies to $arg \in \{true, false\}$. We can then start our translation:

$$\begin{aligned}[\![S_1]\!] := {} & move\_true;\ \ldots\ []\ move\_true;\ \ldots\ []\ \ldots \\ & []\ move\_false;\ \ldots\ []\ move\_false;\ \ldots\ []\ \ldots \\ & []\ \ldots\end{aligned}$$

Next, we substitute S_1 for *before* and *true* for *arg* in *move*'s predicate, to obtain the possibilities for the second action after *move_true*:

$$(result, after) \in \{(2, S_2)\}$$

So there is just one possibility:

$$\begin{aligned}[\![S_1]\!] := {} & move\_true;\ move\_2;\ [\![S_2]\!] \\ & []\ move\_false;\ \ldots\ []\ move\_false;\ \ldots\ []\ \ldots \\ & []\ \ldots\end{aligned}$$

Doing the same for $arg = false$, we get

$$(result, after) \in \{(3, S_3)\}$$

So our translation of state S_1 becomes:

$$\begin{aligned} [\![S_1]\!] := {} & move\_true;\ move\_2;\ [\![S_2]\!] \\ & []\ move\_false;\ move\_3;\ [\![S_3]\!] \\ & []\ \ldots \end{aligned}$$

Next, we do the same for the operation *random*. This operation's argument is always *nil*, because of the implicit conjunct to the predicate, $arg \in \{nil\}$, which is due to the argument type given in the specification. *random*'s postcondition does not depend on *before* or *arg* at all, and is as follows regardless of before-state or argument value:

$$(result, after) \in \{(1, S_1), (2, S_2), (3, S_3)\}$$

Putting all this together, we can continue our translation as follows:

$$\begin{aligned} [\![S_1]\!] := {} & move\_true;\ move\_2;\ [\![S_2]\!] \\ & []\ move\_false;\ move\_3;\ [\![S_3]\!] \\ & []\ random\_nil;\ random\_1;\ [\![S_1]\!] \\ & []\ random\_nil;\ random\_2;\ [\![S_2]\!] \\ & []\ random\_nil;\ random\_3;\ [\![S_3]\!] \\ & []\ \ldots \end{aligned}$$

We can now begin to see how the nondeterminism works: if the environment offers $[\![S_1]\!]$ the action *random_nil*, then it cannot be refused. After *random_nil* has been performed, $[\![S_1]\!]$ can evolve to one of three behaviours, each of which offers one of the actions *random_1*, *random_2* or *random_3*.

Let us now complete the translation by translating the final operation, *lie*. *lie*'s precondition, as *random*'s, is $arg \in \{nil\}$. This means that only one action can be used to start the *lie* operation: *lie_nil*. *lie*'s second action can be drawn from a larger set, since the predicate, with $before = S_0$ and $arg = nil$ is the same as *lie*'s given predicate:

$$result \in \{1, 2, 3\} \wedge after \in \{S_1, S_2, S_3\}$$

This given nine possible second actions for the *lie* operation. The finished translation of S_1 is therefore as follows:

$$\begin{aligned} [\![S_1]\!] := {} & move\_true;\ move\_2;\ [\![S_2]\!]\ []\ move\_false;\ move\_3;\ [\![S_3]\!] \\ & []\ random\_nil;\ random\_1;\ [\![S_1]\!]\ []\ random\_nil;\ random\_2;\ [\![S_2]\!] \\ & []\ random\_nil;\ random\_3;\ [\![S_3]\!] \\ & []\ lie\_nil;\ lie\_1;\ [\![S_1]\!]\ []\ lie\_nil;\ lie\_2;\ [\![S_1]\!]\ []\ lie\_nil;\ lie\_3;\ [\![S_1]\!] \\ & []\ lie\_nil;\ lie\_1;\ [\![S_2]\!]\ []\ lie\_nil;\ lie\_2;\ [\![S_2]\!]\ []\ lie\_nil;\ lie\_3;\ [\![S_2]\!] \\ & []\ lie\_nil;\ lie\_1;\ [\![S_3]\!]\ []\ lie\_nil;\ lie\_2;\ [\![S_3]\!]\ []\ lie\_nil;\ lie\_3;\ [\![S_3]\!] \end{aligned}$$

The *lie* action demonstrates the action of our method of translating nondeterminism: After performing the first action of the pair, the choice of which is entirely determinable by the environment, the choice of which second action is then offered is entirely nondeterministic; the environment is not able to influence that choice at all. This is as we wish; as we showed in §6.1, this effect will help the translation of a subtype to be a reduction of the translation of its supertype.

6.4 Unspecified behaviour

In chapter 4, we added unspecified behaviour to our labelled transition systems by adding a "cloud" to each node, then interpreting these clouds as representing a choice between all behaviours of the form $a;\ \Omega$, where a was not an action which the process could otherwise accept, and Ω was a process exhibiting "unspecified behaviour".

In this section, we will again add unspecified behaviour to our translation. However, we will take a somewhat different approach this time. Instead of adding unspecified behaviour wherever the process algebraic translation refuses an action, we will add it only where the behaviour of the state-based specification permits unspecified behaviour.

Consider again the example specification given above:

> *shuffler* :
>
> **State space**: S_1, S_2, S_3
> **Initial states**: S_1, S_2
> **Operation** move ($boolean \to \mathbb{N}$): $(before, arg, result, after)$
> $\in \{(S_1, true, 2, S_2), (S_2, true, 3, S_3),$
> $(S_3, true, 1, S_1), (S_1, false, 3, S_3),$
> $(S_3, false, 2, S_2), (S_2, false, 1, S_1)\}$
> **Operation** random ($nil \to \mathbb{N}$): $(result, after) \in \{(1, S_1), (2, S_2), (3, S_3)\}$
> **Operation** lie ($nil \to \mathbb{N}$): $result \in \{1, 2, 3\} \wedge after \in \{S_1, S_2, S_3\}$

Let us examine the operations in this specification in turn.

We previously translated the *move* operation as follows:

$$
\begin{aligned}
[\![S_1]\!] :=\ & move\_true;\ move\_2;\ [\![S_2]\!]\ []\ move\_false;\ move\_3;\ [\![S_3]\!] \\
& []\ \ldots
\end{aligned}
$$

This corresponds to the explicitly specified part of *shuffler*'s behaviour: if *move* is called with argument *true*, 2 is returned and the after-state is S_2; if *move* is called with argument *false*, 3 will be returned and the after-state is S_3. What behaviour is unspecified, though?

As we saw in §4.4.1, we interpret the operation's precondition ($before \in \{S_1, S_2, S_3\} \wedge arg \in \{true, false\}$) to define the conditions under which the operation is specified; so, if *move* is called with argument *true* or *false*, when the object is in state S_1, S_2 or S_3, the object's behaviour is determined by the specification. But if the operation is called with a different argument, or in a different state—when *move*'s precondition does not hold—the specification does not define the outcome. This is the case in which we consider the operation's behaviour to be unspecified.

Let us now rewrite the translation of the *move* operation above to include this unspecified behaviour:

$$
\begin{aligned}
[\![S_1]\!] :=\ & move\_true;\ move\_2;\ [\![S_2]\!] \\
& []\ move\_false;\ move\_3;\ [\![S_3]\!] \\
& []\ move\_nil;\ \ldots\ []\ move\_0;\ \ldots\ []\ move\_1;\ \ldots\ []\ move\_2;\ \ldots\ []\ \ldots \\
& []\ \ldots
\end{aligned}
$$

The intent here is to add a choice to the behaviour for each argument not acceptable in the state we are translating which accepts that argument and then becomes unspecified. Because the argument type given in the specification is considered to be an extension of the predicate, and not a statically checked type, we must add undefined behaviour for all the possible values which could be passed.

Now we need to decide what our "unspecified" behaviour will be; we need to decide how Ω is to be manifest. What does it mean for an operation's behaviour to be unspecified, in the context of the present version of the translation? Its before-state and argument are already decided by the object's history and its environment respectively. The return value and after-state are decided internally, so they are the aspects which we will make unspecified.

Firstly, when one of the actions we added above is performed, the operation must then be able to return any value. We will modify the behaviour as follows to achieve this:

$$
\begin{aligned}
[\![S_1]\!] := {} & \mathit{move\_true};\ \mathit{move\_2};\ [\![S_2]\!] \\
& [\,]\ \mathit{move\_false};\ \mathit{move\_3};\ [\![S_3]\!] \\
& [\,]\ \mathit{move\_nil};\ \mathit{move\_nil};\ \ldots\ [\,]\ \mathit{move\_nil};\ \mathit{move\_true};\ \ldots \\
& [\,]\ \mathit{move\_nil};\ \mathit{move\_false};\ \ldots\ [\,]\ \mathit{move\_nil};\ \mathit{move\_0};\ \ldots \\
& [\,]\ \mathit{move\_nil};\ \mathit{move\_1};\ \ldots\ [\,]\ \mathit{move\_nil};\ \mathit{move\_2};\ \ldots \\
& [\,]\ \ldots \\
& [\,]\ \mathit{move\_0};\ \mathit{move\_nil};\ \ldots\ [\,]\ \mathit{move\_0};\ \mathit{move\_true};\ \ldots\ [\,]\ \mathit{move\_0};\ \mathit{move\_false};\ \ldots \\
& [\,]\ \mathit{move\_0};\ \mathit{move\_0};\ \ldots\ [\,]\ \mathit{move\_0};\ \mathit{move\_1};\ \ldots\ [\,]\ \mathit{move\_0};\ \mathit{move\_2};\ \ldots \\
& [\,]\ \ldots \\
& [\,]\ \mathit{move\_1};\ \ldots\ [\,]\ \mathit{move\_2};\ \ldots\ [\,]\ \ldots \\
& [\,]\ \ldots
\end{aligned}
$$

For each possible initial action, we have supplied each possible second action in the same way we dealt with nondeterministic choice.

Lastly, after the second action is performed, the translation must be able to evolve to any state. That is, to the *translation* of any state. We previously defined Ω, the behaviour of the process after an unspecified operation, to include anything a LOTOS process could do. Now that we have refined our translation somewhat, we need not include *all* behaviour, just that which any translation of a state-based specification could do.

The way we will achieve this effect is to pretend that Ω is a state in the original state-based specification, and assume that the predicate of every operation is false when the before-state is Ω. Then, after the second action of an unspecified operation, the translation will evolve to $[\![\Omega]\!]$. In this state, every operation will be unspecified, giving the behaviour we want. The *move* operation now translates as follows:

$$
\begin{aligned}
[\![S_1]\!] := {} & \mathit{move\_true};\ \mathit{move\_2};\ [\![S_2]\!] \\
& [\,]\ \mathit{move\_false};\ \mathit{move\_3};\ [\![S_3]\!] \\
& [\,]\ \mathit{move\_nil};\ \mathit{move\_nil};\ [\![\Omega]\!]\ [\,]\ \mathit{move\_nil};\ \mathit{move\_true};\ [\![\Omega]\!] \\
& [\,]\ \mathit{move\_nil};\ \mathit{move\_false};\ [\![\Omega]\!]\ [\,]\ \mathit{move\_nil};\ \mathit{move\_0};\ [\![\Omega]\!] \\
& [\,]\ \mathit{move\_nil};\ \mathit{move\_1};\ [\![\Omega]\!]\ [\,]\ \mathit{move\_nil};\ \mathit{move\_2};\ [\![\Omega]\!] \\
& [\,]\ \ldots \\
& [\,]\ \mathit{move\_0};\ \mathit{move\_nil};\ [\![\Omega]\!]\ [\,]\ \mathit{move\_0};\ \mathit{move\_true};\ [\![\Omega]\!] \\
& [\,]\ \mathit{move\_0};\ \mathit{move\_false};\ [\![\Omega]\!]\ [\,]\ \mathit{move\_0};\ \mathit{move\_0};\ [\![\Omega]\!] \\
& [\,]\ \mathit{move\_0};\ \mathit{move\_1};\ [\![\Omega]\!]\ [\,]\ \mathit{move\_0};\ \mathit{move\_2};\ [\![\Omega]\!] \\
& [\,]\ \ldots \\
& [\,]\ \mathit{move\_1};\ \ldots\ [\,]\ \mathit{move\_2};\ \ldots\ [\,]\ \ldots \\
& [\,]\ \ldots
\end{aligned}
$$

We have one final change to make before the translation is finished. Imagine that an operation's predicate is equal to *true*. Such an operation would be able to be applied in any state, with any argument value, and would be able to return any return value and evolve to any state. This is the behaviour we have given our Ω state, and the behaviour we attribute to unspecified operations. Since we feel that there is a difference between an operation which is unspecified and an operation which is specified to have the predicate *true*, we would like our translation to distinguish the two.

In a real system, there is one possible outcome of calling an operation which is not represented in our state-based specification language. An operation can fail to return. This could be for any of a number of reasons: the run-time system could detect a call to a nonexistent routine, and halt the program; the routine could enter an endless loop; in a multiprocessing or distributed system, a deadlock or livelock could occur within the routine.

We will model these various calamities, any of which could prevent the operation completing, by pretending that they cause the operation to return a distinguished value: $\perp$. No operation whose behaviour is specified should ever return $\perp$, and the environment should not be able to pass $\perp$ as an argument value; its only purpose is to be a possible return value of any unspecified operation.

This gives us the property we wanted. The operation with predicate *true* is now distinguishable from an unspecified operation: If the value $\perp$ can be returned from an operation invocation, then that operation invocation is unspecified; if $\perp$ is not a possibility, the outcome of that invocation was defined in the specification.

The final translation of our example specification will therefore be as follows:

$$
\begin{array}{ll}
[\![S_1]\!] := & \mathit{move\_true};\ \mathit{move\_2};\ [\![S_2]\!] \\
& [\,]\ \mathit{move\_false};\ \mathit{move\_3};\ [\![S_3]\!] \\
& [\,]\ \mathit{move\_nil};\ \mathit{move\_\perp};\ [\![\Omega]\!]\ [\,]\ \mathit{move\_nil};\ \mathit{move\_nil};\ [\![\Omega]\!] \\
& [\,]\ \mathit{move\_nil};\ \mathit{move\_true};\ [\![\Omega]\!]\ [\,]\ \mathit{move\_nil};\ \mathit{move\_false};\ [\![\Omega]\!] \\
& [\,]\ \mathit{move\_nil};\ \mathit{move\_0};\ [\![\Omega]\!]\ [\,]\ \mathit{move\_nil};\ \mathit{move\_1};\ [\![\Omega]\!] \\
& [\,]\ \ldots \\
& [\,]\ \mathit{move\_0};\ \mathit{move\_\perp};\ [\![\Omega]\!]\ [\,]\ \mathit{move\_0};\ \mathit{move\_nil};\ [\![\Omega]\!] \\
& [\,]\ \mathit{move\_0};\ \mathit{move\_true};\ [\![\Omega]\!]\ [\,]\ \mathit{move\_0};\ \mathit{move\_false};\ [\![\Omega]\!] \\
& [\,]\ \mathit{move\_0};\ \mathit{move\_0};\ [\![\Omega]\!]\ [\,]\ \mathit{move\_0};\ \mathit{move\_1};\ [\![\Omega]\!] \\
& [\,]\ \ldots \\
& [\,]\ \mathit{move\_1};\ \mathit{move\_\perp};\ [\![\Omega]\!]\ [\,]\ \mathit{move\_1};\ \mathit{move\_nil};\ [\![\Omega]\!] \\
& [\,]\ \mathit{move\_1};\ \mathit{move\_true};\ [\![\Omega]\!]\ [\,]\ \mathit{move\_1};\ \mathit{move\_false};\ [\![\Omega]\!] \\
& [\,]\ \ldots \\
& [\,]\ \mathit{move\_2};\ \ldots\ [\,]\ \mathit{move\_3};\ \ldots\ [\,]\ \ldots \\
& [\,]\ \ldots \\
& [\,]\ \mathit{random\_nil};\ \mathit{random\_1};\ [\![S_1]\!] \\
& [\,]\ \mathit{random\_nil};\ \mathit{random\_2};\ [\![S_2]\!] \\
& [\,]\ \mathit{random\_nil};\ \mathit{random\_3};\ [\![S_3]\!] \\
& [\,]\ \mathit{random\_true};\ \mathit{random\_\perp};\ [\![\Omega]\!]\ [\,]\ \mathit{random\_true};\ \mathit{random\_nil};\ [\![\Omega]\!] \\
& [\,]\ \mathit{random\_true};\ \mathit{random\_true};\ [\![\Omega]\!]\ [\,]\ \mathit{random\_true};\ \mathit{random\_false};\ [\![\Omega]\!] \\
& [\,]\ \mathit{random\_true};\ \mathit{random\_0};\ [\![\Omega]\!]\ [\,]\ \mathit{random\_true};\ \mathit{random\_1};\ [\![\Omega]\!] \\
& [\,]\ \ldots \\
& [\,]\ \mathit{random\_false};\ \mathit{random\_\perp};\ [\![\Omega]\!]\ [\,]\ \mathit{random\_false};\ \mathit{random\_nil};\ [\![\Omega]\!] \\
& [\,]\ \mathit{random\_false};\ \mathit{random\_true};\ [\![\Omega]\!]\ [\,]\ \mathit{random\_false};\ \mathit{random\_false};\ [\![\Omega]\!] \\
& [\,]\ \ldots \\
& [\,]\ \mathit{random\_0};\ \mathit{random\_nil};\ [\![\Omega]\!] \\
& [\,]\ \ldots \\
& [\,]\ \mathit{lie\_nil};\ \mathit{lie\_1};\ [\![S_1]\!]\ [\,]\ \mathit{lie\_nil};\ \mathit{lie\_2};\ [\![S_1]\!] \\
& [\,]\ \mathit{lie\_nil};\ \mathit{lie\_3};\ [\![S_1]\!]\ [\,]\ \mathit{lie\_nil};\ \mathit{lie\_1};\ [\![S_2]\!] \\
& [\,]\ \mathit{lie\_nil};\ \mathit{lie\_2};\ [\![S_2]\!]\ [\,]\ \mathit{lie\_nil};\ \mathit{lie\_3};\ [\![S_2]\!] \\
& [\,]\ \mathit{lie\_nil};\ \mathit{lie\_1};\ [\![S_3]\!]\ [\,]\ \mathit{lie\_nil};\ \mathit{lie\_2};\ [\![S_3]\!] \\
& [\,]\ \mathit{lie\_nil};\ \mathit{lie\_3};\ [\![S_3]\!] \\
& [\,]\ \mathit{lie\_true};\ \mathit{lie\_\perp};\ [\![\Omega]\!]\ [\,]\ \mathit{lie\_true};\ \mathit{lie\_nil};\ [\![\Omega]\!] \\
& [\,]\ \mathit{lie\_true};\ \mathit{lie\_true};\ [\![\Omega]\!]\ [\,]\ \mathit{lie\_true};\ \mathit{lie\_false};\ [\![\Omega]\!] \\
& [\,]\ \mathit{lie\_true};\ \mathit{lie\_0};\ [\![\Omega]\!]\ [\,]\ \mathit{lie\_true};\ \mathit{lie\_1};\ [\![\Omega]\!] \\
& [\,]\ \ldots \\
& [\,]\ \mathit{count\_nil};\ \mathit{count\_\perp};\ [\![\Omega]\!]\ [\,]\ \mathit{count\_nil};\ \mathit{count\_nil};\ [\![\Omega]\!] \\
& [\,]\ \mathit{count\_nil};\ \mathit{count\_true};\ [\![\Omega]\!]\ [\,]\ \mathit{count\_nil};\ \ldots \\
& [\,]\ \mathit{count\_true};\ \ldots\ [\,]\ \mathit{count\_false};\ \ldots \\
& [\,]\ \mathit{count\_0};\ \ldots\ [\,]\ \mathit{count\_1};\ \ldots \\
& [\,]\ \ldots \\
& [\,]\ \mathit{xyzzy\_nil};\ \ldots \\
& [\,]\ \ldots
\end{array}
$$

As can be seen, the specification above is infinite; *data* includes the natural numbers, and, for each operation, the behaviour must include at least one branch for each element of *data*. Our translation is intended to produce LOTOS specifications, but LOTOS specifications may not be infinite (although a finite LOTOS specification may produce infinite behaviour).

In addition, we have only translated a single state of the original state based specification. A state based specification may have a large, even infinite, state space. Our method of translating each state individually produces an infinite LOTOS specification when applied to a state based specification with an infinite state space.

We will examine these problems in more detail after we have described the translation.

6.5 Tuple form of the state-based specification notation

The state-based notation described in §3.1 has been a useful concrete syntax for presenting examples, but we will need a more mathematical representation when we come to translate state-based specifications into process algebra. We now present such a representation.

Before we examine the specification notation itself, we will need to introduce two sets, from which certain values are drawn. *ops* is the set of all operation names, and *data* is the set of all data values which can be passed as arguments or return values of any operations. When we wish to quantify an expression over "all operation names" or "all data values", we quantify over the appropriate set: *ops* or *data*.

For any given example we will choose appropriate values for *ops* and *data* which include all the operation names and data values we will need to demonstrate how the example works. For example, imagine that we are comparing two specifications, a and b. Specification a contains operations called *get*, *put* and *count* and specification b contains operations *get*, *count* and *read*. We can take $ops = \{get, put, count, read\}$, since we will not be concerned with what happens to any other operation.

The tuple form of our state-based specification notation consists of tuples of the form (s, t, m). s is the state-space of the type. t is the set of initial states of the type; it is a non-empty subset of s. The semantics of t is the same as for the syntactic notation: when an object of the type is created and initialised, it begins in one of the states in t. The choice of which initial state is up to the implementation, and need not be deterministic. m is of the form $\{(op_1, p_1), (op_2, p_2), \ldots, (op_n, p_n)\}$. It describes the operations which objects of the type can perform. m associates the name (op_i) and predicate (p_i) of each method in the specification. $op_1, \ldots, op_n$ are distinct operation names drawn from the set *ops*, the set of all possible operation names.

Each predicate, p_i, operates on a 4-tuple, $(before, arg, result, after)$, and defines whether op_i is able to return *result* and change the state to *after*, when invoked in state *before* with argument *arg*. This works much the same as the predicate in the syntactic notation, except that the extra terms implied by the types given for the argument and return values must be written out explicitly when using our notation's tuple form, and that we use λ-notation to write the function.

As before, we simplify our handling of operation arguments and results by mandating that all operations have one argument and one result. If no value is to be passed, we pass a 0-tuple, (), which we write as *nil*, to avoid confusion. If more than one value is to be passed, we pass a tuple made up of the required values. In cases where exactly one value is to be passed, we omit the brackets, and just write the value.

6.5.1 Examples

In §3.1.1, we presented an example specification, *buffer*1:

$buffer1$:

State space: $\mathbb{N} \cup \{VOID\}$
Initial states: $VOID$
Operation receive ($\mathbb{N} \to nil$): $before = VOID \wedge after = arg$
Operation transmit ($nil \to \mathbb{N}$): $before \neq VOID \wedge result = before$
$\wedge\ after = VOID$

We saw that the full predicate of *receive* is

$$before = VOID \wedge after = arg \wedge arg \in \mathbb{N} \wedge result = nil$$

and that the full predicate for *transmit* is

$$before \neq VOID \wedge result = before \wedge after = VOID \wedge arg = nil \wedge result \in \mathbb{N}$$

We can now re-express this specification in our tuple notation, as follows:

$$\begin{aligned} buffer1 = (&\mathbb{N} \cup \{VOID\}, \{VOID\}, \\ &\{(receive, \lambda(before, arg, result, after) \bullet \\ &\quad before = VOID \wedge after = arg \wedge arg \in \mathbb{N} \wedge result = nil), \\ &(transmit, \lambda(before, arg, result, after) \bullet \\ &\quad before \neq VOID \wedge result = before \wedge after = VOID \\ &\quad \wedge\ arg = nil \wedge result \in \mathbb{N})\}) \end{aligned}$$

Note that the state-space and the initial states are the same in both the syntactic and tuple forms, and the predicate body is changed in the tuple form only to include those terms which were implicit in the syntactic form, and to express the predicate as a λ-expression.

Although less immediately understandable in this form than the same specification was in the syntactic form, this notation will be useful because it is more amenable to mathematical treatment. When we specify our translation from state-based to process algebraic specifications, we will use the tuple form of our state-based notation as input to the translation.

Stacks, bags and queues

Let us now further illustrate the tuple form by converting our stack, bag and queue type specifications from §3.1.2 into this form. First, recall the definitions, which were as follows:

bag :

State space: multisets of $\mathbb{N}$
Initial states: $\langle\!\vert\,\vert\!\rangle$
Operation put ($\{0,1\} \to nil$): $|before| < 2 \wedge after = before \cup \langle\!\vert\ arg\ \vert\!\rangle$
Operation get ($nil \to \{0,1\}$): $|before| > 0 \wedge before = after \cup \langle\!\vert\ result\ \vert\!\rangle$

$stack$:

State space: lists of $\mathbb{N}$
Initial states: $[\,]$
Operation put ($\{0,1\} \to nil$): $|before| < 2 \wedge after = before ++ [arg]$
Operation get ($nil \to \{0,1\}$): $|before| > 0 \wedge before = after ++ [result]$

queue :

State space: lists of $\mathbb{N}$
Initial states: $[\,]$
Operation put ($\{0,1\} \to nil$): $|\,before\,| < 2 \wedge after = before ++ [arg]$
Operation get ($nil \to \{0,1\}$): $|\,before\,| > 0 \wedge before = [result] ++ after$

As mentioned above, the state-space and initial states of the specifications will be unchanged as we convert to the tuple form. However, we will need to convert the predicates by determining what the implicit terms are. For the *put* operation of the *bag* specification, the argument type is given as $\{0,1\}$. This translates to the term $arg \in \{0,1\}$ in the predicate. The result type is *nil*, giving a predicate term of $result = nil$. Similarly, the terms for the *get* operation are $arg = nil$ and $result \in \{0,1\}$. The terms for the *stack* and *queue* specifications are the same, since their operations' argument and result types are the same as those of the corresponding operations in *bag*.

We can now write down the tuple form of the *bag*, *stack* and *queue* specifications:

$$\begin{aligned}
bag = (&\text{multisets of } \mathbb{N}, \{\langle\!|\,|\!\rangle\},\\
&\{(put, \lambda(before, arg, result, after) \bullet\\
&\qquad |\,before\,| < 2 \wedge after = before \cup \langle\!|\ arg\ |\!\rangle \wedge arg \in \{0,1\} \wedge result = nil),\\
&\ (get, \lambda(before, arg, result, after) \bullet\\
&\qquad |\,before\,| > 0 \wedge before = after \cup \langle\!|\ result\ |\!\rangle \wedge arg = nil \wedge result \in \{0,1\}
\end{aligned}$$

$$\begin{aligned}
stack = (&\text{lists of } \mathbb{N}, \{[\,]\},\\
&\{(put, \lambda(before, arg, result, after) \bullet\\
&\qquad |\,before\,| < 2 \wedge after = before ++ [arg] \wedge arg \in \{0,1\} \wedge result = nil),\\
&\ (get, \lambda(before, arg, result, after) \bullet\\
&\qquad |\,before\,| > 0 \wedge before = after ++ [result] \wedge arg = nil \wedge result \in \{0,1\})\}
\end{aligned}$$

$$\begin{aligned}
queue = (&\text{lists of } \mathbb{N}, \{[\,]\},\\
&\{(put, \lambda(before, arg, result, after) \bullet\\
&\qquad |\,before\,| < 2 \wedge after = before ++ [arg] \wedge arg \in \{0,1\} \wedge result = nil),\\
&\ (get, \lambda(before, arg, result, after) \bullet\\
&\qquad |\,before\,| > 0 \wedge before = [result] ++ after \wedge arg = nil \wedge result \in \{0,1\})\}
\end{aligned}$$

6.6 Translation

We have now seen informally how we intend to translate state-based specifications into LOTOS, and we have introduced the notation we will use to represent our state-based language. In this section, we will draw on the above to present a more formal definition of our translation from state-based specifications to LOTOS.

Further to the discussion in §6.2, a specification *spec* of the form (s,t,m) translates into a nondeterministic choice between the translations of the specification's initial states, as follows:

$$[\![(s,t,m)]\!] := choice\ e : t\ [\,]\ i;\ [\![e,(s,t,m)]\!]$$

We now have to translate each state of the specification separately. As discussed in §6.1, each state translates into a choice between a number of alternatives. There is a separate branch of the choice for each combination of argument, result value and after-state consistent with the operation's predicate in the before-state we are translating. We can calculate the set of $(arg, result, after)$ triples which correspond to these choices from the operation's predicate, *pred*, given a before-state *before* and the specification's state-space s:

$$f'(before, pred, s) = \{(arg, result, after) : data \times data \times s \mid pred(before, arg, result, after)\}$$

(Recall that the predicate in the tuple form of our notation incorporates those terms due to the argument and result data types which are only implicit in the syntactic form.)

As we discussed in §6.4, we will now modify this definition to incorporate the "unspecified" behaviour implicit in the state-based specification. The contents of our set of triples must now depend on the operation's precondition. Given the operation's predicate *pred*, the specification's state-space s, the before-state *before* and the argument *arg*, the precondition is:

$$pre(pred, s, before, arg) = \exists\, result : data, after : s \bullet pred(before, arg, result, after)$$

When the operation's precondition holds for a given argument value, we get the same triples as in the definition of f' above. When the operation's precondition does not hold for a given argument value, however, we get triples representing "unspecified" behaviour. The triples we get include all possible return values, together with the inclusion of the distinguished return value $\perp$, with the distinguished after-state Ω.

We define f to be this set of triples, as follows:

$$\begin{aligned} &f(before, pred, s)\\ &\quad = \{(arg, result, after) : data \times (data \cup \{\perp\}) \times (s \cup \{\Omega\})\\ &\qquad\qquad \mid (before \neq \Omega \wedge pre(pred, s, before, arg))\\ &\qquad\qquad\quad ?(result \neq \perp \wedge after \neq \Omega \wedge pred(before, arg, result, after)) : after = \Omega\} \end{aligned}$$

(We use the conditional expression notation $x : y?z$, whose value is y if x is true, otherwise z.)

This definition ensures that, for values of *before* and *arg* for which the operation's precondition holds (and in state $before = \Omega$, the precondition is taken to be *false*), the values of *arg*, *result* and *after* are consistent with the operation's predicate. If the precondition does not hold, or $before = \Omega$, the values of *result* and *after* are allowed to take on any value in the sets $data \cup \{\perp\}$ and $s \cup \{\Omega\}$ respectively.

In a given state, we will need to give a choice between all the behaviours corresponding to the triples in f for each of the operations in ops. We will therefore construct quadruples representing all the branches of behaviour possible in state *before*, where the specification's state-space is s and its set of operation name-predicate pairs is m, as follows:

$$\begin{aligned} &g(before, s, m)\\ &\quad = \{(op, arg, result, after) : ops \times data \times (data \cup \{\perp\}) \times (s \cup \{\Omega\})\\ &\qquad\qquad \mid (arg, result, after) \in f(before, m(op), s)\} \end{aligned}$$

Since m is in the form of a function, we have used it as such to obtain the predicate for each operation.

To complete the translation, we now simply have to construct the choice between the behaviours represented by these quadruples:

$$\begin{aligned} [\![e, (s, t, m)]\!] := {}& choice\ (op, arg, result, after) : g(e, s, m)\\ & [\,]\ op\_arg;\ op\_result;\ [\![after, (s, t, m)]\!] \end{aligned}$$

This completes the definition of our translation.

6.6.1 Problems with this translation

The translation as defined above has two major problems.

Infinite state spaces As we perform a translation, we translate each state into a separate LOTOS process definition. In order to translate a state based specification to LOTOS, we must first translate each of its initial states, then we must translate the states which can be reached from them, and then all the states which can further be reached from those states, recursively. This usually means that all the states in a specification must be translated.

Many state based specifications have infinite state spaces. For example, a specification whose state space is the natural numbers has an infinite state space. Translating a specification with an infinite state space can result in an infinitely long translation; but a valid LOTOS specification must be finite.

If all the states of a specification with an infinite state space are required to be translated, then the work of translating the specification can never be completed!

This is a consequence of the "flattening" nature of our translation (see §4.1.3). However, the problem of infinite state spaces does not prevent the translation from being used to translate specifications with finite state spaces.

Choice between processes The LOTOS translation of each state comprises a choice between many branches of behaviour. There is one branch for each combination of argument, result and after-state allowed by each operation's predicate or by our interpretation of unspecified behaviour.

As we saw in the example above, a typical specification requires *data* to contain an infinite number of elements, typically because it includes $\mathbb{N}$. This results in an infinite number of branches being produced for each translated state.

In our translation, we have avoided producing an infinitely large LOTOS definition for each translated state by using LOTOS's generalised choice construct. We choose the values for the operation name, argument, result value and after-state from a set, thus pushing the potentially infinite number of choices from the LOTOS specification into the world of set theory, where it is quite valid.

Unfortunately, in order to do this, we had to abuse the generalised choice construct. Standard LOTOS allows us to use this construct to choose data values or action names from a set. For example, the following are valid LOTOS behaviours:

$$\begin{array}{l} \mathit{choice}\ x : \{1, 2, 3\}\ []\ a!x;\ \mathit{stop} \\ \mathit{choice}\ x : \{a, b, c\}\ []\ x!2;\ \mathit{stop} \end{array}$$

This works well for the operation name, argument and result, but does not accommodate our choosing the after-state from a set. There does not appear to be any technical obstacle to allowing processes to be chosen from a set as in this example, where Q, R and S are process names:

$$\mathit{choice}\ X : \{Q, R, S\}\ []\ a;\ X$$

Unfortunately, the above is not legal in LOTOS. If it were, however, we could modify our translation as follows and obtain a valid LOTOS translation of each state:

$$\begin{array}{l} g(\mathit{before}, (s, t, m))' \\ \qquad = \{(\mathit{op}, \mathit{arg}, \mathit{result}, C) : \mathit{ops} \times \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times \mathit{Beh} \\ \qquad\qquad\qquad \mid (\mathit{arg}, \mathit{result}, \mathit{after}) \in f(\mathit{before}, m(\mathit{op}), s) \wedge C = [\![\mathit{after}, (s, t, m)]\!]\} \\ [\![e, \mathit{spec}]\!]' := \mathit{choice}\ (\mathit{op}, \mathit{arg}, \mathit{result}, C) : g(e, \mathit{spec})\ []\ \mathit{op\_arg};\ \mathit{op\_result};\ C \end{array}$$

This is unfortunately no more legal according to standard LOTOS than the definition we have already given. We will therefore consider only examples where *data* is finite (and small), and we will expand out the generalised choice into a number of ordinary LOTOS choice constructs.

6.7 Example

To demonstrate how our translation works, we will now use it to translate the bag specification from §6.5.1 into LOTOS. The specification is as follows:

$$\begin{aligned}
bag = {} & (\text{multisets of } \mathbb{N}, \{\Lbag\Rbag\}, \\
& \quad \{(put, \lambda(before, arg, result, after) \bullet \\
& \qquad |\, before\,| < 2 \wedge after = before \cup \Lbag\ arg\ \Rbag \wedge arg \in \{0,1\} \wedge result = nil), \\
& \quad\ (get, \lambda(before, arg, result, after) \bullet \\
& \qquad |\, before\,| > 0 \wedge before = after \cup \Lbag\ result\ \Rbag \wedge arg = nil \wedge result \in \{0,1\})\})
\end{aligned}$$

We will take $ops = \{put, get, xyzzy\}$ and $data = \{nil, 0, 1, 2\}$ for the purpose of translating this example.

The overall translation of the specification is given by the following definition:

$$[\![(s,t,m)]\!] := choice\ \ e : t\ [\,]\ i;\ [\![e,(s,t,m)]\!]$$

Applying this to the case of bag, we see that $t = \{\Lbag\Rbag\}$, giving the following definition:

$$[\![bag]\!] := choice\ \ e : \{\Lbag\Rbag\}\ [\,]\ i;\ [\![e, bag]\!]$$

Because the bag specification only has a single initial state, the choice construct has only one possible branch here, so we can simplify the choice away entirely to get the following equivalent definition:

$$[\![bag]\!] := i;\ [\![\Lbag\Rbag, bag]\!]$$

Now we need to work out the definition of $[\![\Lbag\Rbag, bag]\!]$. For this, we will need to use the following definition:

$$\begin{aligned}
[\![e,(s,t,m)]\!] := {} & choice\ (op, arg, result, after) : g(e,s,m) \\
& [\,]\ op_arg;\ op_result;\ [\![after,(s,t,m)]\!]
\end{aligned}$$

The specific definition we need, then, is:

$$\begin{aligned}
[\![\Lbag\Rbag, bag]\!] := {} & choice\ (op, arg, result, after) : g(\Lbag\Rbag, \text{multisets of } \mathbb{N}, m) \\
& [\,]\ op_arg;\ op_result;\ [\![after, bag]\!]
\end{aligned}$$
where m is the methods element of the bag specification

We now need to evaluate $g(\Lbag\Rbag, \text{multisets of } \mathbb{N}, m)$. The definition of g, from above, is:

$$\begin{aligned}
& g(before, s, m) \\
& \quad = \{(op, arg, result, after) : ops \times data \times (data \cup \{\bot\}) \times (s \cup \{\Omega\}) \\
& \qquad\quad |\ (arg, result, after) \in f(before, m(op), s)\}
\end{aligned}$$

Substituting the particular values for this case, we get the following:

$$\begin{aligned}
& g(\Lbag\Rbag, \text{multisets of } \mathbb{N}, m) \\
& \quad = \{(op, arg, result, after) : ops \times data \times (data \cup \{\bot\}) \times ((\text{multisets of } \mathbb{N}) \cup \{\Omega\}) \\
& \qquad\quad |\ (arg, result, after) \in f(\Lbag\Rbag, m(op), \text{multisets of } \mathbb{N})\}
\end{aligned}$$

To work this out, we need to evaluate $f(\Lbag\Rbag, m(op), \text{multisets of } \mathbb{N})$ for each op in ops. Let us first take $op = put$, then $op = get$, then $op = xyzzy$. We defined f as follows:

$$\begin{aligned}
&f(before, pred, s)\\
&\quad = \{(arg, result, after) : data \times (data \cup \{\perp\}) \times (s \cup \{\Omega\})\\
&\qquad\qquad \mid (before \neq \Omega \wedge pre(pred, s, before, arg))\\
&\qquad\qquad\qquad ?(result \neq \perp \wedge after \neq \Omega \wedge pred(before, arg, result, after)) : after = \Omega\}
\end{aligned}$$

We then substitute as follows:

$$\begin{aligned}
&s = \text{multisets of } \mathbb{N}\\
&before = \langle\!|\,|\!\rangle
\end{aligned}$$

$$\begin{aligned}
pred = m(put) = \lambda(before, arg, result, after) \bullet |\, before\,| < 2\\
\wedge\ after = before \cup \langle\!|\ arg\ |\!\rangle \wedge arg \in \{0,1\} \wedge result = nil
\end{aligned}$$

$$\begin{aligned}
&pre(m(put), s, before, arg)\\
&\quad = \exists\, result : data, after : s \bullet m(put)(before, arg, result, after)\\
&\quad = \exists\, result : data, after : \text{multisets of } \mathbb{N} \bullet m(put)(\langle\!|\,|\!\rangle, arg, result, after)\\
&\quad = \exists\, result : data, after : \text{multisets of } \mathbb{N} \bullet\ |\langle\!|\,|\!\rangle| < 2 \wedge after = \langle\!|\,|\!\rangle \cup \langle\!|\ arg\ |\!\rangle\\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ arg \in \{0,1\} \wedge result = nil\\
&\quad = \exists\, result : data, after : \text{multisets of } \mathbb{N} \bullet after = \langle\!|\ arg\ |\!\rangle \wedge arg \in \{0,1\} \wedge result = nil\\
&\quad = arg \in \{0,1\}
\end{aligned}$$

We then get the following:

$$\begin{aligned}
&f(\langle\!|\,|\!\rangle, m(put), \text{multisets of } \mathbb{N})\\
&\quad = \{(arg, result, after) : data \times (data \cup \{\perp\}) \times (\text{multisets of } \mathbb{N} \cup \{\Omega\})\\
&\qquad\qquad \mid (\langle\!|\,|\!\rangle \neq \Omega \wedge pre(m(op), \text{multisets of } \mathbb{N}, \langle\!|\,|\!\rangle, arg))\\
&\qquad\qquad\qquad ?(result \neq \perp \wedge after \neq \Omega \wedge m(op)(\langle\!|\,|\!\rangle, arg, result, after)) : after = \Omega\}\\
\\
&\quad = \{(arg, result, after) : data \times (data \cup \{\perp\}) \times (\text{multisets of } \mathbb{N} \cup \{\Omega\})\\
&\qquad\qquad \mid arg \in \{0,1\}\\
&\qquad\qquad\qquad ?(result \neq \perp \wedge after \neq \Omega\\
&\qquad\qquad\qquad\quad \wedge\ |\langle\!|\,|\!\rangle| < 2 \wedge after = \langle\!|\,|\!\rangle \cup \langle\!|\ arg\ |\!\rangle \wedge arg \in \{0,1\} \wedge result = nil)\\
&\qquad\qquad\qquad : after = \Omega\}\\
\\
&\quad = \{(arg, result, after) : data \times (data \cup \{\perp\}) \times (\text{multisets of } \mathbb{N} \cup \{\Omega\})\\
&\qquad\qquad \mid arg \in \{0,1\}\ ?\ (after = \langle\!|\ arg\ |\!\rangle \wedge result = nil) : after = \Omega\}\\
\\
&\quad = \{(0, nil, \langle\!|\ 0\ |\!\rangle), (1, nil, \langle\!|\ 1\ |\!\rangle),\\
&\qquad (nil, nil, \Omega), (nil, 0, \Omega), (nil, 1, \Omega), (nil, 2, \Omega), (nil, \perp, \Omega),\\
&\qquad (2, nil, \Omega), (2, 0, \Omega), (2, 1, \Omega), (2, 2, \Omega), (2, \perp, \Omega)\}
\end{aligned}$$

As discussed above, this set represents all the possible outcomes of invoking the put operation. In those cases where put's precondition for state $\langle\!|\,|\!\rangle$, which is $arg \in \{0,1\}$, holds, the outcome is as defined in put's predicate. In the other cases, we get triples representing all possible combinations of result value (including $\perp$), and with after-state Ω.

We can do the same thing with $op = get$. The predicate and precondition of get when

$\mathit{before} = \Lbag\Rbag$ is as follows:

$$\begin{aligned}
\mathit{pred} = m(\mathit{get}) = \lambda(&\mathit{before}, \mathit{arg}, \mathit{result}, \mathit{after}) \bullet |\, \mathit{before}\,| > 0 \\
&\wedge \mathit{before} = \mathit{after} \cup \Lbag\ \mathit{result}\ \Rbag \wedge \mathit{arg} = \mathit{nil} \wedge \mathit{result} \in \{0,1\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{pre}(m(\mathit{put}), s, \mathit{before}, \mathit{arg}) \\
&\quad = \exists\, \mathit{result} : \mathit{data}, \mathit{after} : \text{multisets of } \mathbb{N} \bullet\ |\Lbag\Rbag| > 0 \wedge \Lbag\Rbag = \mathit{before} \cup \Lbag\ \mathit{result}\ \Rbag \\
&\qquad\qquad \wedge \mathit{arg} = \mathit{nil} \wedge \mathit{result} \in \{0,1\} \\
&\quad = \exists\, \mathit{result} : \mathit{data}, \mathit{after} : \text{multisets of } \mathbb{N} \bullet \mathit{false} \\
&\quad = \mathit{false}
\end{aligned}$$

Since the precondition is false, the value of f is as follows:

$$\begin{aligned}
&f(\Lbag\Rbag, m(\mathit{get}), \text{multisets of } \mathbb{N}) \\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \mid \mathit{after} = \Omega\} \\
&\quad = \{(\mathit{nil}, \mathit{nil}, \Omega), (\mathit{nil}, 0, \Omega), (\mathit{nil}, 1, \Omega), (\mathit{nil}, 2, \Omega), (\mathit{nil}, \bot, \Omega), \\
&\qquad (0, \mathit{nil}, \Omega), (0,0,\Omega), (0,1,\Omega), (0,2,\Omega), (0,\bot,\Omega), \\
&\qquad (1, \mathit{nil}, \Omega), (1,0,\Omega), (1,1,\Omega), (1,2,\Omega), (1,\bot,\Omega), \\
&\qquad (2, \mathit{nil}, \Omega), (2,0,\Omega), (2,1,\Omega), (2,2,\Omega), (2,\bot,\Omega)\}
\end{aligned}$$

Since *xyzzy* is not mentioned in the specification, we take its predicate to be false. As a result, we know that its precondition must also be false, and therefore $f(\Lbag\Rbag, m(\mathit{xyzzy}), \text{multisets of } \mathbb{N})$ must be equal to $f(\Lbag\Rbag, m(\mathit{get}), \text{multisets of } \mathbb{N})$.

We can now substitute our results above for f and thus obtain the following value for g:

$$\begin{aligned}
&g(\Lbag\Rbag, \text{multisets of } \mathbb{N}, m) \\
&\quad = \{(\mathit{put}, 0, \mathit{nil}, \Lbag\ 0\ \Rbag), (\mathit{put}, 1, \mathit{nil}, \Lbag\ 1\ \Rbag), \\
&\qquad (\mathit{put}, \mathit{nil}, \mathit{nil}, \Omega), (\mathit{put}, \mathit{nil}, 0, \Omega), (\mathit{put}, \mathit{nil}, 1, \Omega), (\mathit{put}, \mathit{nil}, 2, \Omega), (\mathit{put}, \mathit{nil}, \bot, \Omega), \\
&\qquad (\mathit{put}, 2, \mathit{nil}, \Omega), (\mathit{put}, 2, 0, \Omega), (\mathit{put}, 2, 1, \Omega), (\mathit{put}, 2, 2, \Omega), (\mathit{put}, 2, \bot, \Omega), \\
&\qquad (\mathit{get}, \mathit{nil}, \mathit{nil}, \Omega), (\mathit{get}, \mathit{nil}, 0, \Omega), (\mathit{get}, \mathit{nil}, 1, \Omega), (\mathit{get}, \mathit{nil}, 2, \Omega), (\mathit{get}, \mathit{nil}, \bot, \Omega), \\
&\qquad (\mathit{get}, 0, \mathit{nil}, \Omega), (\mathit{get}, 0, 0, \Omega), (\mathit{get}, 0, 1, \Omega), (\mathit{get}, 0, 2, \Omega), (\mathit{get}, 0, \bot, \Omega), \\
&\qquad (\mathit{get}, 1, \mathit{nil}, \Omega), (\mathit{get}, 1, 0, \Omega), (\mathit{get}, 1, 1, \Omega), (\mathit{get}, 1, 2, \Omega), (\mathit{get}, 1, \bot, \Omega), \\
&\qquad (\mathit{get}, 2, \mathit{nil}, \Omega), (\mathit{get}, 2, 0, \Omega), (\mathit{get}, 2, 1, \Omega), (\mathit{get}, 2, 2, \Omega), (\mathit{get}, 2, \bot, \Omega), \\
&\qquad (\mathit{xyzzy}, \mathit{nil}, \mathit{nil}, \Omega), (\mathit{xyzzy}, \mathit{nil}, 0, \Omega), (\mathit{xyzzy}, \mathit{nil}, 1, \Omega), (\mathit{xyzzy}, \mathit{nil}, 2, \Omega), \\
&\qquad (\mathit{xyzzy}, \mathit{nil}, \bot, \Omega), \\
&\qquad (\mathit{xyzzy}, 0, \mathit{nil}, \Omega), (\mathit{xyzzy}, 0, 0, \Omega), (\mathit{xyzzy}, 0, 1, \Omega), (\mathit{xyzzy}, 0, 2, \Omega), (\mathit{xyzzy}, 0, \bot, \Omega), \\
&\qquad (\mathit{xyzzy}, 1, \mathit{nil}, \Omega), (\mathit{xyzzy}, 1, 0, \Omega), (\mathit{xyzzy}, 1, 1, \Omega), (\mathit{xyzzy}, 1, 2, \Omega), (\mathit{xyzzy}, 1, \bot, \Omega), \\
&\qquad (\mathit{xyzzy}, 2, \mathit{nil}, \Omega), (\mathit{xyzzy}, 2, 0, \Omega), (\mathit{xyzzy}, 2, 1, \Omega), (\mathit{xyzzy}, 2, 2, \Omega), (\mathit{xyzzy}, 2, \bot, \Omega)\}
\end{aligned}$$

Each of the above quadruples corresponds to a branch of the choice in the definition of $[\![\Lbag\Rbag, \mathit{bag}]\!]$:

$$\begin{aligned}
[\![\Lbag\Rbag, \mathit{bag}]\!] := \mathit{choice}\ &(\mathit{op}, \mathit{arg}, \mathit{result}, \mathit{after}) : g(\Lbag\Rbag, \text{multisets of } \mathbb{N}, m) \\
&[]\ \mathit{op_arg};\ \ \mathit{op_result};\ [\![\mathit{after}, \mathit{bag}]\!]
\end{aligned}$$

Using the value of g we have obtained, we can expand this definition as follows:

$$
\begin{aligned}
\llbracket \langle\!\langle\rangle\!\rangle, bag \rrbracket := {} & \mathit{put\_0};\ \mathit{put\_nil};\ \llbracket \langle\!\langle\ 0\ \rangle\!\rangle, bag \rrbracket \ [\,]\ \mathit{put\_1};\ \mathit{put\_nil};\ \llbracket \langle\!\langle\ 1\ \rangle\!\rangle, bag \rrbracket \\
& [\,]\ \mathit{put\_nil};\ \mathit{put\_nil};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{put\_nil};\ \mathit{put\_0};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{put\_nil};\ \mathit{put\_1};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{put\_nil};\ \mathit{put\_2};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{put\_nil};\ \mathit{put\_}\bot;\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{put\_2};\ \mathit{put\_nil};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{put\_2};\ \mathit{put\_0};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{put\_2};\ \mathit{put\_1};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{put\_2};\ \mathit{put\_2};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{put\_2};\ \mathit{put\_}\bot;\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{get\_nil};\ \mathit{get\_nil};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{get\_nil};\ \mathit{get\_0};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{get\_nil};\ \mathit{get\_1};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{get\_nil};\ \mathit{get\_2};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{get\_nil};\ \mathit{get\_}\bot;\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{get\_0};\ \mathit{get\_nil};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{get\_0};\ \mathit{get\_0};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{get\_0};\ \mathit{get\_1};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{get\_0};\ \mathit{get\_2};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{get\_0};\ \mathit{get\_}\bot;\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{get\_1};\ \mathit{get\_nil};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{get\_1};\ \mathit{get\_0};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{get\_1};\ \mathit{get\_1};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{get\_1};\ \mathit{get\_2};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{get\_1};\ \mathit{get\_}\bot;\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{get\_2};\ \mathit{get\_nil};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{get\_2};\ \mathit{get\_0};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{get\_2};\ \mathit{get\_1};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{get\_2};\ \mathit{get\_2};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{get\_2};\ \mathit{get\_}\bot;\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{xyzzy\_nil};\ \mathit{xyzzy\_nil};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{xyzzy\_nil};\ \mathit{xyzzy\_0};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{xyzzy\_nil};\ \mathit{xyzzy\_1};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{xyzzy\_nil};\ \mathit{xyzzy\_2};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{xyzzy\_nil};\ \mathit{xyzzy\_}\bot;\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{xyzzy\_0};\ \mathit{xyzzy\_nil};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{xyzzy\_0};\ \mathit{xyzzy\_0};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{xyzzy\_0};\ \mathit{xyzzy\_1};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{xyzzy\_0};\ \mathit{xyzzy\_2};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{xyzzy\_0};\ \mathit{xyzzy\_}\bot;\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{xyzzy\_1};\ \mathit{xyzzy\_nil};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{xyzzy\_1};\ \mathit{xyzzy\_0};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{xyzzy\_1};\ \mathit{xyzzy\_1};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{xyzzy\_1};\ \mathit{xyzzy\_2};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{xyzzy\_1};\ \mathit{xyzzy\_}\bot;\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{xyzzy\_2};\ \mathit{xyzzy\_nil};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{xyzzy\_2};\ \mathit{xyzzy\_0};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{xyzzy\_2};\ \mathit{xyzzy\_1};\ \llbracket \Omega, bag \rrbracket \ [\,]\ \mathit{xyzzy\_2};\ \mathit{xyzzy\_2};\ \llbracket \Omega, bag \rrbracket \\
& [\,]\ \mathit{xyzzy\_2};\ \mathit{xyzzy\_}\bot;\ \llbracket \Omega, bag \rrbracket
\end{aligned}
$$

This demonstrates how to translate a single state of the state-based specification into LOTOS using our translation. We will not describe the translation of the other states in this specification into LOTOS; they are translated in a similar manner to the above.

6.8 Conclusion

In this chapter, we brought together the ideas from §3.4 and chapters 4 and 5 to define our translation from our state-based notation to LOTOS. This translation can be taken to represent our ideas of a behavioural semantics for the state-based specification notation, given in terms of LOTOS, whose semantics are in turn defined in terms of labelled transition systems. The translation combines our understanding of the meaning of unspecified behaviour in state-based specifications, together with a technique, involving representing each operation by two consecutive actions, for producing nondeterminism in the return value of an operation, while leaving the choice of argument value fully determined by the environment.

We demonstrated our translation by translating part of the bag specification of §3.1.2 into LOTOS.

In the next chapter, we will attempt to prove that reduction corresponds to state-based subtyping.

Chapter 7

Proving that subtyping implies reduction

In the previous chapter, we solidified the ideas we presented in earlier chapters into a translation from the world of state-based specifications to the LOTOS process algebraic world. The translation reflects our understanding of the nature of nondeterminism and unspecified behaviour in state-based specifications.

Our original aim was to find a process algebraic relation corresponding to the state-based subtyping relation. In §4.2, we defined our notion of correspondence: for a process algebraic relation to correspond to state-based subtyping, we have to establish that a pair of state-based specifications are related by the state-based subtyping relation exactly when their translations into LOTOS are related by our process algebraic subtyping relation.

We will attempt to prove this in two parts: first, we must prove that the presence of subtyping between a pair of specifications in the state-based world implies reduction between the LOTOS translations of those specifications. Then we must prove the converse: that the presence of reduction between the LOTOS translations of two specifications implies that the original, untranslated specifications are related by subtyping.

First, we will attempt to prove the former of these two requirements. We will address the latter in §7.9.

7.1 Foundations

In §3.2.2, we defined our subtyping relation, which we will denote $\sqsubseteq$, on state-based specifications. In chapter 6, we defined a translation, $[\![\ldots]\!]$, from state-based specifications to LOTOS, and we have the standard LOTOS reduction relation, *red*, as described in §4.3.6. We now want to prove the following:

$$CSpec \sqsubseteq ASpec \Rightarrow [\![CSpec]\!] \ red \ [\![ASpec]\!]$$

The definition of *red* is as follows:

$$P \ red \ Q \Leftrightarrow \mathrm{Tr}(P) \subseteq \mathrm{Tr}(Q) \wedge \forall \sigma \in Act^* \bullet \mathrm{Ref}(P, \sigma) \subseteq \mathrm{Ref}(Q, \sigma)$$

Since *red* is defined as the conjunction of two terms, we can separate our hypothesis into two parts, and prove each separately:

$$CSpec \sqsubseteq ASpec \Rightarrow \mathrm{Tr}([\![CSpec]\!]) \subseteq \mathrm{Tr}([\![ASpec]\!])$$
$$CSpec \sqsubseteq ASpec \Rightarrow \forall \sigma \in Act^* \bullet \mathrm{Ref}([\![CSpec]\!], \sigma) \subseteq \mathrm{Ref}([\![ASpec]\!], \sigma)$$

We can restate this as follows:

$$\begin{aligned}
\mathit{CSpec} \sqsubseteq \mathit{ASpec} \Rightarrow \forall\, \sigma \in \mathit{Act}^* \bullet\ & (\sigma \in \mathrm{Tr}([\![\mathit{CSpec}]\!]) \Rightarrow \sigma \in \mathrm{Tr}([\![\mathit{ASpec}]\!])) \\
& \wedge (\mathrm{Ref}([\![\mathit{CSpec}]\!], \sigma) \subseteq \mathrm{Ref}([\![\mathit{ASpec}]\!], \sigma))
\end{aligned}$$

Using the property $\sigma \notin \mathrm{Tr}(P) \Rightarrow \mathrm{Ref}(P, \sigma) = \varnothing$ we can reduce our proof obligation to

$$\begin{aligned}
& \mathit{CSpec} \sqsubseteq \mathit{ASpec} \\
& \quad \Rightarrow \forall\, \sigma \in \mathit{Act}^* \bullet \sigma \in \mathrm{Tr}([\![\mathit{CSpec}]\!]) \\
& \qquad\qquad \Rightarrow (\sigma \in \mathrm{Tr}([\![\mathit{ASpec}]\!]) \wedge \mathrm{Ref}([\![\mathit{CSpec}]\!], \sigma) \subseteq \mathrm{Ref}([\![\mathit{ASpec}]\!], \sigma))
\end{aligned}$$

This gives us, for every trace in $[\![\mathit{CSpec}]\!]$, two things we need to prove: that the trace is in $[\![\mathit{ASpec}]\!]$, and that $[\![\mathit{CSpec}]\!]$'s refusals after the trace are a subset of $[\![\mathit{ASpec}]\!]$'s. We do not need to consider traces which are not in $[\![\mathit{CSpec}]\!]$.

To explain the proof, we will need to identify and name several structures which are related in the translated and untranslated specifications. In this section, we will make these connections.

We will first briefly revisit the definition of our translation. A specification (s, t, m) is translated as follows:

$$[\![(s,t,m)]\!] := \mathit{choice}\ e : t\ [\,]\ i;\ [\![e, (s,t,m)]\!]$$

$$\begin{aligned}
[\![e, (s,t,m)]\!] := \mathit{choice}\ & (\mathit{op}, \mathit{arg}, \mathit{result}, \mathit{after}) : g(e, s, m) \\
& [\,]\ \mathit{op_arg};\ \mathit{op_result};\ [\![\mathit{after}, (s,t,m)]\!]
\end{aligned}$$

$$\begin{aligned}
& g(\mathit{before}, s, m) \\
& \quad = \{(\mathit{op}, \mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{ops} \times \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \\
& \qquad\quad \mid (\mathit{arg}, \mathit{result}, \mathit{after}) \in f(\mathit{before}, m(\mathit{op}), s)\}
\end{aligned}$$

$$\begin{aligned}
& f(\mathit{before}, \mathit{pred}, s) \\
& \quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \\
& \qquad\quad \mid (\mathit{before} \neq \Omega \wedge \mathit{pre}(\mathit{pred}, s, \mathit{before}, \mathit{arg})) \\
& \qquad\qquad ?(\mathit{result} \neq \bot \wedge \mathit{after} \neq \Omega \wedge \mathit{pred}(\mathit{before}, \mathit{arg}, \mathit{result}, \mathit{after})) : \mathit{after} = \Omega\}
\end{aligned}$$

$$\begin{aligned}
& \mathit{pre}(\mathit{pred}, s, \mathit{before}, \mathit{arg}) \\
& \quad = \exists\, \mathit{result} : \mathit{data}, \mathit{after} : s \bullet \mathit{pred}(\mathit{before}, \mathit{arg}, \mathit{result}, \mathit{after})
\end{aligned}$$

Ω is a "fake" state where no operation's precondition is true; it is the state-based analog of *stop*. ($[\![\Omega, \ldots]\!]$ is *not stop*, of course, but is, in a sense, almost its opposite; it is the least-reduced process our translation can produce, as we will later prove.)

We will refer to the abstract and concrete state-based specifications as *ASpec* and *CSpec* respectively. $[\![\mathit{ASpec}]\!]$ is the translated version of *ASpec*, and $[\![\mathit{CSpec}]\!]$ is the translated version of *CSpec*. For our purposes, we will assume that the state-spaces of *ASpec* and *CSpec* must be disjoint, and that all the states have distinct names—if this is not the case, we can modify the specifications to achieve this by tagging each state with the name of the specification it belongs to. We will refer to states in *ASpec* as A_i, and states in *CSpec* as C_i, where i is drawn from some index set. (The examples we will examine mostly have small state-spaces, so we will normally use $\mathbb{N}$ for the index set.)

Since each operation invocation in a state-based specification *spec* corresponds to a pair of actions in $[\![\mathit{spec}]\!]$, and since $[\![\mathit{spec}]\!]$ contains many actions which do not correspond to an operation invocation in *spec*, we do not have a name for every state in $[\![\mathit{spec}]\!]$. Those states whose behaviour is of the form $[\![S_n, \mathit{spec}]\!]$, where S_n is a state in *spec*, correspond to S_n; we will usually refer to them as $[\![S_n]\!]$, omitting the specification name *spec* when it is not ambiguous to do so.

If the state-based specification *spec* can perform operation o, starting in state S_n, taking argument a and returning r, and thereby performing a transition to state S_m, this transition has a corresponding pair of actions in the translated specification, $[\![\mathit{spec}]\!]$. The transition from

S_n to S_m is matched by a pair of actions together making the transition from $[\![S_n]\!]$ to $[\![S_m]\!]$; specifically,

$$[\![S_n]\!] \xRightarrow{o\_a} \xRightarrow{o\_r} [\![S_m]\!]$$

This allows us to identify traces in a translated specification, $[\![spec]\!]$ with sequences of operation invocations in the original specification, *spec*.

7.2 States with only one initial

We will now present a property which we will use later in the proof. Consider the problem of finding the refusals of a process P after a trace σ. We first find P *after* σ, the set of states which P can become after performing the trace σ. The refusals we are trying to find are the union of the refusals for each of the states in P *after* σ.

Now consider the case where all the states in P *after* σ have exactly one initial; that is, they each have exactly one action which they can perform before evolving into another behaviour, and once we know which state in P *after* σ has been reached, we know exactly the action which is that state's initial. For any one of these states, the refusals are easy to determine: they are the set of all actions other than that state's initial, and subsets of that set.

The refusals of P after trace σ, then, consist of several sets of actions, each of which contains all but one of the actions in Act, and all subsets of those sets. For each refusal consisting of all but one of the actions in Act, the missing action is the one which one of the states in P *after* σ can perform.

Let us now examine an example. Consider the following process definitions:

$$Q := a;\ b;\ stop\ [\,]\ a;\ c;\ stop\ [\,]\ a;\ d;\ stop$$
$$R := a;\ b;\ stop\ [\,]\ a;\ d;\ stop$$

The labelled transition systems corresponding to these definitions are shown in figure 47. We will assume, for the moment, that $Act = \{a, b, c, d\}$.

After performing trace $\langle a\rangle$, Q may evolve to any of three states, each of which can perform one of the actions b, c and d. The refusals of the state which can accept b, for example, are $\{a, c, d\}$ and all subsets of it. The maximal refusals (that is, not including subsets of those shown) for each of the states in Q *after* $\langle a\rangle$ are shown in figure 47. Also shown (emboldened) are the combined maximal refusals from all the states each process can be in after performing $\langle a\rangle$. By "maximal" refusals, we mean only those refusals which are not a subset of another refusal.

We can see that the total set of refusals of P after trace σ is related directly to the possible actions which can be performed by the states in P *after* σ. In our example, there are three states which Q can reach after trace $\langle a\rangle$; their initials are b, c and d. After the same trace, R can be in one of two states whose initials are b and d. Since the refusals of these processes after $\langle a\rangle$ are directly related to their initials after $\langle a\rangle$, we do not need to know any more about the processes to tell that $\mathrm{Ref}(R, \langle a\rangle) \subseteq \mathrm{Ref}(Q, \langle a\rangle)$.

In general, if two processes, P and P', after a particular trace σ, can each evolve only to behaviours with a single initial each, then the refusals of P and P' after σ are directly related to the initials of those states. If the initials of the states in P *after* σ are a subset of the initials of the states in P' *after* σ, then the refusals of P after σ must be a subset of the refusals of P' after σ.

We will refer to this as the *property of singleton states.*

7.3 Properties of $[\![State]\!]$

We will now examine the definition of $[\![State]\!]$, and deduce some properties of the LOTOS processes which it produces. We will use the following specification, taken from the examples section at

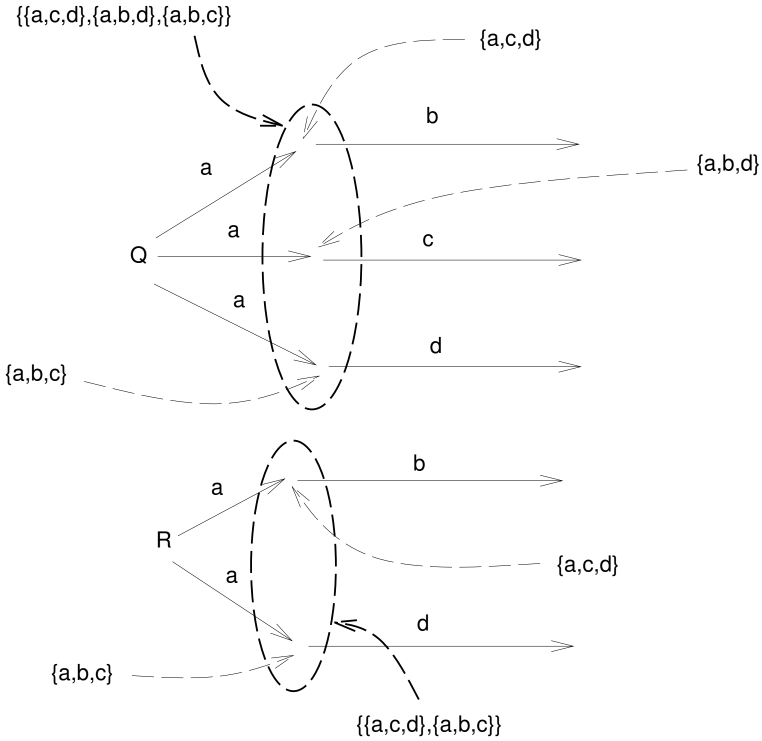

Figure 47: Processes Q and R, showing maximal refusals at certain points

the end of this chapter, as an example in this section:

Iterator :

State space: $\{T_1\}$
Initial states: T_1
Operation double $(\mathbb{N} \to \mathbb{N})$: $result = arg \times 2$
Operation inc $(\mathbb{N} \to \mathbb{N})$: $result = arg + 1$

For the purposes of this example, we will take $ops = \{double, inc\}$ and $data = \mathbb{N}$.

7.3.1 Traces of $[\![\Omega]\!]$ are a superset of traces of $[\![State]\!]$

In the main part of the proof, we will be following the structure of the abstraction relation, proving that, if C_s is related to A_n by Abs, then $[\![C_s]\!]$ *red* $[\![A_n]\!]$. However, sometimes, we will need to know that $[\![C_s]\!]$ *red* $[\![\Omega]\!]$. Ω, however, is a state which is introduced by the translation from the state-based world to LOTOS; it does not appear in the translated abstraction relation. We will therefore have to treat it specially: we need to show that any translated state $[\![S_i]\!]$ is a reduction of $[\![\Omega]\!]$. We will do that in this section.

We wish to show that $[\![\Omega]\!]$ is the least-reduced process which can be produced by our translation, $[\![\ldots]\!]$. To do this, we must show that both the traces and the refusals of $[\![\Omega]\!]$ are supersets of the traces and refusals of any other translated process, $[\![State]\!]$.

We will now perform the first step of this proof: to show that $\mathrm{Tr}([\![\Omega]\!]) \supseteq \mathrm{Tr}([\![State]\!])$.

The definition of $[\![State]\!]$ contains two auxiliary definitions, f and g, which we will look at first:

$$\begin{aligned}
&g(\mathit{before}, s, m) \\
&\quad = \{(\mathit{op}, \mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{ops} \times \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \\
&\qquad\qquad \mid (\mathit{arg}, \mathit{result}, \mathit{after}) \in f(\mathit{before}, m(\mathit{op}), s)\}
\end{aligned}$$

$$\begin{aligned}
&f(\mathit{before}, \mathit{pred}, s) \\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \\
&\qquad\qquad \mid (\mathit{before} \neq \Omega \wedge \mathit{pre}(\mathit{pred}, s, \mathit{before}, \mathit{arg})) \\
&\qquad\qquad\quad ?(\mathit{result} \neq \bot \wedge \mathit{after} \neq \Omega \wedge \mathit{pred}(\mathit{before}, \mathit{arg}, \mathit{result}, \mathit{after})) : \mathit{after} = \Omega\}
\end{aligned}$$

Firstly, let us examine these definitions under two sets of circumstances: those where $\mathit{before} = \Omega$, and those where $\mathit{before} \neq \Omega$. These circumstances arise when we are generating the definition of $[\![\mathit{state}, \mathit{spec}]\!]$, where $\mathit{state} = \Omega$ or $\mathit{state} \neq \Omega$ respectively.

If $\mathit{before} = \Omega$, we can simplify the definitions of f and g as follows:

$$\begin{aligned}
&f(\Omega, \mathit{pred}, s) \\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times \{\Omega\}\}
\end{aligned}$$

$$\begin{aligned}
&g(\Omega, s, m) \\
&\quad = \{(\mathit{op}, \mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{ops} \times \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \\
&\qquad\qquad \mid (\mathit{arg}, \mathit{result}, \mathit{after}) \in f(\Omega, m(\mathit{op}), s)\} \\
&\quad = \{(\mathit{op}, \mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{ops} \times \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times \{\Omega\}\} \\
&\quad = \mathit{ops} \times \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times \{\Omega\}
\end{aligned}$$

In other words, the values of each element in the tuples $(\mathit{op}, \mathit{arg}, \mathit{result}, \mathit{after})$ in g, which each translate into a branch in the behaviour of $[\![\Omega]\!]$, are constrained by the sets they are drawn from; respectively, the sets ops, data, $\mathit{data} \cup \{\bot\}$, and $\{\Omega\}$.

In our example specification, $\mathit{ops} = \{\mathit{double}, \mathit{inc}\}$ and $\mathit{data} = \mathbb{N}$, so that:

$$\begin{aligned}
g(\Omega, s, m) = \{&(\mathit{double}, 0, \bot, \Omega), (\mathit{double}, 0, 0, \Omega), (\mathit{double}, 0, 1, \Omega), \ldots \\
&(\mathit{double}, 1, \bot, \Omega), (\mathit{double}, 1, 0, \Omega), (\mathit{double}, 1, 1, \Omega), \ldots \\
&\ldots \\
&(\mathit{inc}, 0, \bot, \Omega), (\mathit{inc}, 0, 0, \Omega), (\mathit{inc}, 0, 1, \Omega), \ldots \\
&(\mathit{inc}, 1, \bot, \Omega), (\mathit{inc}, 1, 0, \Omega), (\mathit{inc}, 1, 1, \Omega), \ldots \\
&\ldots\}
\end{aligned}$$

g contains a tuple for every possible combination of operation, argument value and result value, including the case where the result is $\bot$ (which represents the case where the operation never terminates). We will come back to this shortly.

If $\mathit{before} \neq \Omega$, the definitions of f and g simplify as follows:

$$\begin{aligned}
&f(\mathit{before}, \mathit{pred}, s) \\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \\
&\qquad\qquad \mid (\mathit{pre}(\mathit{pred}, s, \mathit{before}, \mathit{arg})) \\
&\qquad\qquad\quad ?(\mathit{result} \neq \bot \wedge \mathit{after} \neq \Omega \wedge \mathit{pred}(\mathit{before}, \mathit{arg}, \mathit{result}, \mathit{after})) : \mathit{after} = \Omega\}
\end{aligned}$$

$$\begin{aligned}
&g(\mathit{before}, s, m) \\
&\quad = \{(\mathit{op}, \mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{ops} \times \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \\
&\qquad\qquad \mid (\mathit{arg}, \mathit{result}, \mathit{after}) \in f(\mathit{before}, m(\mathit{op}), s)\} \\
&\quad = \{(\mathit{op}, \mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{ops} \times \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \\
&\qquad\qquad \mid (\mathit{pre}(m(\mathit{op}), s, \mathit{before}, \mathit{arg})) \\
&\qquad\qquad\quad ?(\mathit{result} \neq \bot \wedge \mathit{after} \neq \Omega \wedge m(\mathit{op})(\mathit{before}, \mathit{arg}, \mathit{result}, \mathit{after})) : \mathit{after} = \Omega\}
\end{aligned}$$

In this case, the values of op, arg and result are drawn from the same sets as before, ops, data and $\mathit{data} \cup \{\bot\}$, but they are now constrained by the following requirements. If the precondition holds, the operation must return a value and progress to an after-state, both of which are consistent with the postcondition (and since $\perp$ and Ω are peculiar to the translation, they cannot result from a well-defined operation). If the precondition does not hold, the operation behaves as it would in state Ω.

In our example, when $state = T_1$,

$$\begin{aligned} g(T_1, s, m) = \{ & (double, 0, 0, T_1), (double, 1, 2, T_1), (double, 2, 4, T_1), \ldots \\ & (inc, 0, 1, T_1), (inc, 1, 2, T_1), (inc, 2, 3, T_1), \ldots \} \end{aligned}$$

We now refer to the definition of $[\![state]\!]$, in terms of g:

$$\begin{aligned} [\![e, (s, t, m)]\!] := & \ choice\ (op, arg, result, after) : g(e, s, m) \\ & [\,]\ op\_arg;\ op\_result;\ [\![after, (s, t, m)]\!] \end{aligned}$$

We can see that the traces of length 1 and 2 of $[\![state]\!]$ depend only on the values of op, arg and $result$ in the tuples in g. If we use this definition to calculate the definitions of $[\![\Omega, Iterator]\!]$ and $[\![T_1, Iterator]\!]$, we get the following:

$$\begin{aligned} [\![\Omega, Iterator]\!] := & \ double\_0;\ double\_\perp;\ [\![\Omega, Iterator]\!]\ [\,]\ double\_0;\ double\_0;\ [\![\Omega, Iterator]\!] \\ & [\,]\ double\_0;\ double\_1;\ [\![\Omega, Iterator]\!]\ [\,]\ \ldots \\ & [\,]\ double\_1;\ double\_\perp;\ [\![\Omega, Iterator]\!]\ [\,]\ double\_1;\ double\_0;\ [\![\Omega, Iterator]\!] \\ & [\,]\ double\_1;\ double\_1;\ [\![\Omega, Iterator]\!]\ [\,]\ \ldots \\ & \ldots \\ & [\,]\ inc\_0;\ inc\_\perp;\ [\![\Omega, Iterator]\!]\ [\,]\ inc\_0;\ inc\_0;\ [\![\Omega, Iterator]\!] \\ & [\,]\ inc\_0;\ inc\_1;\ [\![\Omega, Iterator]\!]\ [\,]\ \ldots \\ & [\,]\ inc\_1;\ inc\_\perp;\ [\![\Omega, Iterator]\!]\ [\,]\ inc\_1;\ inc\_0;\ [\![\Omega, Iterator]\!] \\ & [\,]\ inc\_1;\ inc\_1;\ [\![\Omega, Iterator]\!]\ [\,]\ \ldots \\ & \ldots \end{aligned}$$

$$\begin{aligned} [\![T_1, Iterator]\!] := & \ double\_0;\ double\_0;\ [\![T_1, Iterator]\!]\ [\,]\ double\_1;\ double\_2;\ [\![T_1, Iterator]\!] \\ & [\,]\ double\_2;\ double\_4;\ [\![T_1, Iterator]\!]\ [\,]\ \ldots \\ & [\,]\ inc\_0;\ inc\_1;\ [\![T_1, Iterator]\!]\ [\,]\ inc\_1;\ inc\_2;\ [\![T_1, Iterator]\!] \\ & [\,]\ inc\_2;\ inc\_3;\ [\![T_1, Iterator]\!]\ [\,]\ \ldots \end{aligned}$$

Labelled transition systems of these two processes are shown in figures 48 and 49. It is easy to see that the traces of $[\![T_1]\!]$ are a subset of the traces of $[\![\Omega]\!]$. We will now examine why this is.

Let us disregard the fourth element of each of the 4-tuples in g. From the definition of $[\![state, spec]\!]$, we can see that only the first three elements of each tuple in g influence the first or second action of any branch of the behaviour of a translated state. We will examine g', defined like this:

$$g'(before, s, m) = \{(op, arg, result) \mid \exists\ after \bullet (op, arg, result, after) \in g(before, s, m)\}$$

The process $[\![\Omega]\!]$ depends on the value of $g(\Omega, s, m)$, so we will examine the value of $g'(\Omega, s, m)$:

$$\begin{aligned} g'(\Omega, s, m) = \{ & (double, 0, \perp), (double, 0, 0), (double, 0, 1), \ldots \\ & (double, 1, \perp), (double, 1, 0), (double, 1, 1), \ldots \\ & \ldots \\ & (inc, 0, \perp), (inc, 0, 0), (inc, 0, 1), \ldots \\ & (inc, 1, \perp), (inc, 1, 0), (inc, 1, 1), \ldots \\ & \ldots \} \end{aligned}$$

For state T_1, we get the following:

$$\begin{aligned} g'(T_1, s, m) = \{ & (double, 0, 0), (double, 1, 2), (double, 2, 4), \ldots \\ & (inc, 0, 1), (inc, 1, 2), (inc, 2, 3), \ldots \} \end{aligned}$$

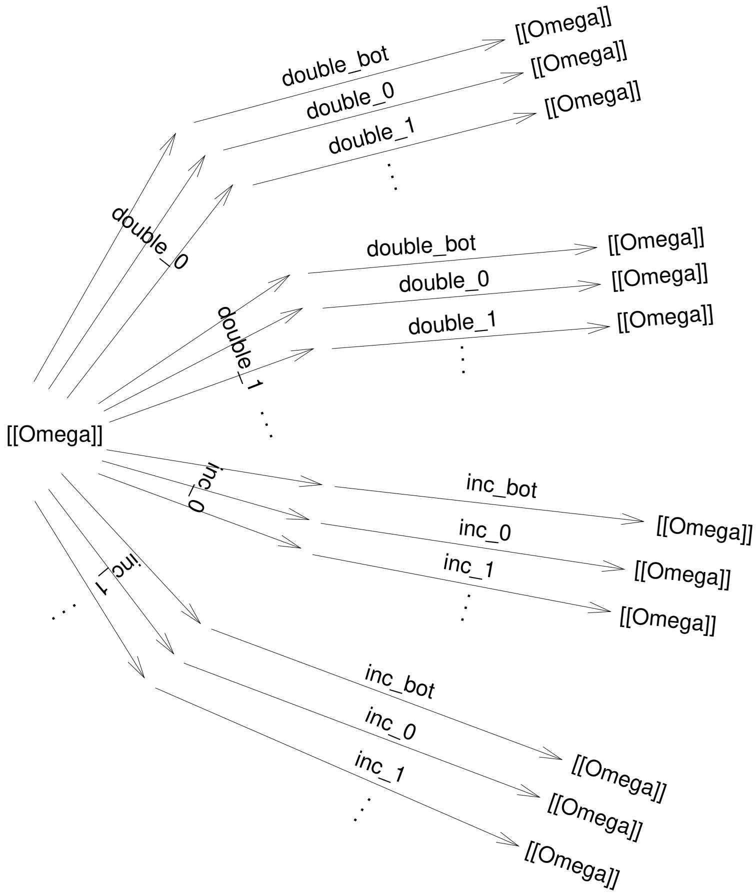

Figure 48: Labelled transition system for $[\![\Omega]\!]$

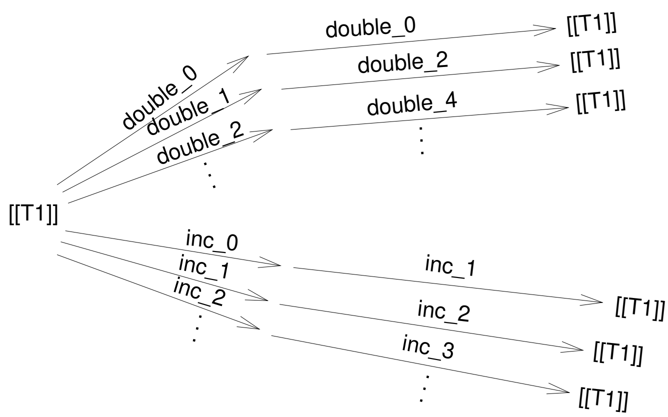

Figure 49: Labelled transition system for $[\![T_1]\!]$

We can see that $g'(T_1, s, m)$, is a subset of $g'(\Omega, s, m)$.

Since the traces of length not greater than 2 of $[\![state, spec]\!]$ depend only on these first three values of each tuple in g, we can conclude that those traces of length not greater than 2 in $[\![T_1]\!]$ must be a subset of those traces of length not greater than 2 in $[\![\Omega]\!]$.

We can now generalise this, to prove that the traces of $[\![\Omega]\!]$ are a subset of the traces of $[\![state]\!]$ for any *state*.

The values contained in the tuples of $g'(\Omega, \ldots)$—that is, the first three values of each tuple in $g(\Omega, \ldots)$—are, as mentioned above, constrained only by the bounds of the sets they are drawn from. When $before \neq \Omega$, the values are drawn from the same sets, but are subject to additional constraints. This means that, if we ignore the last element of each of the the 4-tuples in g, the value we get for $g(\Omega, \ldots)$ is a superset of the values we can get for $g(before, \ldots)$ with any value of $before \neq \Omega$.

Thus, $[\![\Omega]\!]$'s traces of length 1 and 2 must be a superset of those of $[\![state]\!]$, for any other *state*.

We can further see that the traces of $[\![state]\!]$ of length greater than 2 are formed by concatenating a trace of $[\![state]\!]$ of length 2 with a trace of one of the behaviours which can be reached from $[\![state]\!]$ by performing a trace of length 2. Such a behaviour is always of the form $[\![after]\!]$, where *after* is either Ω or is some state in the same specification as *state*. In the case where $state = \Omega$, *after* must always be Ω. It follows that the traces of $[\![\Omega]\!]$ are always a superset of the traces of $[\![state]\!]$, for any *state*.

7.3.2 Initials of $[\![state]\!]$ are independent of *state*

Before we examine the refusals of $[\![state]\!]$, we must first demonstrate a property of the set of initials of $[\![state]\!]$, namely that that set is independent of the value of *state*.

All actions which are initials of $[\![state]\!]$ are of the form o_a, where $o \in ops$ is the name of an operation, and $a \in data$ is an argument value passed to that operation. Whether a particular o_a is in $[\![state]\!]$'s initials depends on whether o and a appear as the first two elements in a tuple in $g(state, \ldots)$. Recall the definitions of f, g and *pre*:

$$
\begin{aligned}
&g(before, s, m)\\
&\quad = \{(op, arg, result, after) : ops \times data \times (data \cup \{\bot\}) \times (s \cup \{\Omega\})\\
&\qquad\qquad \mid (arg, result, after) \in f(before, m(op), s)\}
\end{aligned}
$$

$$
\begin{aligned}
&f(before, pred, s)\\
&\quad = \{(arg, result, after) : data \times (data \cup \{\bot\}) \times (s \cup \{\Omega\})\\
&\qquad\qquad \mid (before \neq \Omega \wedge pre(pred, s, before, arg))\\
&\qquad\qquad\qquad ?(result \neq \bot \wedge after \neq \Omega \wedge pred(before, arg, result, after)) : after = \Omega\}
\end{aligned}
$$

$$pre(pred, s, before, arg) = \exists\, result : data, after : s \bullet pred(before, arg, result, after)$$

From these definitions, it can be seen that there is always a tuple in g whose first two elements are o and a, for any $o \in ops$ and $a \in data$. Specifically, if the precondition holds, the definition of *pre* guarantees that there will exist values or r and c which produce an element in f containing o and a. That element of f will then be included in an element of g, which will in turn cause a branch of the behaviour $[\![before]\!]$ to begin with the action o_a.

If the precondition does not hold, or $before = \Omega$, the definition of f will contain many tuples containing o and a as their first two elements: specifically, there will be one for each value of $after \in s$. Each of those tuples will then give rise to a branch in the behaviour $[\![before]\!]$ beginning with action o_a, in a similar manner.

From this, we can see that the set of initials of $[\![state]\!]$ is, in fact, independent of the value of *state*; the initials of $[\![state]\!]$ are always the set of all actions o_a, where $o \in ops$ and $a \in data$.

Example Consider again the example specification, *Iterator*, which was translated into LOTOS in §7.3.1. Let us demonstrate by enumerating the initials of $[\![T_1]\!]$ and $[\![\Omega]\!]$. Looking first at $[\![T_1]\!]$, we see that its initials are

$$\begin{aligned} \mathit{initials}([\![T_1]\!]) = \{&\mathit{double\_0}, \mathit{double\_1}, \mathit{double\_2}, \ldots, \\ &\mathit{inc\_0}, \mathit{inc\_1}, \mathit{inc\_2}, \ldots\} \end{aligned}$$

If we now examine the definition of $[\![\Omega]\!]$, we can easily see that the initials are the same; $[\![T_1]\!] = [\![\Omega]\!]$.

Examining the LOTOS definitions of $[\![T_1]\!]$ and $[\![\Omega]\!]$, we can see that, despite these two states being quite different—Ω behaves as a state where no operation's precondition is ever true, while both *double*'s and *inc*'s preconditions are true in state T_1—both states have the exact same set of initials. This is not coincidence: as we have shown, it is a consequence of the definition of our translation.

7.3.3 Refusals of $[\![\Omega]\!]$ are a superset of refusals of $[\![\mathit{state}]\!]$

In §7.3.1, we showed that the traces of $[\![\Omega]\!]$ are a superset of the traces of $[\![\mathit{state}]\!]$, for any *state*. We now need to show that the refusals of $[\![\Omega]\!]$ after any trace σ are a superset of the refusals of $[\![\mathit{state}]\!]$ after the same trace.

We can break down the traces $[\![\Omega]\!]$ and $[\![\mathit{state}]\!]$ into three sets of traces: traces of length 0 (of which there is only one, $\langle\,\rangle$), traces of length 1, and traces whose length is 2 or greater. Of those traces whose length is 2 or greater, we will separate these into a set of traces of even length and a set of traces of odd length. We will now consider each of these cases in turn, and, for each case, show that the refusals of $[\![\Omega]\!]$ after each of the traces in the set are a superset of the refusals of $[\![\mathit{state}]\!]$ after the same trace.

Refusals after traces of length 0

We consider first the case where $\sigma = \langle\,\rangle$. In this case, $[\![\Omega]\!]$ *after* $\sigma = \{[\![\Omega]\!]\}$, and $[\![\mathit{state}]\!]$ *after* $\sigma = \{[\![\mathit{state}]\!]\}$. We saw above that the initials of $[\![\Omega]\!]$ and $[\![\mathit{state}]\!]$ are the same, and since neither state can perform an internal action, the refusals of each state are the complement of that state's initials (and, of course, all subsets of the complement of the state's initials). Thus the refusals of $[\![\Omega]\!]$ and $[\![\mathit{state}]\!]$ after $\langle\,\rangle$ are the same.

Example The initials of $[\![T_1]\!]$ are

$$\{\mathit{double\_0}, \mathit{double\_1}, \ldots, \mathit{inc\_0}, \mathit{inc\_1}, \ldots\}$$

We will take Act to be the set of all action names of the form o_a, where $o \in \mathit{ops}$ and $a \in \mathit{data} \cup \{\perp\}$. As mentioned in §7.3, $\mathit{ops} = \{\mathit{double}, \mathit{inc}\}$ and $\mathit{data} = \mathbb{N}$. Thus, the refusals of $[\![T_1]\!]$ after the trace $\langle\,\rangle$ are

$$\begin{aligned} \mathrm{Ref}([\![T_1]\!], \langle\,\rangle) &= \mathbb{P}\{\mathit{double\_0}, \mathit{double\_1}, \ldots, \mathit{inc\_0}, \mathit{inc\_1}, \ldots\}' \\ &= \mathbb{P}\{\mathit{double\_\perp}, \mathit{inc\_\perp}\} \end{aligned}$$

(We use the notation S' to refer to the complement of set S.)

The initials of $[\![\Omega]\!]$ are the same as those of $[\![T_1]\!]$, so its refusals after $\langle\,\rangle$ are also the same as those of $[\![T_1]\!]$.

Refusals after traces of length 1

When $\sigma = \langle o\_a \rangle$, for some $o \in \mathit{ops}$ and $a \in \mathit{data}$, we have the situation described in §7.2: all the states in $[\![\mathit{state}]\!]$ *after* σ are singleton states. We can see this easily from the definition of $[\![\mathit{state}]\!]$.

Since we know that the traces of $[\![\mathit{state}]\!]$ are a subset of the traces of $[\![\Omega]\!]$, and that each trace of length 2 of $[\![\mathit{state}]\!]$ corresponds to the initial of one of the states in $[\![\mathit{state}]\!]$ *after* σ, we know that the initials of states $[\![\mathit{state}]\!]$ *after* σ are a subset of the initials of states $[\![\Omega]\!]$ *after* σ, and therefore, because of the principle of singleton states (§7.2), that the refusals of $[\![state]\!]$ after σ are a subset of the refusals of $[\![\Omega]\!]$ after σ.

This proves that $\mathrm{Ref}([\![state]\!], \sigma) \subseteq \mathrm{Ref}([\![\Omega]\!], \sigma)$ for any trace σ where $|\sigma| < 2$.

Example Let us take $\sigma = \langle inc\_0 \rangle$. By performing this trace, $[\![T_1]\!]$ can evolve to only one behaviour, namely $inc\_1;\ [\![T_1]\!]$. The refusals of $[\![T_1]\!]$ *after* $\langle inc\_0 \rangle$ are therefore as follows:

$$\begin{aligned}\mathrm{Ref}([\![T_1]\!], \langle inc\_0 \rangle) &= \mathbb{P}\{inc\_1\}' \\ &= \mathbb{P}\{double\_\bot, double\_0, double\_1, \ldots, inc\_\bot, inc\_0, inc\_2, inc\_3, \ldots\}\end{aligned}$$

$[\![\Omega]\!]$, on the other hand, can, after performing $\langle inc\_0 \rangle$, become any of these behaviours:

$$inc\_\bot;\ [\![T_1]\!] \qquad inc\_0;\ [\![T_1]\!] \qquad inc\_1;\ [\![T_1]\!] \qquad inc\_2;\ [\![T_1]\!] \qquad \ldots$$

Each possible behaviour after the given trace contributes separately to the refusals after that trace. Thus, the refusals of $[\![\Omega]\!]$ *after* $\langle inc\_0 \rangle$ are as follows:

$$\begin{aligned}&\mathrm{Ref}([\![\Omega]\!], \langle inc\_0 \rangle) \\ &\quad = \mathbb{P}\{inc\_\bot\}' \cup \mathbb{P}\{inc\_0\}' \cup \mathbb{P}\{inc\_1\}' \cup \mathbb{P}\{inc\_2\}' \ldots \\ &\quad = \mathbb{P}\{double\_\bot, double\_0, double\_1, \ldots, inc\_0, inc\_1, inc\_2, inc\_3, \ldots\} \\ &\qquad \cup \mathbb{P}\{double\_\bot, double\_0, double\_1, \ldots, inc\_\bot, inc\_1, inc\_2, inc\_3, \ldots\} \\ &\qquad \cup \mathbb{P}\{double\_\bot, double\_0, double\_1, \ldots, inc\_\bot, inc\_0, inc\_2, inc\_3, \ldots\} \\ &\qquad \cup \mathbb{P}\{double\_\bot, double\_0, double\_1, \ldots, inc\_\bot, inc\_0, inc\_1, inc\_3, \ldots\} \\ &\qquad \ldots\end{aligned}$$

Clearly, the refusals of $[\![\Omega]\!]$ *after* $\langle inc\_0 \rangle$ are a superset of the refusals of $[\![T_1]\!]$ *after* $\langle inc\_0 \rangle$.

Refusals after traces of length 2, or greater

Even length Let us consider cases where $|\sigma| = 2n$ (where $n \in \mathbb{N}$). In these cases, all of the states in $[\![\Omega]\!]$ *after* σ are necessarily $[\![\Omega]\!]$; after performing any trace of two actions, $[\![\Omega]\!]$ can only become $[\![\Omega]\!]$. For example,

$$[\![\Omega]\!]\ \mathit{after}\ \langle inc\_0, inc\_1 \rangle = \{[\![\Omega]\!]\}$$
$$[\![\Omega]\!]\ \mathit{after}\ \langle inc\_0, inc\_1, double\_4, double\_\bot, inc\_2, inc\_3 \rangle = \{[\![\Omega]\!]\}$$

On the other hand, all the states in $[\![before]\!]$ *after* σ will be of the form $[\![after]\!]$ (where *after* is some state in the specification, or possibly Ω).

(For example, in the case of $before = T_1$, *after* is $[\![T_1]\!]$, for any even-length value of σ which $[\![T_1]\!]$ can perform.)

As shown previously, the refusals of $[\![\Omega]\!]$ *after* $\langle\rangle$ are a superset of those of $[\![state]\!]$ (for any *state*); therefore, the refusals of $[\![\Omega]\!]$ after any trace σ, where $|\sigma| = 2n$, are a superset of the refusals of $[\![state]\!]$ *after* σ.

Odd length The remaining cases are those where $|\sigma| = 2n + 1$, where $n \in \mathbb{N}$. As with the case where $|\sigma| = 1$, all the states in $[\![state]\!]$ *after* σ and in $[\![\Omega]\!]$ *after* σ are singleton states. Since we know that $\mathrm{Tr}([\![state]\!]) \subseteq \mathrm{Tr}([\![\Omega]\!])$, we can apply the same argument as we did in the case where $|\sigma| = 1$, and conclude that, where $|\sigma| = 2n + 1$, $\mathrm{Ref}([\![state]\!], \sigma) \subseteq \mathrm{Ref}([\![\Omega]\!], \sigma)$.

Conclusion

We have now covered all the cases; we have shown that $\mathrm{Ref}([\![state]\!], \sigma) \subseteq \mathrm{Ref}([\![\Omega]\!], \sigma)$ for all traces σ and all states *state*.

Combined with our earlier conclusion that $\mathrm{Tr}([\![state]\!]) \subseteq \mathrm{Tr}([\![\Omega]\!])$, we can conclude that $[\![state]\!]\ \mathrm{red}\ [\![\Omega]\!]$ for all states *state*. A complete example of this result can be found in §7.8.

7.4 Abstraction relation

We will now use the abstraction relation, Abs, between the original specifications, $ASpec$ and $CSpec$, to generate a partial abstraction relation, $[\![Abs]\!]$, between $[\![ASpec]\!]$ and $[\![CSpec]\!]$, which we will then use in explaining the proof. (Of course, the reduction relation does not normally need an abstraction relation. The abstraction relation we produce here is only used as a structure over which we can perform induction; it is not integral to the hypothesis we are proving.)

Our partial abstraction relation between $[\![ASpec]\!]$ and $[\![CSpec]\!]$ is very simple: if A_n and C_m are related by the state-based abstraction relation, then $[\![A_n]\!]$ and $[\![C_m]\!]$ are related by the new abstraction relation, $[\![Abs]\!]$.

Note that there is not necessarily a single abstraction relation between $ASpec$ and $CSpec$; any such relation which can be used to show that $CSpec$ is a subtype of $ASpec$ may be used.

7.5 States as processes

It will be useful, later on, for us to treat individual states within a specification as if they were whole specifications themselves; we now explain how we will do this.

Consider the following processes. The first has a single initial state; the second has more than one initial state.

Spec :

State space: $\{S_1, \ldots\}$
Initial states: S_1
...

Spec':

State space: $\{S'_1, S'_2, \ldots\}$
Initial states: S'_1, S'_2
...

The translated versions of these two specifications, $[\![Spec]\!]$ and $[\![Spec']\!]$, are as follows:

$[\![Spec]\!] := i;\ [\![S_1]\!]$
$[\![Spec']\!] := i;\ [\![S'_1]\!]\ []\ i;\ [\![S'_2]\!]$

Spec is a specification which has only one initial state, S_1; it is clear that $[\![Spec]\!]$ and $[\![S_1]\!]$ are weakly bisimilar (see §4.3.7). Because of this similarity between a translated specification with a single initial state and its translated initial state, we will treat states as specifications when it is convenient to do so; *State*, when treated as a specification, refers to the specification which *State* originally belonged to, but modified so that *State* is the single initial state of the new specification.

$Spec'$ has two initial states, S'_1 and S'_2. While $[\![Spec']\!]$ is not necessarily directly related to either of these two states alone, there is a relationship between all three. Specifically, the traces that $[\![Spec']\!]$ can perform are exactly those which can also be performed by one or both of $[\![S'_1]\!]$ and $[\![S'_2]\!]$. When $[\![Spec']\!]$ performs a trace, the sets of actions it can then refuse are exactly those which can be refused by either or both of $[\![S'_1]\!]$ and $[\![S'_2]\!]$ after the same trace.

In other words, the traces or refusals of a translated specification are the union of, respectively, the traces or refusals of each translated initial state of that specification. We say that the translated specification is the union of its translated initial states.

7.6 Concrete/abstract state pairs

The major part of this proof will involve showing that if, in the state-based world, $CSpec \sqsubseteq ASpec$ and $CState\ Abs\ AState$, then the LOTOS translations of those states will be related—specifically that $[\![CState]\!]$ red $[\![AState]\!]$. We will now set about proving this inductively.

The proof above that any translated state reduces $[\![\Omega]\!]$ will act as the base-case for the induction; the proof will be inductive over the structure of the two specifications, *ASpec* and *CSpec*, as linked together by the abstraction relation.

We have seen previously how each state, S_i, in the state-based specification corresponds to a state in the LOTOS translation, $[\![S_i]\!]$; how transitions which can be performed in the state-based specification correspond to pairs of actions in the translation; and that the state-based abstraction relation, *Abs*, translates to a corresponding partial abstraction relation $[\![Abs]\!]$.

Recall the definition of $[\![state]\!]$:

$$\begin{aligned}[\![e,(s,t,m)]\!] := {} & choice\ (op, arg, result, after) : g(e,s,m)\\ & \quad [\,]\ op\_arg;\ op\_result;\ [\![after,(s,t,m)]\!]\end{aligned}$$

$$\begin{aligned}& g(before, s, m)\\ & \quad = \{(op, arg, result, after) : ops \times data \times (data \cup \{\bot\}) \times (s \cup \{\Omega\})\\ & \qquad\quad \mid (arg, result, after) \in f(before, m(op), s)\}\end{aligned}$$

$$\begin{aligned}& f(before, pred, s)\\ & \quad = \{(arg, result, after) : data \times (data \cup \{\bot\}) \times (s \cup \{\Omega\})\\ & \qquad\quad \mid (before \neq \Omega \wedge pre(pred, s, before, arg))\\ & \qquad\qquad ?(result \neq \bot \wedge after \neq \Omega \wedge pred(before, arg, result, after)) : after = \Omega\}\end{aligned}$$

Notice that g always contains a tuple containing each possible combination of *op* and *arg*, where $o \in ops$ and $arg \in data$ (see §7.3.2). To recap briefly, the condition used to filter the members of f (the values of which are then replicated in g) is a conditional expression $(\ldots ? \ldots : \ldots)$. If *op*'s precondition holds with a particular value of *arg*, then the second part of the conditional expression applies and the postcondition necessarily holds for at least one combination of *result* and *after*. Otherwise, the third part of the conditional expression applies, in which case tuples will be produced containing all possible values of *result*, and with $after = \Omega$.

We will consider several cases, which can each be proved separately. We will divide the universe of traces into three parts according to their length: the first part consists of the trace whose length is 0 (which contains only the empty trace), the second consists of traces whose length is 1, and the third consists of traces whose length is 2 or greater. For each of these groups of traces, we will show first that all the traces in that group which $[\![CState]\!]$ can perform can also be performed by $[\![AState]\!]$. We will then show that, for any of the traces in the group, the refusals of $[\![CState]\!]$ after the trace are a subset of the refusals of $[\![AState]\!]$ after the same trace.

7.6.1 Example

We will use the following specification, taken from §7.8, as an example throughout this section:

Doubler :

State space: $\{D_1, D_2\}$
Initial states: D_1
Operation double $(\mathbb{N} \rightarrow \mathbb{N})$: $before = D_1 \wedge result = arg \times 2 \wedge after = D_2$

For this example, we will take $ops = \{double, inc\}$ and $data = \mathbb{N}$. Since the operation *inc* is not mentioned in the specification of *Doubler*, we take its precondition—indeed, its entire predicate—to be *false*, meaning that the operation's behaviour is unspecified.

7.6.2 Traces of length 0

Both $[\![CState]\!]$ and $[\![AState]\!]$ can perform the trace $\langle\,\rangle$, which is the only trace of length 0, by definition. This must be true, since *every* process can perform the trace $\langle\,\rangle$, which involves simply doing nothing. Trace subsetting is thus trivially proved for this case.

7.6.3 Refusals after $\langle\,\rangle$

As described in §7.3.2, the initials of $[\![state]\!]$ do not depend on the details of the specification, only on the definitions of *data* and *ops*.

Since *data* and *ops* are "universe" sets, which we require are kept the same between any translated processes which we want to compare, we can conclude that the initials of $[\![CSpec]\!]$ and $[\![ASpec]\!]$ are equal. Since, as can be seen from the definition above, no internal actions appear in any behaviour of the form $[\![state]\!]$, $[\![state]\!]$'s refusals after $\langle\,\rangle$ consist of exactly the set of actions which are *not* the initials of $[\![state]\!]$, and all subsets of that set. Therefore, to prove that $\mathrm{Ref}([\![CSpec]\!], \langle\,\rangle) \subseteq \mathrm{Ref}([\![ASpec]\!], \langle\,\rangle)$, we simply need to prove that the initials of $[\![CSpec]\!]$ are a superset of the initials of $[\![ASpec]\!]$. Since the initials of $[\![CSpec]\!]$ and $[\![ASpec]\!]$ are the same, this is trivially proved.

Example From the definitions of $[\![D_1]\!]$ and $[\![D_2]\!]$ worked out fully in §7.8, we can see that the initials of $[\![D_1]\!]$ are

$$\{double\_0, double\_1, double\_2, \ldots, inc\_0, inc\_1, inc\_2, \ldots\}$$

Examining the definition of $[\![D_2]\!]$, we can see that its initials are the same. If the environment offers one of these processes an action in this common set of initials, the process will be able to perform that action and advance to a new state; if the only actions offered are not in that set, they will be refused. We can thus clearly see that both $[\![D_1]\!]$ and $[\![D_2]\!]$ have the same refusals after $\langle\,\rangle$.

7.6.4 Traces of length 1

Since $[\![state]\!]$ contains no internal actions, its traces of length 1 consist of its initials. Since, as seen in §7.3.2, these are the same in both $[\![CSpec]\!]$ and $[\![ASpec]\!]$, the traces of length 1 of these two processes are the same, and are therefore each a subset of the other.

Example In our example, the set of traces is, in the cases of both $[\![D_1]\!]$ and $[\![D_2]\!]$,

$$\{\langle double\_0\rangle, \langle double\_1\rangle, \langle double\_2\rangle, \ldots, \langle inc\_0\rangle, \langle inc\_1\rangle, \langle inc\_2\rangle, \ldots\}$$

These traces correspond to the initials of these processes, which were shown in the previous subsection.

7.6.5 Refusals after traces of length 1

All states which can be reached from $[\![state]\!]$ by performing a single action share this property: they each have only one initial action. This can easily be seen from the definition of $[\![state]\!]$: each branch of the behaviour of $[\![state]\!]$ is of the form *op_arg*; *op_result*; $[\![after]\!]$. Therefore, for each state which can be reached from $[\![state]\!]$ by performing a single action, the refusals consist of the set of all actions other than that single action which is the initial of the state in question, and all subsets of that set. So, given the initial of that state, the refusals in that state are entirely determined. This is the situation which we examined in detail in §7.2.

Since each of these states is reached by a trace of length 1, and each has a single initial, we can set up a one-to-one relationship between each of these states and a trace of length 2 which the translated state can perform. Each such trace of length 2 thus corresponds to single state with one initial.

Therefore, in order to prove that the refusals of $[\![CSpec]\!]$ after a given trace of length 1 are a subset of the refusals of $[\![ASpec]\!]$ after the same trace, it suffices to show that the possible initials of $[\![CSpec]\!]$ after the trace are a subset of the possible initials of $[\![ASpec]\!]$ after the trace. Since each initial corresponds to a trace of length 2, we can equivalently show that the possible traces of length 2 of $[\![CSpec]\!]$ are a subset of the traces of length 2 of $[\![ASpec]\!]$. We will do this next.

Example $[\![D_1]\!]$ can perform the trace $\langle double\_1, double\_2\rangle$. After performing the first action, $double\_1$, $[\![D_1]\!]$ therefore evolves to a behaviour whose only initial is $double\_2$, and whose refusals comprise all sets of actions not containing $double\_2$.

$[\![D_2]\!]$, on the other hand, can reach many different states by performing $double\_1$. One of these has initial $double\_\perp$, another has initial $double\_0$, yet another has initial $double\_1$, and so on. Each of these states adds more refusals; so the more initials the process has after performing the trace, the more its refusals will be after that trace.

7.6.6 Traces of length 2 and above

As explained in §7.1, any trace of length 2, of the form $\langle op\_arg;\ op\_result\rangle$, in $[\![CState]\!]$ corresponds to an invocation of operation *op*, with before-state *CState*, argument *arg* and return value *result*. We will consider the case when this trace is performed and

$$[\![CState]\!] \xRightarrow{op\_arg} \xRightarrow{op\_result} [\![CState']\!]$$

This corresponds to the case where *CState* performs the operation invocation and results in after-state $CState'$, or, if $CState' = \Omega$, the case there *CState* is not specified to be able to perform the operation. We will also look at traces whose length is greater than 2. We will call the trace we are considering σ. The first two actions of σ are op_arg and op_result. All the remaining actions of σ after the first two are removed we will call σ'.

We will now show that, for traces whose length is 2 or greater, $\mathrm{Tr}([\![CState]\!]) \subseteq \mathrm{Tr}([\![AState]\!])$ and $\mathrm{Ref}([\![CState]\!], \sigma) \subseteq \mathrm{Ref}([\![AState]\!], \sigma)$. For each possible trace σ, there are three cases to consider. The cases are: where *op*'s precondition does not hold in *CState*; where *op*'s precondition does not hold in *AState*; and where *op*'s precondition holds in both *AState* and *CState*. (There is, of course, some overlap between the first two of these cases.) We will now examine each of these three cases in turn.

Case 1: where *op*'s precondition does not hold in *CState*

If *op*'s precondition does not hold in *CState* with argument *arg*, then the translation allows any value to be returned, including the value $\perp$ which cannot normally be returned by an operation. The after-state, however, is required to be Ω, which ensures that further actions are similarly unconstrained.

(An example of this case is where $CState = I_2$, $AState = D_2$, and $op = inc$. *inc* cannot be invoked on I_2, because the precondition is *false*. $[\![I_2]\!]$, therefore, can perform the trace $\langle inc\_1, inc\_\perp\rangle$ and thereby become $[\![\Omega]\!]$.)

Given that *CSpec* is a subtype of *ASpec* and that *CState* and *AState* are related by *Abs*, the first condition of the state-based subtyping relation (see §3.2.2) ensures that if the precondition holds in *AState*, it must also hold in *CState*. In the cases we are considering, where the precondition does not hold in *CState*, this means that the precondition cannot hold in *AState* either. Therefore, both $[\![AState]\!]$ and $[\![CState]\!]$ can perform the trace $\langle op\_arg;\ op\_result\rangle$, for any $result \in data \cup \{\perp\}$, and both evolve to $[\![\Omega]\!]$ by so doing.

(In our example, the processes $[\![I_2]\!]$ and $[\![D_2]\!]$ can both perform traces such as $\langle inc\_1, inc\_\perp\rangle$ or $\langle inc\_1, inc\_2\rangle$, and thereby evolve to $[\![\Omega]\!]$.)

We can conclude that if $[\![CState]\!]$ can perform a trace $\langle op\_arg;\ op\_result \rangle$, and *op*'s precondition does not hold in state *CState*, then $[\![AState]\!]$ must also be able to perform the same trace. Furthermore, since the trace causes both $[\![AState]\!]$ and $[\![CState]\!]$ to evolve to $[\![\Omega]\!]$, both processes have the same refusals after the trace; namely, the same refusals as $[\![\Omega]\!]$ after $\langle\,\rangle$.

We must now consider traces longer than 2 which start with $\langle op\_arg, op\_result \rangle$. If we remove $\langle op\_arg, op\_result \rangle$ from the start of σ, the trace we are considering, and name the remainder σ', we can see that for $[\![CState]\!]$ to perform σ, it must first perform $\langle op\_arg, op\_result \rangle$, thereby becoming $[\![\Omega]\!]$, then perform σ'. To perform σ, $[\![AState]\!]$ must similarly perform $\langle op\_arg, op\_result \rangle$, thus becoming $[\![\Omega]\!]$, and then perform σ'. Since both $[\![CState]\!]$ and $[\![AState]\!]$ can perform $\langle op\_arg, op\_result \rangle$ and become $[\![\Omega]\!]$, it must therefore be the case that any trace starting with $\langle op\_arg, op\_result \rangle$ which $[\![CState]\!]$ can perform must necessarily be a trace which $[\![AState]\!]$ can perform, too. For the same reason, the refusals of $[\![CState]\!]$ must be the same as the refusals of $[\![AState]\!]$, after any trace beginning with $\langle op\_arg;\ op\_result \rangle$.

(In our example, consider a trace $\sigma = \langle inc\_1, inc\_3, double\_2 \rangle$. $[\![I_2]\!]$ can perform σ and so become any of the states in $[\![\Omega]\!]$ *after double_2*. Similarly, $[\![D_2]\!]$ can perform trace σ, and become any of the states in $[\![\Omega]\!]$ *after double_2*.)

This gives us the trace and refusal subsetting properties we require for case 1.

Case 2: where *op*'s precondition does not hold in *AState*

If *op*'s precondition does not hold in *AState*, then it may or may not hold in *CState*. We already considered the case where the precondition holds in neither *AState* nor *CState* in case 1, so we now consider the case where *op*'s precondition holds in *CState* but not *AState*.

(An example of this case is that where $CState = I_1$, $AState = D_1$ and $op = inc$. In D_1, *inc*'s precondition is false; in I_1, it is *true*.)

Since *op*'s precondition holds in *CState*, $CState'$ must be a state other than Ω. *result* must be a return value which can be returned if *op* is called with argument *arg* in *CState*, because we are only considering those traces which *CState* can perform, and which are covered by the case we are considering. In *AState*, *op*'s precondition does not hold, so $[\![AState]\!]$ must be able to perform the trace $\langle op\_arg;\ op\_result \rangle$, and thereby become $[\![\Omega]\!]$. As was explained in detail in §7.3, $[\![\Omega]\!]$ can perform any trace, and then refuse any actions, which $[\![State]\!]$ can, for any *State*. We can also explain §7.3's result by noting that Ω behaves like a state where no operation's precondition holds; thus, the argument being made in case 1 above and in this case (case 2) can be applied to show that $[\![\Omega]\!]$ can perform every trace, and refuse every set of actions after those traces, which $[\![CState']\!]$ could perform or refuse.

Through reasoning similar to that in case 1, and with the knowledge that $[\![CState']\!]$ red $[\![\Omega]\!]$ we can see that any trace which $[\![CState]\!]$ can perform which starts with $\langle op\_arg, op\_result \rangle$, where *op*'s precondition does not hold in *AState* with argument *arg*, can also be performed by $[\![AState]\!]$, and any actions which $[\![CState]\!]$ can refuse after such a trace can be similarly refused by $[\![AState]\!]$.

(In our example, $[\![I_1]\!]$ can perform the trace $\langle inc\_0, inc\_1 \rangle$. Since *inc*'s precondition is *true* in this case, the behaviour it becomes after this trace is not $[\![\Omega]\!]$, but rather is $[\![I_2]\!]$. $[\![D_1]\!]$'s precondition is *false*, so it becomes $[\![\Omega]\!]$ after the same trace. We know from §7.3 that $[\![I_2]\!]$ red $[\![\Omega]\!]$, so we can conclude that any traces starting with *inc_0* which $[\![I_1]\!]$ can perform can also be performed by $[\![D_1]\!]$, and that any actions which $[\![I_1]\!]$ can refuse after such a trace can also be refused by $[\![D_1]\!]$ after the same trace.)

Case 3: where *op*'s precondition holds in both *AState* and *CState*

The last case to consider is that where *op*'s precondition holds in both *AState* and *CState*.

(An example of this case is where $CState = T_1$, $AState = I_1$. The precondition in both of these states is *true*.)

We now have to consider the second condition of state-based subtyping which states that, if *CState* can perform operation *op* with argument *arg*, producing return value *result* and after-state $CState'$, then *AState* can do the same, but with after-state $AState'$, such that $CState'$ and

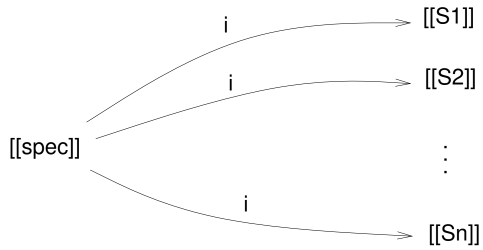

Figure 50: Choice of initial states in a translated specification

$AState'$ will be related by the abstraction relation.

Translating this into LOTOS terms, this means that for any *result*, if $[\![CState]\!]$ can perform $\langle op\_arg;\ op\_result\rangle$ and become $[\![CState']\!]$, $[\![AState]\!]$ can do the same and become $[\![AState']\!]$. By induction, and because $CState'$ is related to $AState'$ by Abs, we can assume that $[\![CState']\!]$ reduces $[\![AState']\!]$. Since both $[\![CState]\!]$ and $[\![AState]\!]$ can perform the trace $\langle op\_arg;\ op\_result\rangle$ and become $[\![CState']\!]$ and $[\![AState']\!]$ respectively, and because these two behaviours are related by reduction, we can apply similar reasoning to that used in cases 1 and 2, and conclude that any trace which $[\![CState]\!]$ can perform which starts with $\langle op\_arg;\ op\_result\rangle$ can also be done by $[\![AState]\!]$, and that any set of actions which $[\![CState]\!]$ can refuse after performing a trace starting with $\langle op\_arg;\ op\_result\rangle$ can also be refused by $[\![AState]\!]$ after the same trace.

(In our example, both $[\![T_1]\!]$ and $[\![I_1]\!]$ can perform the trace $\langle double\_1, double\_2\rangle$. After that trace, $[\![T_1]\!]$ becomes $[\![T_1]\!]$, and $[\![I_1]\!]$ becomes $[\![I_2]\!]$. Because this entire proof can be applied to any two states linked by the abstraction relation, we can inductively deduce that $[\![T_1]\!]\ \mathrm{red}\ [\![I_2]\!]$ and from that conclude that $[\![T_1]\!]\ \mathrm{red}\ [\![I_1]\!]$.)

7.6.7 Conclusion

We have now covered all the cases, including all possible traces (of length 0, length 1 and length 2 and greater), and proved for each case that, if $[\![CState]\!]$ can perform such a trace then $[\![AState]\!]$ will also be able to perform it, and that any actions which $[\![CState]\!]$ can refuse after performing the trace, $[\![AState]\!]$ can refuse, too. Thus we have proved that if *CState* is a subtype of *AState*—that is, if *CSpec* is a subtype of *ASpec*, and *CState* is a state in *CSpec* and *AState* is a state in *ASpec*, and if *CState* and *AState* are related by the abstraction relation—then $[\![CState]\!]\ \mathrm{red}\ [\![AState]\!]$.

7.7 Initial states

In the previous section, we proved that, if any two states *AState* and *CState* in the specifications *ASpec* and *CSpec* are related by an abstraction relation *Abs* which can be used to show that *CSpec* is a subtype of *ASpec*, then it follows that $[\![CState]\!]\ \mathrm{red}\ [\![AState]\!]$. We now wish to show that it is also the case that $[\![CSpec]\!]\ \mathrm{red}\ [\![ASpec]\!]$.

Recall the definition of $[\![spec]\!]$, where $spec = (s, t, m)$:

$$[\![(s, t, m)]\!] := choice\ e : t\ []\ i;\ [\![e, (s, t, m)]\!]$$

In other words, the process $[\![spec]\!]$ starts, nondeterministically, in one of *spec*'s initial states. The labelled transition system is shown in figure 50, where S_1, S_2, ..., S_n are the initial states of *spec*.

We can easily see that the traces of $[\![spec]\!]$ are exactly the union of the traces of $[\![S_i]\!]$ for all S_i which are initial states of *spec*. Similarly, for any trace σ, $\mathrm{Ref}([\![spec]\!], \sigma)$ is the union of $\mathrm{Ref}([\![S_i]\!], \sigma)$ for all initial states S_i of *spec*.

Consider, for example, that *spec* has two initial states S_1 and S_2. Imagine that $[\![S_1]\!]$ could perform the trace $\langle a\_1, a\_2, b\_2\rangle$ and then refuse $\{b\_1\}$, and that $[\![S_2]\!]$ could perform the same trace and then refuse $\{b\_2\}$. We can see, then, that $[\![spec]\!]$ could also perform the same trace, then refuse either $\{b\_1\}$ or $\{b\_2\}$.

To summarise: the traces of $[\![spec]\!]$, and its refusals after those traces, are the union of the traces and refusals of $[\![S_i]\!]$, for each initial state S_i of *spec*. This applies to the case of the empty trace, too: $[\![spec]\!]$ can refuse, after the empty trace, any set of actions which any $[\![S_i]\!]$ can refuse after the empty trace.

We want to prove that any trace which $[\![CSpec]\!]$ can perform can also be performed by $[\![ASpec]\!]$. Recall the third condition of state-based subtyping: any initial state of *CSpec* must be related by the abstraction relation to an initial state of *ASpec*. Any trace that $[\![CSpec]\!]$ can perform must be a trace of one of the translated initial states of *CSpec*, and each of those states is related by *Abs* to at least one initial state of *ASpec*. Since we proved above that if *AState* and *CState* are related by *Abs*, their translations must be related by *red*, it follows that $[\![ASpec]\!]$ must be able to perform the trace, too.

We can prove similarly that any set of actions which can be refused by $[\![CSpec]\!]$ after a trace σ can also be refused by $[\![ASpec]\!]$ after σ. If $[\![CSpec]\!]$ can perform σ and then refuse a set of actions R, then this must be because the translation of one of the initial states of *CSpec* can refuse R after σ. There must also be an initial state of *ASpec* related by *Abs* to each initial state of *CSpec*. We also know that if some *AState* and *CState* are related by *Abs* then $[\![CState]\!]$ red $[\![AState]\!]$. We can conclude that there must be an initial state of $[\![ASpec]\!]$ which can perform σ and then refuse R.

This completes the proof that if $CSpec \sqsubseteq ASpec$ then $[\![CSpec]\!]$ red $[\![ASpec]\!]$.

7.8 Examples

In this section, we will examine several larger examples, to see how the translation from state-based to process algebra works, and to show that translating state-based subtypes really does produce LOTOS reductions.

In the following examples, $ops = \{double, inc\}$ and $data = \mathbb{N}$. First, we will look at this state-based specification:

Doubler :

State space: $\{D_1, D_2\}$
Initial states: D_1
Operation double ($\mathbb{N} \to \mathbb{N}$): $before = D_1 \wedge result = arg \times 2 \wedge after = D_2$

Doubler has two states, and a single operation, *double*. The other operation in *ops*, *inc*, is not mentioned, so its predicate (and thus its precondition) is taken to be *false*, meaning that its behaviour is unspecified.

Firstly, we will rewrite the specification in the tuple notation described in §6.5:

$$s = \{D_1, D_2\}$$
$$t = \{D_1\}$$
$$\begin{aligned} m = \{&(double, \lambda(before, arg, result, after) \bullet before = D_1 \wedge after = D_2 \wedge result = arg \times 2),\\ &(inc, \lambda(before, arg, result, after) \bullet false)\}\end{aligned}$$
$$Doubler = (s, t, m)$$

(It is safe to drop the implicit $arg, result \in \mathbb{N}$ from the predicate of *double*, because $data = \mathbb{N}$.)

We can write down the behaviour of the translated specification immediately, up to a point:

$$[\![Doubler]\!] := i;\ [\![D_1, Doubler]\!]$$

To complete the translation, we will need to translate state D_1, D_2 and Ω. For variety, let us translate D_2 first. We now determine the value of $g(D_2, s, m)$, which is needed to translate state D_2:

$$\begin{aligned} g(D_2, s, m) = \{ & (op, arg, result, after) : ops \times data \times (data \cup \{\bot\}) \times (s \cup \{\Omega\}) \\ & \mid (arg, result, after) \in f(D_2, m(op), s)\} \end{aligned}$$

Since *double*'s predicate includes the term $before = D_1$, its precondition in state D_2 is false:

$$\begin{aligned} & pre(m(double), s, D_2, arg) \\ & \quad = \exists\, result : data, after : s \bullet m(double)(D_2, arg, result, after) \\ & \quad = false \end{aligned}$$

We now need to calculate two values for f, one for each of the two operations in ops:

$$\begin{aligned} & f(D_2, m(double), s) \\ & \quad = \{(arg, result, after) : data \times (data \cup \{\bot\}) \times (s \cup \{\Omega\}) \\ & \qquad\quad \mid (before \neq \Omega \wedge pre(m(double), s, D_2, arg)) \\ & \qquad\qquad ?(result \neq \bot \wedge after \neq \Omega \wedge m(double)(D_2, arg, result, after)) \\ & \qquad\qquad : after = \Omega\} \\ & \quad = \{(arg, result, after) : data \times (data \cup \{\bot\}) \times (s \cup \{\Omega\}) \\ & \qquad\quad \mid (before \neq \Omega \wedge false) \\ & \qquad\qquad ?\ldots \\ & \qquad\qquad : after = \Omega\} \\ & \quad = \{(arg, result, after) : data \times (data \cup \{\bot\}) \times (s \cup \{\Omega\}) \mid after = \Omega\} \\ & \quad = \{(0, \bot, \Omega), (0, 0, \Omega), (0, 1, \Omega), (0, 2, \Omega), (0, 3, \Omega), \ldots \\ & \qquad (1, \bot, \Omega), (1, 0, \Omega), (1, 1, \Omega), (1, 2, \Omega), (1, 3, \Omega), \ldots \\ & \qquad (2, \bot, \Omega), (2, 0, \Omega), (2, 1, \Omega), (2, 2, \Omega), (2, 3, \Omega), \ldots \\ & \qquad \ldots\} \end{aligned}$$

$$\begin{aligned} & f(D_2, m(inc), s) \\ & \quad = f(D_2, false, s) \\ & \quad = \{(0, \bot, \Omega), (0, 0, \Omega), (0, 1, \Omega), (0, 2, \Omega), (0, 3, \Omega), \ldots \\ & \qquad (1, \bot, \Omega), (1, 0, \Omega), (1, 1, \Omega), (1, 2, \Omega), (1, 3, \Omega), \ldots \\ & \qquad (2, \bot, \Omega), (2, 0, \Omega), (2, 1, \Omega), (2, 2, \Omega), (2, 3, \Omega), \ldots \\ & \qquad \ldots\} \end{aligned}$$

Since *inc*'s precondition is false, and *double*'s precondition is $before = D_1$, the over-all precondition in state D_2 is *false*. This gives us the following value for g:

$$\begin{aligned} g(D_2, s, m) = \{ & (double, 0, \bot, \Omega), (double, 0, 0, \Omega), (double, 0, 1, \Omega), \ldots \\ & (double, 1, \bot, \Omega), (double, 1, 0, \Omega), (double, 1, 1, \Omega), \ldots \\ & \ldots \\ & (inc, 0, \bot, \Omega), (inc, 0, 0, \Omega), (inc, 0, 1, \Omega), \ldots \\ & (inc, 1, \bot, \Omega), (inc, 1, 0, \Omega), (inc, 1, 1, \Omega), \ldots \\ & \ldots\} \end{aligned}$$

We can now put all this together, giving us a translation for the state D_2 as follows:

$$\begin{aligned}
&[\![D_2, \mathit{Doubler}]\!]\\
&\quad := \mathit{double_0};\ \mathit{double_\bot};\ [\![\Omega]\!]\ []\ \mathit{double_0};\ \mathit{double_0};\ [\![\Omega]\!]\\
&\qquad []\ \mathit{double_0};\ \mathit{double_1};\ [\![\Omega]\!]\ []\ \ldots\\
&\qquad []\ \mathit{double_1};\ \mathit{double_\bot};\ [\![\Omega]\!]\ []\ \mathit{double_1};\ \mathit{double_0};\ [\![\Omega]\!]\\
&\qquad []\ \mathit{double_1};\ \mathit{double_1};\ [\![\Omega]\!]\ []\ \ldots\\
&\qquad \ldots\\
&\qquad []\ \mathit{inc_0};\ \mathit{inc_\bot};\ [\![\Omega]\!]\ []\ \mathit{inc_0};\ \mathit{inc_0};\ [\![\Omega]\!]\ []\ \mathit{inc_0};\ \mathit{inc_1};\ [\![\Omega]\!]\ []\ \ldots\\
&\qquad []\ \mathit{inc_1};\ \mathit{inc_\bot};\ [\![\Omega]\!]\ []\ \mathit{inc_1};\ \mathit{inc_0};\ [\![\Omega]\!]\ []\ \mathit{inc_1};\ \mathit{inc_1};\ [\![\Omega]\!]\ []\ \ldots\\
&\qquad \ldots
\end{aligned}$$

Now, let us translate state Ω. To do this, we will have to evaluate f with $\mathit{before} = \Omega$. As we can see, the value of f in this case does not depend on the predicate passed to it:

$$\begin{aligned}
&f(\Omega, \mathit{pred}, s)\\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\})\\
&\qquad\qquad \mid (\mathit{before} \neq \Omega \wedge \mathit{pre}(\mathit{pred}, s, \Omega, \mathit{arg}))\\
&\qquad\qquad\qquad ?(\mathit{result} \neq \bot \wedge \mathit{after} \neq \Omega \wedge \mathit{pred}(\Omega, \mathit{arg}, \mathit{result}, \mathit{after})) : \mathit{after} = \Omega\}\\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\})\\
&\qquad\qquad \mid (\mathit{false} \wedge \mathit{pre}(\mathit{pred}, s, \Omega, \mathit{arg}))? \ldots : \mathit{after} = \Omega\}\\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \mid \mathit{after} = \Omega\}\\
&\quad = \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times \{\Omega\}
\end{aligned}$$

So, we can now find the value of g:

$$\begin{aligned}
&g(\Omega, s, m)\\
&\quad = \{(\mathit{op}, \mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{ops} \times \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\})\\
&\qquad\qquad \mid (\mathit{arg}, \mathit{result}, \mathit{after}) \in f(\Omega, m(\mathit{op}), s)\}\\
&\quad = \mathit{ops} \times \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times \{\Omega\}\\
&\quad = \{(\mathit{double}, 0, \bot, \Omega), (\mathit{double}, 0, 0, \Omega), (\mathit{double}, 0, 1, \Omega), \ldots\\
&\qquad (\mathit{double}, 1, \bot, \Omega), (\mathit{double}, 1, 0, \Omega), (\mathit{double}, 1, 1, \Omega), \ldots\\
&\qquad \ldots\\
&\qquad (\mathit{inc}, 0, \bot, \Omega), (\mathit{inc}, 0, 0, \Omega), (\mathit{inc}, 0, 1, \Omega), \ldots\\
&\qquad (\mathit{inc}, 1, \bot, \Omega), (\mathit{inc}, 1, 0, \Omega), (\mathit{inc}, 1, 1, \Omega), \ldots\\
&\qquad \ldots\}
\end{aligned}$$

This is exactly the same value that we got for g in state D_2. This confirms that the "fake" state Ω is just like any other state where every operation's precondition is *false*. Thus, conveniently, $[\![\Omega, \mathit{Doubler}]\!] = [\![D_2, \mathit{Doubler}]\!]$.

Now, let us determine f, for the two cases of $\mathit{op} = \mathit{double}$ and $\mathit{op} = \mathit{inc}$, in state D_1:

$$\begin{aligned}
&f(D_1, m(\mathit{double}), s)\\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\})\\
&\qquad\qquad \mid (D_1 \neq \Omega \wedge \mathit{pre}(m(\mathit{double}), s, D_1, \mathit{arg}))\\
&\qquad\qquad\qquad ?(\mathit{result} \neq \bot \wedge \mathit{after} \neq \Omega \wedge m(\mathit{double})(D_1, \mathit{arg}, \mathit{result}, \mathit{after}))\\
&\qquad\qquad\qquad : \mathit{after} = \Omega\}\\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\})\\
&\qquad\qquad \mid \mathit{before} = D_1\ ?\ (\mathit{result} = \mathit{arg} \times 2 \wedge \mathit{after} = D_2) : \mathit{after} = \Omega\}
\end{aligned}$$

$$= \{(arg, result, after) : data \times (data \cup \{\bot\}) \times (s \cup \{\Omega\}) \\ \mid result = arg \times 2 \wedge after = D_2\}$$

$$= \{(0, 0, D_2), (1, 2, D_2), (2, 4, D_2), (3, 6, D_2), (4, 8, D_2), \ldots\}$$

$$f(D_1, m(inc), s) \\ = f(D_1, false, s) \\ = \{(arg, result, after) : data \times (data \cup \{\bot\}) \times (s \cup \{\Omega\}) \mid after = \Omega\}$$

$$= \{(0, \bot, \Omega), (0, 0, \Omega), (0, 1, \Omega), (0, 2, \Omega), (0, 3, \Omega), \ldots \\ (1, \bot, \Omega), (1, 0, \Omega), (1, 1, \Omega), (1, 2, \Omega), (1, 3, \Omega), \ldots \\ (2, \bot, \Omega), (2, 0, \Omega), (2, 1, \Omega), (2, 2, \Omega), (2, 3, \Omega), \ldots \\ \ldots\}$$

From this we can calculate g, for $before = D_1$:

$$g(D_1, s, m) \\ = \{(op, arg, result, after) : ops \times data \times (data \cup \{\bot\}) \times (s \cup \{\Omega\}) \\ \mid (arg, result, after) \in f(D_1, m(op), s)\}$$

$$= \{(double, 0, 0, D_2), (double, 1, 2, D_2), (double, 2, 4, D_2), \\ (double, 3, 6, D_2), (double, 4, 8, D_2), \ldots \\ (inc, 0, \bot, \Omega), (inc, 0, 0, \Omega), (inc, 0, 1, \Omega), (inc, 0, 2, \Omega), (inc, 0, 3, \Omega), \ldots \\ (inc, 1, \bot, \Omega), (inc, 1, 0, \Omega), (inc, 1, 1, \Omega), (inc, 1, 2, \Omega), (inc, 1, 3, \Omega), \ldots \\ (inc, 2, \bot, \Omega), (inc, 2, 0, \Omega), (inc, 2, 1, \Omega), (inc, 2, 2, \Omega), (inc, 2, 3, \Omega), \ldots \\ \ldots\}$$

Finally, we can substitute this into the definition of $[\![state]\!]$ to obtain the following:

$$[\![D_1, Doubler]\!] \\ := double\_0;\ double\_0;\ [\![D_2]\!]\ []\ double\_1;\ double\_2;\ [\![D_2]\!] \\ []\ double\_2;\ double\_4;\ [\![D_2]\!]\ []\ \ldots \\ []\ inc\_0;\ inc\_\bot;\ [\![\Omega]\!]\ []\ inc\_0;\ inc\_0;\ [\![\Omega]\!]\ []\ inc\_0;\ inc\_1;\ [\![\Omega]\!]\ []\ \ldots \\ []\ inc\_1;\ inc\_\bot;\ [\![\Omega]\!]\ []\ inc\_1;\ inc\_0;\ [\![\Omega]\!]\ []\ inc\_1;\ inc\_1;\ [\![\Omega]\!]\ []\ \ldots \\ []\ \ldots$$

Putting it all together, we can draw the labelled transition system for $[\![Doubler]\!]$, which is shown in figure 51.

Now, we will examine a second specification, *Incrementer*, which is a subtype of *Doubler*.

Incrementer :

State space: $\{I_1, I_2\}$
Initial states: I_1
Operation double ($\mathbb{N} \to \mathbb{N}$): $before = I_1 \wedge result = arg \times 2 \wedge after = I_2$
Operation inc ($\mathbb{N} \to \mathbb{N}$): $before = I_1 \wedge result = arg + 1 \wedge after = I_2$

As before, $ops = \{double, inc\}$ and $data = \mathbb{N}$. This is similar to the definition of *Doubler*, but with the *inc* operation added. This makes use of the "weakening preconditions" part of the subtyping relation to relax *inc*'s precondition in *Doubler* (*false*) to its precondition in *Incrementer* ($before = I_1$). The abstraction relation by virtue of which *Incrementer* is a subtype of *Doubler* is in this case quite simple:

$$Abs = \{(D_1, I_1), (D_2, I_2)\}$$

Since *inc*'s precondition has been relaxed, but *double*'s predicate is the same in each specification, *Incrementer* is a subtype of *Doubler*.

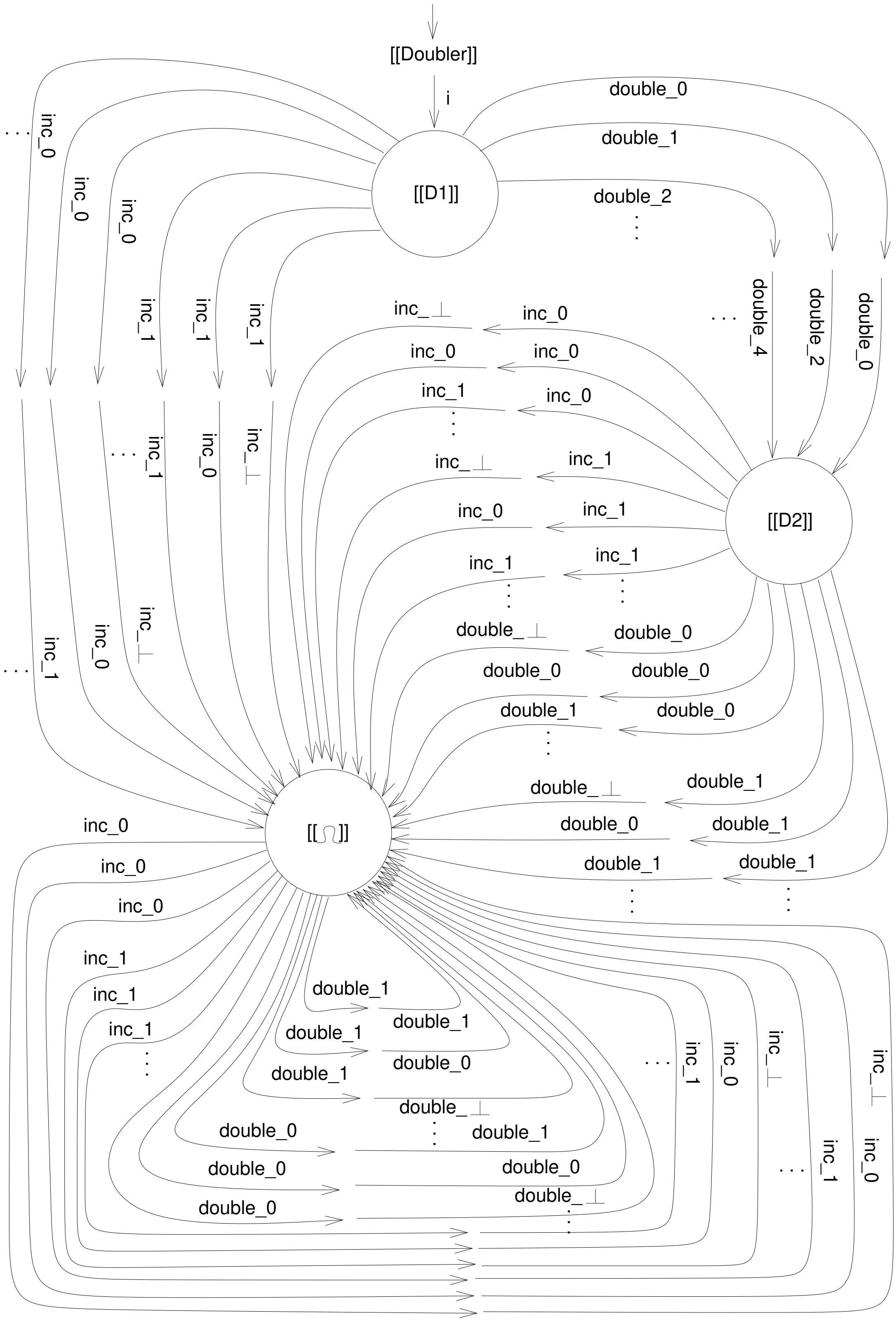

Figure 51: LOTOS translation of specification *Doubler*

The first stage of the translation is trivial:

$$\begin{aligned}
[\![\mathit{Incrementer}]\!] &:= \mathit{choice}\ e : \{I_1\}\ [\!]\ i;\ [\![e,(s,t,m)]\!] \\
&:= i;\ [\![I_1, \mathit{Incrementer}]\!]
\end{aligned}$$

Since *ops* and *data* are the same as before (a necessity if we are to meaningfully compare the two translations), the definition of $[\![\Omega]\!]$ which we obtained for *Doubler* still applies for *Incrementer*. For the same reasons as $[\![D_2]\!] = [\![\Omega]\!]$—namely that both *double* and *inc*'s preconditions are *false* in both states D_2 and I_2—it is also the case that $[\![I_2]\!] = [\![\Omega]\!]$. This leaves only $[\![I_1]\!]$ to be determined.

We will now translate state I_1. We first determine the operations' preconditions in state I_1:

$$\begin{aligned}
&\mathit{pre}(m(\mathit{double}), s, I_1, \mathit{arg}) \\
&\quad = \exists\, \mathit{result} : \mathit{data}, \mathit{after} : s \bullet m(\mathit{double})(I_1, \mathit{arg}, \mathit{result}, \mathit{after}) \\
&\quad = \exists\, \mathit{result} : \mathit{data}, \mathit{after} : s \bullet I_1 = I_1 \wedge \mathit{result} = \mathit{arg} \times 2 \wedge \mathit{after} = I_2 \\
&\quad = \mathit{true}
\end{aligned}$$

$$\begin{aligned}
&\mathit{pre}(m(\mathit{inc}), s, I_1, \mathit{arg}) \\
&\quad = \exists\, \mathit{result} : \mathit{data}, \mathit{after} : s \bullet m(\mathit{inc})(I_1, \mathit{arg}, \mathit{result}, \mathit{after}) \\
&\quad = \exists\, \mathit{result} : \mathit{data}, \mathit{after} : s \bullet I_1 = I_1 \wedge \mathit{result} = \mathit{arg} + 1 \wedge \mathit{after} = I_2 \\
&\quad = \mathit{true}
\end{aligned}$$

We can now determine f for each operation in state I_1:

$$\begin{aligned}
&f(I_1, m(\mathit{double}), s) \\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \\
&\qquad\qquad \mid (I_1 \neq \Omega \wedge \mathit{pre}(m(\mathit{double}), s, I_1, \mathit{arg})) \\
&\qquad\qquad\qquad ?(\mathit{result} \neq \bot \wedge \mathit{after} \neq \Omega \wedge m(\mathit{double})(I_1, \mathit{arg}, \mathit{result}, \mathit{after})) : \mathit{after} = \Omega\} \\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \\
&\qquad\qquad \mid \mathit{result} \neq \bot \wedge \mathit{after} \neq \Omega \wedge m(\mathit{double})(I_1, \mathit{arg}, \mathit{result}, \mathit{after})\} \\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times \mathit{data} \times s \mid \mathit{result} = \mathit{arg} \times 2 \wedge \mathit{after} = I_2\} \\
&\quad = \{(0, 0, I_2), (1, 2, I_2), (2, 4, I_2), (3, 6, I_2), \ldots\}
\end{aligned}$$

$$\begin{aligned}
&f(I_1, m(\mathit{inc}), s) \\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \\
&\qquad\qquad \mid (I_1 \neq \Omega \wedge m(\mathit{inc})(\mathit{pred}, s, I_1, \mathit{arg})) \\
&\qquad\qquad\qquad ?(\mathit{result} \neq \bot \wedge \mathit{after} \neq \Omega \wedge m(\mathit{inc})(I_1, \mathit{arg}, \mathit{result}, \mathit{after})) : \mathit{after} = \Omega\} \\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \\
&\qquad\qquad \mid \mathit{result} \neq \bot \wedge \mathit{after} \neq \Omega \wedge m(\mathit{inc})(I_1, \mathit{arg}, \mathit{result}, \mathit{after})\} \\
&\quad = \{(\mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{data} \times \mathit{data} \times s \mid \mathit{result} = \mathit{arg} + 1 \wedge \mathit{after} = I_2\} \\
&\quad = \{(0, 1, I_2), (1, 2, I_2), (2, 3, I_2), (3, 4, I_2), \ldots\}
\end{aligned}$$

We can now determine g for state I_1:

$$\begin{aligned}
&g(I_1, s, m) \\
&\quad = \{(\mathit{op}, \mathit{arg}, \mathit{result}, \mathit{after}) : \mathit{ops} \times \mathit{data} \times (\mathit{data} \cup \{\bot\}) \times (s \cup \{\Omega\}) \\
&\qquad\qquad \mid (\mathit{arg}, \mathit{result}, \mathit{after}) \in f(I_1, m(\mathit{op}), s)\} \\
&\quad = \{(\mathit{double}, 0, 0, I_2), (\mathit{double}, 1, 2, I_2), (\mathit{double}, 2, 4, I_2), (\mathit{double}, 3, 6, I_2), \ldots \\
&\qquad\ (\mathit{inc}, 0, 1, I_2), (\mathit{inc}, 1, 2, I_2), (\mathit{inc}, 2, 3, I_2), (\mathit{inc}, 3, 4, I_2), \ldots\}
\end{aligned}$$

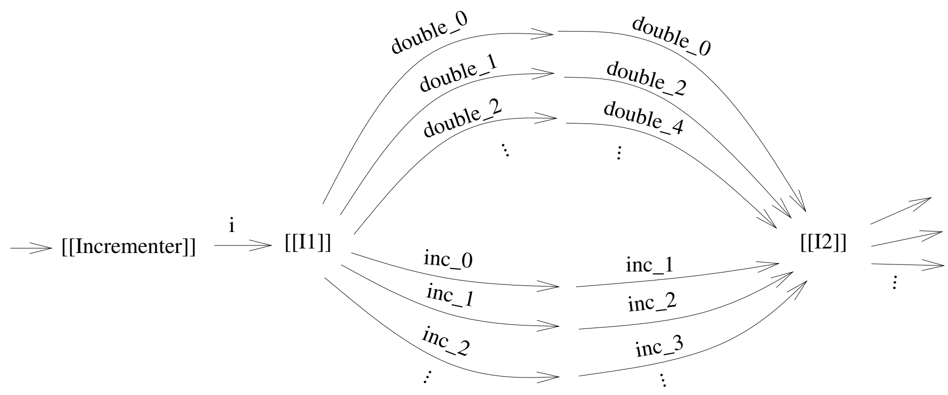

Figure 52: LOTOS translation of *Incrementer*

And from this, we obtain the following behaviour for $[\![I_1]\!]$:

$$
\begin{aligned}
&[\![I_1, \mathit{Incrementer}]\!]\\
&\quad := \mathit{choice}\ (\mathit{op}, \mathit{arg}, \mathit{result}, \mathit{after}) : g(I_1, s, m)\\
&\qquad\quad [\,]\ \mathit{op\_arg};\ \mathit{op\_result};\ [\![\mathit{after}, \mathit{Incrementer}]\!]\\
&\quad := \mathit{double\_}0;\ \mathit{double\_}0;\ [\![I_2, \mathit{Incrementer}]\!]\ [\,]\ \mathit{double\_}1;\ \mathit{double\_}2;\ [\![I_2, \mathit{Incrementer}]\!]\\
&\qquad\quad [\,]\ \mathit{double\_}2;\ \mathit{double\_}4;\ [\![I_2, \mathit{Incrementer}]\!]\ [\,]\ \ldots\\
&\qquad [\,]\ \mathit{inc\_}0;\ \mathit{inc\_}1;\ [\![I_2, \mathit{Incrementer}]\!]\ [\,]\ \mathit{inc\_}1;\ \mathit{inc\_}2;\ [\![I_2, \mathit{Incrementer}]\!]\\
&\qquad\quad [\,]\ \mathit{inc\_}2;\ \mathit{inc\_}3;\ [\![I_2, \mathit{Incrementer}]\!]\ [\,]\ \ldots
\end{aligned}
$$

We can now draw the labelled transition system corresponding to the process $[\![\mathit{Incrementer}]\!]$. The labelled transition system of $[\![I_2]\!]$ is the same as that for $[\![D_2]\!]$, so we will not repeat it here. The labelled transition system is shown in figure 52. We have seen that $[\![D_2, \mathit{Doubler}]\!]$, $[\![I_2, \mathit{Incrementer}]\!]$, $[\![\Omega, \mathit{Doubler}]\!]$ and $[\![\Omega, \mathit{Incrementer}]\!]$ are all equal. With that knowledge, we can now easily see that $[\![\mathit{Incrementer}]\!]\ \mathrm{red}\ [\![\mathit{Doubler}]\!]$, since $[\![I_1]\!]$ can be obtained from $[\![D_1]\!]$ by simply pruning away some of the branches offering *inc_n* actions.

Let us now consider one more specification:

Iterator :

State space: $\{T_1\}$
Initial states: T_1
Operation double ($\mathbb{N} \to \mathbb{N}$): $\mathit{result} = \mathit{arg} \times 2$
Operation inc ($\mathbb{N} \to \mathbb{N}$): $\mathit{result} = \mathit{arg} + 1$

We can use the following abstraction relation to show that *Iterator* is a subtype of *Incrementer*:

$$\mathit{Abs} = \{(I_1, T_1), (I_2, T_1)\}$$

Both *double* and *inc*'s preconditions in state I_2 are false, but in state T_1, they are not. Because of the "weakening preconditions" rule of state-based subtyping, states I_2 and T_1 can be related by the abstraction relation.

The preconditions of *double* and *inc* in state T_1 are exactly the same as their preconditions in state I_1. The return values from these operations are identical. However, if either operation is invoked in state I_1, the after-state will be I_2; with before-state T_1, the after-state will also be T_1. The second rule of state-based subtyping required that the after-states be related by the abstraction relation, which they are.

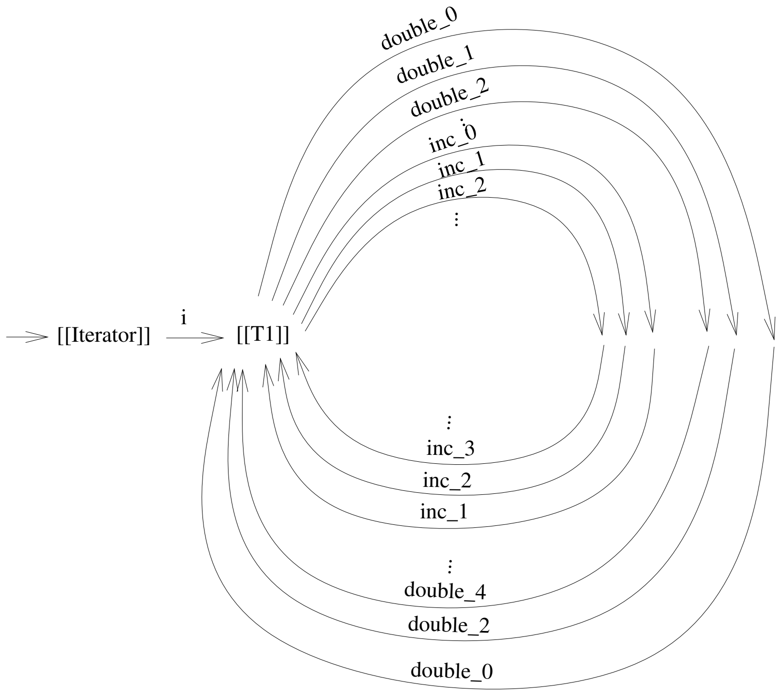

Figure 53: LOTOS translation of *Iterator*

In addition, each of *Iterator* and *Incrementer* has only a single initial state, and these are related by the abstraction relation. We can therefore conclude that *Iterator* is a subtype of *Incrementer*.

Let us now translate *Iterator* into LOTOS. Firstly, we have the following:

$$[\![\mathit{Iterator}]\!] := i;\ [\![T_1, \mathit{Iterator}]\!]$$

Then, we obtain the following values for f:

$$f(T_1, m(\mathit{double}), s) = \{(0,0,T_1),(1,2,T_1),(2,4,T_1),(3,6,T_1),\ldots\}$$
$$f(T_1, m(\mathit{inc}), s) = \{(0,1,T_1),(1,2,T_1),(2,3,T_1),(3,4,T_1),\ldots\}$$

Then g is as follows:

$$\begin{aligned} g(T_1,s,m) = \{&(\mathit{double},0,0,T_1),(\mathit{double},1,2,T_1),(\mathit{double},2,4,T_1),(\mathit{double},3,6,T_1),\ldots \\ &(\mathit{inc},0,1,T_1),(\mathit{inc},1,2,T_1),(\mathit{inc},2,3,T_1),(\mathit{inc},3,4,T_1),\ldots\} \end{aligned}$$

And so we obtain the following behaviour for $[\![T_1]\!]$:

$$\begin{aligned} &[\![T_1, \mathit{Iterator}]\!] \\ &\quad := \mathit{double\_0};\ \mathit{double\_0};\ [\![T_1, \mathit{Iterator}]\!]\ []\ \mathit{double\_1};\ \mathit{double\_2};\ [\![T_1, \mathit{Iterator}]\!] \\ &\qquad []\ \mathit{double\_2};\ \mathit{double\_4};\ [\![T_1, \mathit{Iterator}]\!]\ []\ \mathit{double\_3};\ \mathit{double\_6};\ [\![T_1, \mathit{Iterator}]\!]\ []\ \ldots \\ &\qquad []\ \mathit{inc\_0};\ \mathit{inc\_1}, [\![T_1, \mathit{Iterator}]\!]\ []\ \mathit{inc\_1}, \mathit{inc\_2}, [\![T_1, \mathit{Iterator}]\!] \\ &\qquad []\ \mathit{inc\_2}, \mathit{inc\_3}, [\![T_1, \mathit{Iterator}]\!]\ []\ \mathit{inc\_3}, \mathit{inc\_4}, [\![T_1, \mathit{Iterator}]\!]\ []\ \ldots \end{aligned}$$

The labelled transition system for $[\![\mathit{Iterator}]\!]$ is shown in figure 53.

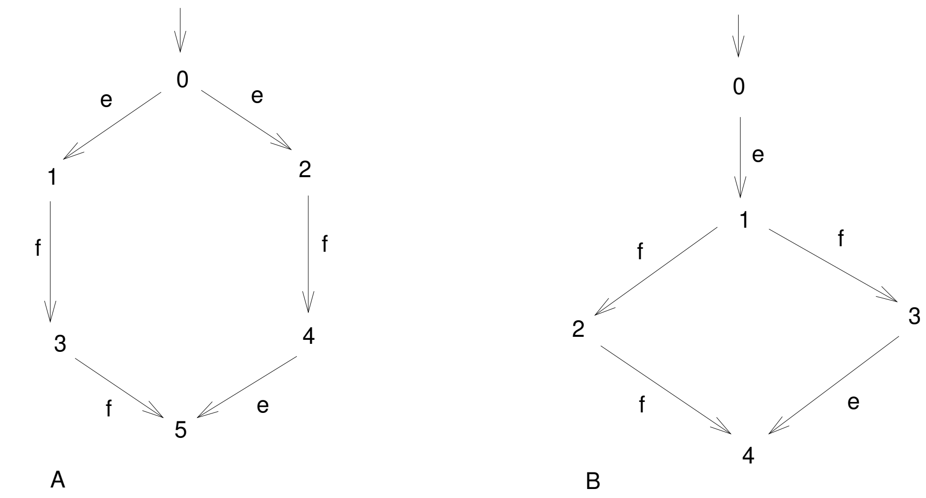

Figure 54: A and B visualised as a state machines

Because any translated state is a reduction of $[\![\Omega]\!]$ (see §7.3.1, §7.3.3), and $[\![I_2]\!]$ is the same behaviour as $[\![\Omega]\!]$, we know that $[\![T_1]\!]$ red $[\![I_2]\!]$. Given this, we can easily see that $[\![T_1]\!]$ red $[\![I_1]\!]$; the transitions leading from these two states are the same, and lead to, respectively, $[\![T_1]\!]$ and $[\![I_2]\!]$, which we already know are related by *red*.

We have now taken three state-based specifications, *Doubler*, *Incrementer* and *Iterator*, and translated them into LOTOS. We have seen that *Iterator* is a subtype of *Incrementer*, and that *Incrementer* is a subtype of *Doubler*. We also saw that $[\![Iterator]\!]$ red $[\![Incrementer]\!]$ and $[\![Incrementer]\!]$ red $[\![Doubler]\!]$. These results are consistent with the theorem proved earlier in the chapter:

$$CSpec \sqsubseteq ASpec \Rightarrow [\![CSpec]\!] \text{ red } [\![ASpec]\!]$$

7.9 Proving that reduction does not imply subtyping

We have now proved that subtyping implies reduction, or, more accurately, that subtyping between two state-based specifications implies reduction between their process algebraic translations. We now want to prove the reverse: that if reduction holds between two translated specifications subtyping must necessarily hold between the untranslated specifications.

Unfortunately, we cannot do this; it is not the case. In this section we present a counterexample showing that reduction does not imply subtyping.

Consider the following two specifications.

A :

State space: $\{S_0, S_1, S_2, S_3, S_4, S_5\}$
Initial states: S_0
Operation e ($nil \rightarrow nil$): $(before, after) \in \{(S_0, S_1), (S_0, S_2), (S_4, S_5)\}$
Operation f ($nil \rightarrow nil$): $(before, after) \in \{(S_1, S_3), (S_2, S_4), (S_3, S_5)\}$

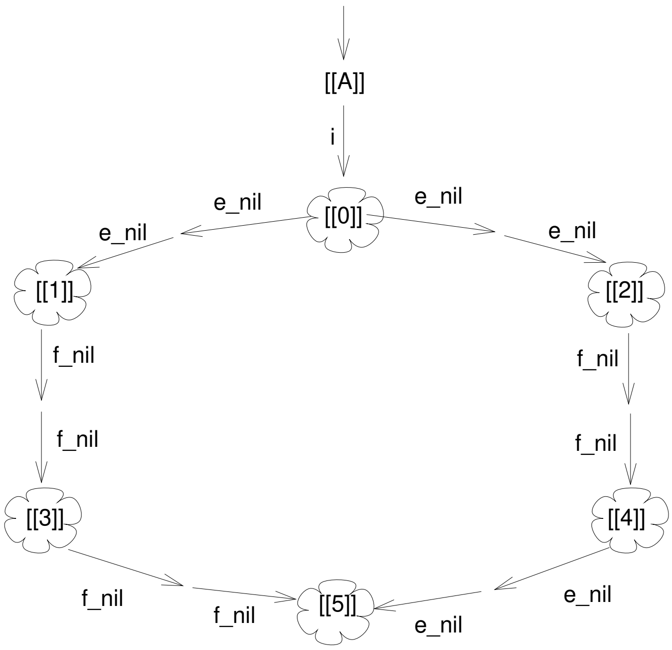

Figure 55: The labelled transition system for ⟦A⟧

B :

State space: $\{S_0, S_1, S_2, S_3, S_4\}$
Initial states: S_0
Operation e $(nil \to nil)$: $(before, after) \in \{(S_0, S_1), (S_3, S_4)\}$
Operation f $(nil \to nil)$: $(before, after) \in \{(S_1, S_2), (S_1, S_3), (S_2, S_4)\}$

These specifications can be visualised as transition systems as shown in figure 54. We can also translate these specifications into LOTOS, and display the LOTOS specifications as labelled transition systems, as shown in figures 55 and 56.

In these two figures, we have used the "cloud" notation we first used in §4.5.1 to indicate that the labelled transition system contains many transitions leading from this state which are not shown in the diagram. Specifically, the cloud notation denotes a transition from each of these "cloudy" states for each combination of operation name *op*, argument value *arg* and result value *result*, where the argument value is drawn from *data*, the result value is drawn from $data \cup \{\perp\}$, and the state cannot otherwise perform the trace $\langle op\_arg, op\_result\rangle$. The transition is labelled *op_arg* and leads to a state representing the LOTOS behaviour *op_result*; $[\![\Omega]\!]$.

We can easily see that the labelled transition systems in figures 55 and 56 are testing equivalent; they both have the same set of traces and the same set of refusals after each of those traces. The difference between A and B is that, in B, a single state (⟦1⟧) appears where, in A, there are two states (⟦1⟧ and ⟦2⟧). Although these two states in A do not have the same traces or refusals, the union of the traces and the union of the refusals of those states are equal to the traces and refusals of the corresponding state in B. From this observation, we can see that ⟦A⟧ and ⟦B⟧ are testing equivalent.

However, although the LOTOS translations of A and B are testing equivalent (and therefore reduce one another), and A is a subtype of B, B is not a subtype of A. We can prove this by attempting to construct an abstraction relation by virtue of which B would be a subtype of A.

We will need to refer to the conditions set out in §3.2.2 for Z refinement. The conditions are

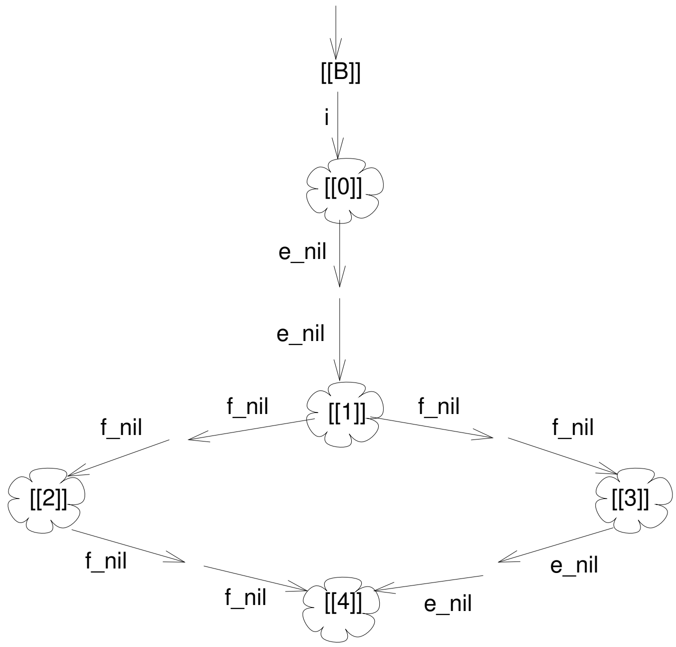

Figure 56: The labelled transition system for $[\![\mathrm{B}]\!]$

as follows. The first is the "preconditions" rule:

$$\forall\, Astate, Cstate, arg \bullet Abs \wedge pre\ AOp \Rightarrow pre\ COp$$

The second is the "postconditions" rule:

$$\forall\, Astate, Cstate, Cstate', arg, result \bullet \\ Abs \wedge pre\ AOp \wedge pred\ COp \Rightarrow (\exists\, Astate' \bullet Abs' \wedge pred\ AOp)$$

The third is the "initial states" rule:

$$\forall\, Cstate \bullet Cinit \Rightarrow (\exists\, Astate \bullet Ainit \wedge Abs)$$

Taking $ASpec = A$ and $CSpec = B$, we assume that there is an abstraction relation Abs which satisfies all three of the rules given above. We will now deduce some constraints which must apply to any such abstraction relation, and show that there is no abstraction relation which satisfies the constraints.

Consider the initial state of B. Given that state 0 in B is an initial state, the third rule ensures that any initial state of B must be linked by the abstraction relation to an initial state of A. Since A's only initial state is state 0, the arc labelled A in figure 57 must be present in the abstraction relation.

Now that we have established that transition A must be present in the abstraction relation, we can show that other transitions must also be present. Consider the second rule. If we consider the case where $AState = 0$, $CState = 0$, $AOp = COp = e$ and $CState' = 1$, the second rule comes into effect and implies that at least one of the arcs B and C must be present in the abstraction relation. So any abstraction relation by means of which A is a subtype of B must include at least arc A and one of the arcs B and C.

Now, if arc B is present in the abstraction relation, and we let $AState = 1$, $CState = 1$, $AOp = COp = f$ and $CState' = 3$, the second rule implies that arc D must be present. However,

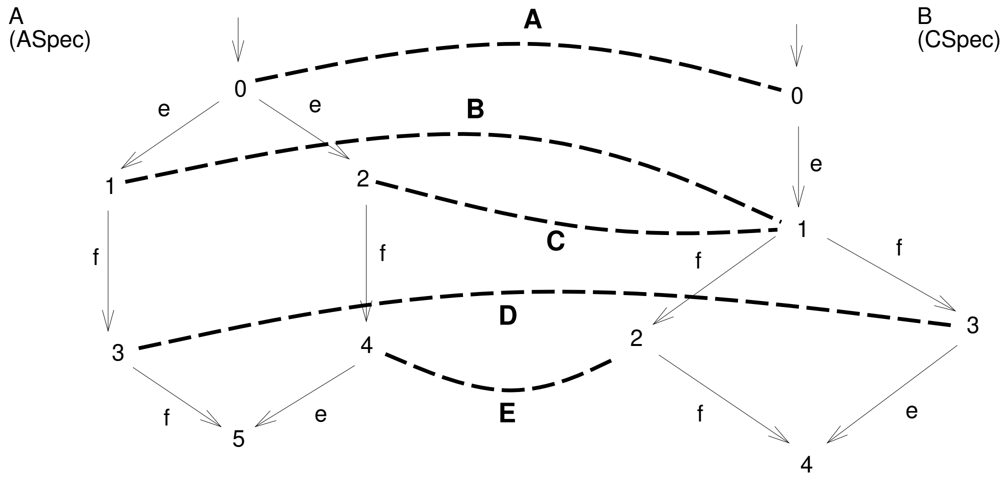

Figure 57: Possible relational associations between states in A and B

we can invoke the first rule to show that arc D cannot be included in the abstraction relation. If $AState = 3$, $CState = 3$, and transition D is present, the first rule implies that state 3 in B must be able to perform the action f, which it cannot. Thus, arc D cannot be present. It follows that arc B cannot be present either. The only remaining option is that C must be present.

However, if C is present, we can show that arc E must also be present. In the same way that we showed that arc D is not consistent with the first rule, we can show that arc E is also not consistent with that rule. Therefore, arc E cannot be present, and it follows that arc C cannot be present, either.

If neither of arcs B or C can be present, neither can arc A. Finally, if arc A cannot be present in the abstraction relation, this means that the third rule cannot be satisfied and that we do not have an abstraction relation which shows that B is a subtype of A.

Because our reasoning applies to any relation which could be an abstraction relation appropriate for showing that B is a subtype of A, we have now proved that there is no such abstraction relation and that B is therefore not a subtype of A.

Taking together the observations that $[\![B]\!]$ is a reduction of $[\![A]\!]$ and that B is not a subtype of A, we can see that reduction cannot correspond to subtyping. In fact, since $[\![B]\!]$ is testing equivalent to $[\![A]\!]$, we can see that no LOTOS relation based solely on trace and refusal semantics can correspond to subtyping. We need a relation based on a semantics with a stronger notion of observation.

The strong and weak bisimulation relations are both stronger than reduction, but, as we saw in §4.3, they are both equivalences, and are thus not suitable for use as a subtyping relation. Nevertheless, [BD99] suggests that subtyping has the character of bisimulation. There are other relations, such as 1/3-bisimulation and 2/3-bisimulation which also have something of the nature of (bi)simulation, but which are preorders and not equivalences. Further investigation would show whether one of these relations corresponds to subtyping.

7.10 Conclusion

In this chapter, we have proved that the LOTOS reduction relation is implied by the Z refinement relation, when using the translation given in the previous chapter. Since the Z refinement relation is the interesting subset of Liskov and Wing's behavioural subtyping relation[LW94], and is equivalent to the downward simulation relation, this evidence tends to confirm our intuitive notion that reduction is an appropriate instantiation of the subtyping relation in the process algebraic setting. However, we have also proved, by counterexample, that it is not the case that $[\![CSpec]\!]\ red\ [\![ASpec]\!] \Rightarrow CSpec \sqsubseteq ASpec$. This proves that reduction does not correspond to subtyping, but that any relation which does must be based on a stronger semantics than reduction's trace and refusal semantics.

Chapter 8

Conclusion

In this chapter, we will summarise the results obtained in this thesis, and attempt to draw some conclusions from them. We also make some suggestions for further work.

8.1 Summary

In this thesis, we have been trying to determine which process algebraic relation corresponds to the state-based subtyping relation. We took the Z refinement relation to be subtyping, both because it is very similar to the well-known downward simulation relation, and because its conditions are the same as the core conditions of Liskov and Wing's "behavioural subtyping" relation.

To compare state-based relations with process algebraic relations, we use a translation. We established the following criterion by which we will judge whether a process algebraic relation corresponds to a state-based relation: if the process algebraic relation holds between the translations of any two state-based specifications exactly when the state-based relation holds between the same two specifications, then we consider the two relations to correspond to one another.

In §4.3, we saw that although the LOTOS extension relation appears, at first, to have the properties we would expect from a subtyping relation, we later saw, in §4.4, that undefined behaviour is treated quite differently in each of the two specification paradigms we are concerned with. In state-based specifications, undefined behaviour is treated as being unspecified: the object may behave in any way if an undefined operation is invoked. In process algebraic specifications, undefined behaviour is effectively specified not to be possible: a process will actively refuse any action which it is not specified to be able to accept. We addressed this difference by constructing a behaviour, Ω, which reproduced "unspecified" behaviour in LOTOS.

In §5.1.1, we discussed a discrepancy between the behaviour of a state-based specification when passing its return value back to its caller and the LOTOS behaviour to which we were translating it. In particular, while a state-based specification could choose the value it was to return nondeterministically—that is, when an operation could internally choose one of several values to return—the LOTOS behaviour to which we were translating it could have its choice of return value influenced by its environment.

We considered two modified languages based on LOTOS, each of which contained a new feature designed to facilitate a particular kind of nondeterminism. Our first modified language changed the semantics of the LOTOS output operator such that a choice between the outputting of two or more values on the same gate would be made nondeterministically. Our second LOTOS variant introduced a "nondeterministic choice" operator, $\sqcap$, whose inference rules were capable of "pushing it through" ordinary LOTOS choice constructs. Although we did not use either of these modified LOTOS-like languages in our translation, they provided the inspiration needed to find an appropriate method of producing the needed nondeterminism in result values.

In §6.6, we collected together our solutions to each of the inadequacies of our original simple translation, and produced a definition of a more sophisticated translation. Where the original state-based translation does not define behaviour, our translation produces LOTOS behaviour which is less reduced than any other behaviour which the translation can produce. The argument and result are represented in two separate actions, and the second action is chosen nondeterministically. This translation appeared to correctly capture the behaviour of a state-based specification and represent that behaviour as a set of LOTOS process definitions.

Finally, in chapter 7, we prove that, if a state-based specification CSpec is a subtype of a second state-based specification ASpec, (according to our state-based subtyping relation, whose conditions are those of Z refinement) then the translated process $[\![CSpec]\!]$ is necessarily a reduction of the translated process $[\![ASpec]\!]$. This does not prove that reduction corresponds to state-based subtyping; to obtain that result, we would also have to prove that the presence of reduction between two translated processes implies the presence of subtyping between the specifications from which those processes were translated. At the end of chapter 7, we prove by means of a counterexample that this is not the case; there exists at least one pair of state-based specifications where the LOTOS translation of the first specification is a reduction of the translation of the second, but where the first specification is not itself a subtype of the second.

8.2 Contributions of this thesis

The work described in this thesis makes a number of contributions:

- We developed a method of translating state-based specifications into LOTOS which takes into account the differences in interpretation of undefined behaviour between the state-based and process algebraic settings. Another translation, translating from LOTOS to Z, has been developed in [DBBS96] as part of an effort to enable specifications in multiple notations from multiple RM-ODP viewpoints to be checked for consistency. Our translation could be useful in the process of unifying specifications outlined in [BBDS96].

- While other works such as [Rud92] have taken the extension relation to be subtyping, we have argued that there are problems with the use of extension, namely that an extended process may fail to refuse an action which a less extended process must refuse. If refusals are taken to be part of the specified behaviour of a process, we do not believe that extension makes a good subtyping relation. (However, if a specification is checked before-hand to ensure that no attempt is made to invoke undefined behaviour, then only the operations defined in a supertype could ever be invoked, and extension could be used as a subtyping relation.)

- We presented arguments informally justifying the use of reduction as a subtyping relation in the process algebraic setting. Since reduction requires that all the behaviour, including both traces and refusals, of the supertype must appear in any subtype, reduction appears to naturally behave as a subtyping relation for process algebras.

- We attempted to prove that reduction corresponds to subtyping, which we took to be the Z refinement relation. We have proved that, using our translation, state-based subtyping does imply reduction, but we have also proved by counterexample that reduction does not imply subtyping. Our counterexample showed that it is possible for a subtle change in nondeterministic choice can cause two specifications to be distinguishable by subtyping while being identified by testing (and thus indistinguishable using reduction). We can conclude from this result that any LOTOS relation corresponding to state-based subtyping must be based on a stronger observational model than reduction; that is, it must make use of more than just traces and refusals. From [vG90], which gives an overview of which semantics are stronger than which, we can see that several semantics are located between trace-refusals semantics (which van Glabbeek calls "failure semantics") and bisimulation semantics. The 2/3-bisimulation relation, which uses ready simulation semantics, seems a good possibility for a process algebraic subtyping relation.

8.3 Limitations of our work

The work described in this thesis has a number of shortcomings. These include the following:

- Many state-based specifications have infinite state spaces. Since our translation produces a separate LOTOS definition for each state in the input state-based specification, as described in §6.6.1 the translation of a specification with an infinite state space would itself have to consist of an infinite number of definitions, and would therefore not be valid LOTOS. However, the translation is still valid for state-based specifications which have a finite state space; in these cases, the translated specification contains a finite number of definitions.

- The range of data values available to be used for argument and return values for operations is often infinite: for example, numbers are often passed as arguments. This introduces a problem similar to that for infinite state spaces. LOTOS *can* be used to describe a choice between an infinite number of behaviours. For example, "*choice* $x : \mathbb{N}$ [] $a!x$; B" describes a choice between an infinite number of behaviours equivalent to "a_0; $B[x/0]$", "a_1; $B[x/1]$", "a_2; $B[x/2]$", etc. Our translation, however, selects a state (from the state-based specification) from a set, and later evolves to the translation of that state. As described in §6.6.1, we do not currently have a method of making this valid LOTOS. While we can produce valid LOTOS translations in cases where the set of after-states of an operation is limited, by expanding the LOTOS *choice* construct into a series of alternatives separated by the LOTOS "[]" operator, we cannot do this for an infinite choice of after-state.

  As in the previous item, a state-based specification with a finite state space can still be translated into a valid LOTOS specification: if the *choice* constructs are expanded such that the choice of after-state is not selected from a set, the resulting specification is valid LOTOS. In [DBBS96], a translation from LOTOS to Z is developed, and, when an infinite LOTOS behaviour is being translated, a method of describing the infinite state space finitely is used. This technique could be adapted to our translation.

- We have considered only LOTOS as the target process algebra for our translation, and have only considered the trace-based semantics used by trace preorder and trace equivalence, the traces–refusals semantics used by reduction, extension and testing equivalence, and the strong and weak bisimulation relations. Other important relations we could have looked at include 1/3- and 2/3-bisimulation.

- The LOTOS processes produced by our translation cannot subsequently be composed together meaningfully. If two state-based specifications are translated using our translation, the resulting LOTOS processes cannot, for example, be composed in parallel with the expected results. The extra behaviour added to the process in order to reproduce the unspecified behaviour of the state-based specification would interfere with the composition.

8.4 Further work

There are several directions this work could lead; a few of them are listed here.

- Other LOTOS development relations should be examined to see whether they could serve as a process algebraic behavioural subtyping relation. The 2/3-bisimulation relation, in particular, may be a good candidate.

- It might be possible to overcome the limitations identified above. In particular, the technique used in [DBBS96] to handle infinite behaviour might be adapted to our translation.

- We have taken downward simulation to be the state-based refinement relation. In fact, Liskov and Wing's subtyping relation[LW94] has one significant clause (actually, a choice of two clauses) which we decided to ignore after we examined the relation in §2.3.3. It might be possible to find a process algebraic relation corresponding to the entire Liskov and Wing subtyping relation, including the constraint rule or the extension map.

- The translation developed in this dissertation targets an untimed process algebra with atomic actions. Alternative models such as actions with duration or true concurrency semantics should be investigated as to whether they can support a more natural translation.

- We have investigated only the traces and refusals semantics for our process algebraic subtyping relation. There are many other possibilities worth investigating[vG90].

- The definition of Ω itself might be useful outside the context of this work. Some definitions for Ω are given in §4.4.3, but other definitions might be appropriate. In some semantic models, divergence is significant; this might affect the definition of Ω in such an environment.

- Tools could be built to automate the translation described here, and to check the resulting LOTOS specifications for the presence of reduction. Such tools could find use in multi-paradigm specification environments such as projects based on the RM-ODP framework.

- Is signature-based subtyping a special case of behavioural subtyping, as characterised by Z refinement and/or reduction? A behaviour may contain just a single state which accepts an action for each operation that the object accepts, and which then remains in the same state. This might model signature-based subtyping.

Appendix A

The LOTOS notation and labelled transition systems

This section gives an overview of the LOTOS constructs used in this dissertation, and defines some operators used in defining LOTOS's semantics. In many cases, the information given here is incomplete; only enough is given to cover the LOTOS we use in the thesis. These topics are described in more detail elsewhere, e.g. [BB87].

A LOTOS specification describes the behaviour of a system. The behaviour is usually specified as a set of process definitions. In order not to constrain the possible implementations of the specification, a LOTOS specification describes only the externally visible behaviour of these processes. A process has a number of *gates*: sites at which the process can interact with its environment. These interactions are modelled by *actions*.

Actions are the unit of observable behaviour in the system. They are indivisible in time, and occur instantaneously. As a consequence, two or more actions cannot occur simultaneously.

The actions used in a specification are drawn from the set Act. This set includes all the observable actions which can be used in a behaviour specification. It does not include the distinguished "internal" action named i. Although we often think of Act as being the set of *all possible* observable actions, we can define it, for the purpose of example, to contain fewer actions; in this way, we can make an example more manageable by reducing the size of Act. However, all the actions we use must be drawn from Act, with whatever definition it has at the time. Often, we will not define Act explicitly, but will simply assume that all the actions we use are contained in it.

The set of all observable actions used in a particular specification is denoted $\mathcal{L}$. $\mathcal{L}$ is determined from the behaviour specification, rather than being defined.

A.1 Behaviours

A behaviour expression specifies how a particular process may act. One primitive behaviour is stop, which describes a process which cannot take part in any interactions. This is synonymous with deadlock.

Action prefix is used to specify a process which can do a sequence of actions. Its form is "*action*; *behaviour*". This indicates that the process can do the action *action*, and then must behave according to *behaviour*. This can be used recursively to build up a sequence of actions. We say that, when the action has been performed, the process *evolves to* the given behaviour.

For example,

a; stop

specifies a process which can perform a single action a, then must do nothing more. The

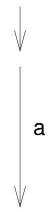

Figure 58: Labelled transition system representing behaviour $a;\ stop$

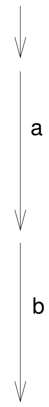

Figure 59: Labelled transition system representing behaviour $a;\ b;\ stop$

behaviour

$a;\ b;$ stop

specifies a process which can perform action a, then action b, and which must then do nothing else.

A.2 Labelled transition systems

LOTOS specifications can be displayed pictorially as labelled transition systems. Labelled transition systems represent the behaviour as a set of states, with the occurrence of actions causing the process to move into a different state. The labelled transition system is shown as a directed graph, with one node representing each possible state of the process, and an arc, labelled with the action, representing the transition from one state to another. Here, we indicate the starting node by drawing a short arrow leading to it from outside the labelled transition system.

For example, the labelled transition system corresponding to the behaviour

$a;$ stop

is shown in figure 58. The labelled transition system for

$a;\ b;$ stop

is shown in figure 59.

A.3 Inference rules

We need a way of formally linking together labelled transition systems with the LOTOS behaviours they represent. We do this by giving inference rules, which define the labelled transition system corresponding to a given LOTOS construct.

The behaviour $a;\ \mathit{stop}$ can perform the action a, and thus evolve to the behaviour stop. We write this symbolically as follows: $a;\ \mathit{stop} \xrightarrow{a} \mathit{stop}$.

An inference rule is written around a horizontal line. The transitions written above the line are conditions of the rule. (We will see examples of these below.) Below the line are written the consequents of the rule. To the right of the line, we can give side-conditions; the rule only applies if the side-conditions are met.

Since $a;\ \mathit{stop}$ can always evolve to stop by performing a, we can write the rule simply as follows:

$$\frac{-}{a;\ \mathit{stop} \xrightarrow{a} \mathit{stop}}$$

There are no conditions on this rule—which we indicate with a dash above the line—so it is always applicable. It says that the process $a;\ \mathit{stop}$ can always evolve to the process stop by performing action a.

We can now write down rules for the LOTOS constructs we have seen so far.

stop cannot evolve at all, so there are no inference rules saying how it may. No LOTOS process may evolve in any way unless there is an inference rule stating how it may do so.

We can write a general rule for action prefix as follows:

$$\frac{-}{a;\ B \xrightarrow{a} B}$$

In this case, we use a as a variable representing "any action", and B as a variable representing "any behaviour". This is customary in inference rules. From now on, we will use small letter variables like a for actions, and capital letter variable such as B for behaviours as we give inference rules for each LOTOS construct we introduce. We will also use e to denote expressions, and P for the process names used in process definitions.

A.4 Action prefix with data

The form of action prefix which we saw above is quite simple: the process interacts on a gate and evolves to some other process. In Basic LOTOS, that is the only kind of interaction that exists. However, we would like to be able to pass data during such interactions. Full LOTOS has an expanded action prefix construct which provides this facility.

We can pass a data value while interacting on a gate by using LOTOS's output notation, like this:

$$a!e;\ B$$

This process interacts on gate a and, when it does so, passes the value of e to any process it synchronises with. It then evolves, as before, to B.

Using this output notation corresponds to writing the following Basic LOTOS, in those cases where e is a ground term:

$$a\_e;\ B$$

Its inference rule reflects this:

$$\frac{-}{a!e;\ B \xrightarrow{a\_[\![e]\!]} B}$$

In this rule, $[\![e]\!]$ denotes the result of evaluating the expression e.

There is another form of action prefix available for receiving a value into a variable. It uses LOTOS's input notation, thus:

$$a?v : T;\ B$$

This process, again, can interact on gate a, but this time it can accept a range of values; any member of the type T. The actual value received is placed in variable v.

This input notation is also equivalent to a Basic LOTOS construct, but this time not necessarily a finite one. For example, $a?v : \mathbb{N};\ B$ is equivalent to the following Basic LOTOS:

$$a\_0;\ B[v/0]\ []\ a\_1;\ B[v/1]\ []\ a\_2;\ B[v/2]\ []\ \cdots$$

The notation $B[x/y]$ refers to a behaviour the same as B, but with every occurrence of the variable name x replaced with the value y.

So, the effect of the input action prefix $a?v : T;\ B$ is to offer for interaction any action of the form a_x where $x \in T$. If and when one of these actions is performed, the process evolves to B, but with the actual value received bound to the variable name v.

The inference rule for this is as follows:

$$\frac{-}{a?v : T;\ B \xrightarrow{a\_n} B[v/n]} \quad [\ n \in T\ ]$$

This rule can be instantiated for any value of n, provided that the condition $n \in T$ is met. The rule then allows the input form of the action prefix to evolve as described above.

A.5 Choice

A behaviour may specify several evolution paths for a process to follow using *choice*. A choice is written as "*behaviour*1 [] *behaviour*2". The process will do either *behaviour1* or *behaviour2*. When a choice must be resolved, the process offers to its environment the first action in each branch of the choice. If the environment then allows one of these actions to occur, the choice may evolve to the corresponding process. Choice is represented in a labelled transition system by a branch-point in the sequence of actions.

For example, the behaviour

$$a;\ \text{stop}\ []\ b;\ \text{stop}$$

has the labelled transition system shown in figure 60.

If the behaviour of its environment is

$$a;\ \text{stop}$$

then the process will choose the first branch of the choice (the arc labelled a in the labelled transition system). If its environment is

$$b;\ \text{stop}$$

then it will choose the second branch (labelled b).

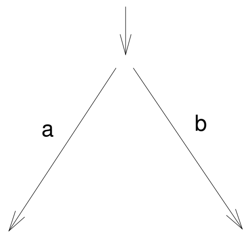

Figure 60: Labelled transition system representing behaviour $a;\ \mathit{stop}\ []\ b;\ \mathit{stop}$

Choice has two inference rules, one for each side of the choice. The first is

$$\frac{B \xrightarrow{a} B'}{B\ []\ C \xrightarrow{a} B'}$$

This rule is the first we have seen with a condition above the line. It says that *if* the behaviour on the left-hand side of the choice can evolve to a second behaviour by performing some action, *then* the whole choice can evolve to that same second behaviour by performing the same action. Notice that performing the action resolves the choice; the process $B\ []\ C$ evolves to B' when it performs action a, *not* to $B'\ []\ C$.

The second inference rule says the same thing as the first rule, but allows the right-hand side to resolve the choice:

$$\frac{C \xrightarrow{a} C'}{B\ []\ C \xrightarrow{a} C'}$$

A.6 Nondeterminism

Consider this process:

$a;\ b;\ \text{stop}\ []\ a;\ c;\ \text{stop}$

In this case, the only action which the process can initially perform is a, so this is the only action which the process will offer to its environment. If and when the environment performs a, the process will take one or other of the branches of the choice. Since both branches begin with a, the environment cannot specify which branch to take. The choice is nondeterministic; when a is performed, the process may evolve to either $b;\ \mathit{stop}$ or $c;\ \mathit{stop}$

A.7 Process definition and instantiation

To define a process identifier P to refer to a behaviour expression B, we write a process definition as follows:

$P := B$

When P is referenced in another behaviour, the effect is as if B were textually substituted for P. It is possible to reference P within the behaviour B, producing a recursive definition. We will not, however, permit the use of "unguarded" recursion. Unguarded recursion occurs when the behaviour B can immediately perform an action which comes from the behaviour of

P, regardless that P is defined to be B. For example,

$$P := a;\ b;\ \mathit{stop}\ []\ c;\ P$$

is allowed, because the right-hand side must perform action c before it can become P.

$$P := i;\ P$$

is also allowed; P must perform an action, in this case an internal action, before it reaches the recursive usage of P. On the other hand,

$$P := a;\ b;\ \mathit{stop}\ []\ P$$

is not allowed, because there is nothing "separating" the use of P from its definition; it is defined *directly* in terms of itself.

Indirect recursion may also be defined, by means of several processes definitions which reference each other.

So far, we have described process instantiation as it exists in Basic LOTOS. In Full LOTOS, we can pass parameters when we instantiate a process definition. For example, we can write:

$$P(x : \mathbb{N}) := a!x;\ (b!x;\ \mathit{stop}\ []\ b!x+1;\ \mathit{stop})$$

This process definition takes one parameter, x. It outputs x on gate a, then outputs either x or $x+1$ on gate b.

Also note that, since $b!x$ and $b!x+1$ map down to actions $b\_[\![x]\!]$ and $b\_[\![x+1]\!]$, the process's environment can influence which of them is actually performed, and thus influence which branch of the choice is taken.

The full inference rule describing how process definitions work is rather involved:

$$\frac{B[\tilde{x}/[\![\tilde{e}]\!]] \xrightarrow{a} B'}{P(\tilde{e}) \xrightarrow{a} B'} \quad [\ P(\tilde{x} : \tilde{T}) := B\ ]$$

In this definition, $\tilde{x}$, $\tilde{e}$ and $\tilde{T}$ are vectors of variables, expressions and types, respectively, and $[\![\tilde{e}]\!]$ denotes the result of evaluating each expression in the vector $\tilde{e}$.

On the right is a side-condition. It says that the rule is only applicable if a process definition for process P exists. In Full LOTOS, process definitions may take parameters; the formal parameters are represented by $\tilde{x}$. The types of these parameters are $\tilde{T}$, but we need not consider them further; we assume that type checking has already been done.

The behaviour we are considering is a reference to P with certain actual parameters $P(\tilde{e})$. $B[\tilde{x}/[\![\tilde{e}]\!]]$ is the process used to define P, but with the actual parameters substituted in place of the formal parameters.

We can now see what the rule says. Given a process definition, *if*, after substituting some actual parameters for the formal ones, the behaviour on the right of the process definition can perform some action and so evolve to a second behaviour, *then* a reference to the process, with the same actual parameters, can perform the same action and thereby evolve to the same second behaviour.

A.8 Hiding

It it useful to use process specifications already written in the definitions of new processes. When this is done, actions used solely within the new process, which are not intended to be used to interact with the environment, are normally *hidden*. This is used, for example, to hide time-out actions within communication protocols. The LOTOS syntax for this is of the form

$$\mathit{hide}\ a_1, \ldots, a_n\ \mathit{in}\ B$$

We will adopt the following abbreviated syntax:

$$B \backslash \{a_1, \ldots, a_n\}$$

The behaviour of this expression is exactly the same as the behaviour of B, except that actions $a_1, \ldots, a_n$ are hidden within the process, and cannot be observed or interacted with by the environment of the process. We model these actions as i, the *internal action*. i cannot be used to interact with the environment, only to indicate in the specification that "something" happens within the process.

For example, the behaviour $a;\ b;\ c;\ stop$ can perform the actions a, b and c in that order, then deadlocks. If we hide b as follows:

$$(a;\ b;\ c;\ stop) \backslash \{b\}$$

we get a process indistinguishable from

$$a;\ i;\ c;\ stop$$

The inference rules for hiding are as follows:

$$\frac{B \xrightarrow{a} B'}{B \backslash G \xrightarrow{i} B' \backslash G} \quad [\ a \in G\ ]$$

$$\frac{B \xrightarrow{a} B'}{B \backslash G \xrightarrow{a} B' \backslash G} \quad [\ a \notin G\ ]$$

In these rules, G is the set of actions being hidden. Any action B can do which is not hidden, $B \backslash G$ can do also. Actions in the set G, however, are transformed into the internal action, i.

We can write internal actions explicitly (as we did in the example above), to avoid the need to write normal actions and then hide them. The inference rule allowing this is:

$$\frac{-}{i;\ B \xrightarrow{i} B}$$

A.9 Internal actions and nondeterminism

We have seen one form of nondeterminism; there are, in fact, two other forms which arise when internal actions appear in a specification.

In a process like this:

$$i;\ a;\ \text{stop}\ []\ i;\ b;\ \text{stop}$$

the decision is made between the two branches of the choice without the process having to interact with its environment at all. The process will offer either a or b to its environment. It is not possible for the environment to know which, or to influence the decision in any way.

The last case arises in processes such as these:

$$i;\ a;\ \text{stop}\ []\ b;\ c;\ \text{stop}$$

$$a;\ b;\ \text{stop}\ []\ i;\ c;\ \text{stop}$$

In these examples, one branch of the choice begins with an internal action, and the other begins with a visible action. When the process starts to resolve the choice, the visible action will be offered to the environment. If the environment is ready to do the visible action that action may be performed and, if so, the choice will take that branch. If not, the internal action can occur, and the other branch of the choice will be taken.

A.10 Parallel evolution

To specify that two processes evolve in parallel, we write:

$$B_1 \; |[a_1, \ldots, a_n]| \; B_2$$

This indicates that B_1 and B_2 can perform any actions independently, except for actions $a_1, \ldots, a_n$, which must occur simultaneously in both processes, if at all.

For example, the process

$$a; \; b; \; \mathit{stop} \; |[b]| \; b; \; c; \; \mathit{stop}$$

may perform a, then b, then c. The action b occurs simultaneously in both branches of the parallel construct.

We write

$$B_1 |||B_2$$

to indicate that B_1 and B_2 evolve independently in parallel, with no shared behaviour and no synchronisation. Writing $P|||Q$ is equivalent to writing $P \; |[\,]| \; Q$.

To compose two processes in parallel with all their actions synchronised (except, of course, internal actions), we use

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

Writing $P \; || \; Q$ is equivalent to writing $P \; |[Act]| \; Q$, and is known as full synchronisation.

We need several inference rules to fully capture the behaviour of parallel evolution. Firstly, we consider internal actions. If internal actions are to be truly internal to the process they occur in, then either side of a parallel composition must be able to perform internal actions at any time, without affecting the other side of the structure.

$$\frac{P \xrightarrow{i} P'}{P \; |[G]| \; Q \xrightarrow{i} P' \; |[G]| \; Q}$$

In this rule, G is the set of gates on which P and Q synchronise. If P can perform an internal action i, this rule allows that internal action to "push through" the parallel composition. Intuitively, if an action is internal to P, then it can also be considered to be internal to $P \; |[G]| \; Q$.

The second rule is the mirror image of the previous one, allowing internal actions to push through from Q:

$$\frac{Q \xrightarrow{i} Q'}{P \; |[G]| \; Q \xrightarrow{i} P \; |[G]| \; Q'}$$

We now need a rule allowing P and Q to synchronise on any gate in G. This rule is as follows:

$$\frac{P \xrightarrow{a} P', \; Q \xrightarrow{a} Q'}{P \; |[G]| \; Q \xrightarrow{a} P' \; |[G]| \; Q'} \quad [\; a \in G \lor (\exists \, v \bullet a = b\_v \land b \in G) \;]$$

Note that we have written the side-condition of this rule in such a way that either the simple Basic LOTOS actions or the data-bearing Full LOTOS actions may be handled. This is also the first inference rule we have seen with two conditions above the line. Both conditions must hold before the rule can be applied.

Finally, we need a rule allowing the processes in a parallel composition to perform, independently, actions which are not governed by the gate set:

$$\frac{P \xrightarrow{a} P'}{P \mid[G]\mid Q \xrightarrow{a} P' \mid[G]\mid Q} \quad [\ a \notin G \wedge (\nexists b, v \bullet a = b\_v \wedge b \in G)\ ]$$

The side-condition on this rule is intended to apply for exactly those values of a where the side-condition of the previous rule does not apply.

Since the above rule does not treat P and Q equally, that rule also has a mirror image rule, as follows:

$$\frac{Q \xrightarrow{a} Q'}{P \mid[G]\mid Q \xrightarrow{a} P \mid[G]\mid Q'} \quad [\ a \notin G \wedge (\nexists b, v \bullet a = b\_v \wedge b \in G)\ ]$$

A.11 Generalised choice

Above, we saw Basic LOTOS's simple choice construct. Full LOTOS includes a more flexible choice construct called generalised choice.

Generalised choice has the form

$$\mathit{choice}\ x : \{x_1, \ldots, x_n\}\ []\ B$$

where x is an action and B is a behaviour expression, possibly involving x. It is equivalent to an ordinary choice between several copies of B, each copy having had x replaced with each of $x_1 \ldots x_n$.

For example,

$$\mathit{choice}\ a : \{e, f, g\}\ []\ i;\ a;\ P$$

is equivalent to

$$i;\ e;\ P\ []\ i;\ f;\ P\ []\ i;\ g;\ P$$

The inference rule for generalised choice is as follows:

$$\frac{B[x/y] \xrightarrow{a} B'}{\mathit{choice}\ x : A\ []\ B \xrightarrow{a} B'} \quad [\ y \in A\ ]$$

This rule contains many of the same elements as the rule given above for process instantiation, and should be understandable.

A.12 Traces and refusals

We now present definitions of some operators and functions which are needed for the definitions in §4.3. Many are taken from [Led92, section 7]. We will define traces and refusals, and describe some related notation. These concepts will be needed in later chapters to describe the LOTOS development relations.

A *trace* is a sequence of observable actions. A trace is written $\langle a_1, a_2, \ldots, a_n \rangle$, where $a_1, a_2, \ldots, a_n$ is an ordered sequence of action names. The empty trace is denoted $\langle\,\rangle$. We use the CSP notation $\langle a, b, c \rangle$ for traces, rather than the LOTOS notation abc, because we use multi-character action names which would get confused using the LOTOS notation. For instance, the trace $\langle get, put \rangle$ would, in the LOTOS notation, be written *getput*.

A process performs a trace if it performs the observable actions listed in the trace in the order they appear in the trace. No other observable actions may be interspersed between the actions in the trace, but the process may perform some internal, unobservable actions while it performs the trace.

We can concatenate traces together using the "." (dot) operator. For example, we can write:

$$\langle a, b \rangle . \langle c, d \rangle = \langle a, b, c, d \rangle$$

Let us define a process P as follows:

$$P := a;\ b;\ a;\ stop$$

The traces that P can perform are $\langle\,\rangle$, $\langle a \rangle$, $\langle a, b \rangle$, and $\langle a, b, a \rangle$. Note that if a process can perform a given non-empty trace, it can also perform any prefix of that trace. In other words, a sequence of actions does not have to lead a process to a state of deadlock or any other particular state in order to be a trace of that process. All that is necessary is that the process can perform the actions in the order given.

We denote the set of traces which process P can perform $\mathrm{Tr}(P)$. We can rewrite our earlier statement as follows:

$$\mathrm{Tr}(P) = \{\langle\,\rangle, \langle a \rangle, \langle a, b \rangle, \langle a, b, a \rangle\}$$

As we saw in §A.3, the notation $B \xrightarrow{a} B'$ means that process B can perform action a and thereby evolve to process B'. Note that this evolution involves only one action, either internal or observable; $\rightarrow$ does not implicitly perform internal actions (unlike $\Rightarrow$, which we will describe below).

So, we can say that $(i;\ B) \xrightarrow{i} B$. This states that the behaviour $(i;\ B)$ can evolve to B by performing an internal action. Referring to our definition of P above, we can also say, for example, that:

$$\exists\, P' \bullet P \xrightarrow{a} P' \wedge P' \xrightarrow{b} (a;\ stop)$$

In terms of labelled transition systems, $B \xrightarrow{a} B'$ states that there is a transition labelled a leading from node B to node B' in the labelled transition system. The notation $B \stackrel{a}{\not\longrightarrow}$ means that there is no transition leading from B labelled a. For example, it is true that $(b;\ a;\ stop) \xrightarrow{b} (a;\ stop)$ but there is no behaviour B such that $(b;\ a;\ stop) \xrightarrow{a} B$. We can express this by writing:

$$(b;\ a;\ stop) \stackrel{a}{\not\longrightarrow}$$

The notation $B \stackrel{\sigma}{\Longrightarrow} B'$ means that process B can perform the trace σ and evolve to B'. As we mentioned above, performing a trace may also involve the process performing some internal actions before, between or after the externally visible actions mentioned in the trace. Let us define Q as follows:

$$Q := i;\ a;\ i;\ b;\ i;\ stop$$

With this definition, the following are all true:

$$Q \stackrel{\langle\rangle}{\Longrightarrow} Q$$
$$Q \stackrel{\langle\rangle}{\Longrightarrow} (a;\ i;\ b;\ i;\ \mathit{stop})$$
$$Q \stackrel{\langle a\rangle}{\Longrightarrow} (i;\ b;\ i;\ \mathit{stop})$$
$$Q \stackrel{\langle a\rangle}{\Longrightarrow} (b;\ i;\ \mathit{stop})$$
$$Q \stackrel{\langle a,b\rangle}{\Longrightarrow} (i;\ \mathit{stop})$$
$$Q \stackrel{\langle a,b\rangle}{\Longrightarrow} \mathit{stop}$$

We can see, then, that $\stackrel{\langle\rangle}{\Longrightarrow}$ allows a process to skip over a number of internal actions without performing any externally visible actions. We can define $\stackrel{\langle\rangle}{\Longrightarrow}$ as follows:

$$B \stackrel{\langle\rangle}{\Longrightarrow} B' \Leftrightarrow B = B' \lor \exists\, C \bullet (B \stackrel{i}{\longrightarrow} C \land C \stackrel{\langle\rangle}{\Longrightarrow} B')$$

Any other trace (any non-empty trace) can be formed by prefixing a single action onto another (possibly empty) trace. We can now complete the definition of $\Rightarrow$ as follows:

$$\text{If } x \neq i,\ B \stackrel{\langle x\rangle.\sigma}{\Longrightarrow} B' \Leftrightarrow \exists\, C, C' \bullet (B \stackrel{\langle\rangle}{\Longrightarrow} C \land C \stackrel{x}{\longrightarrow} C' \land C' \stackrel{\sigma}{\Longrightarrow} B')$$

We can leave off the second behaviour, and write $B \stackrel{\sigma}{\Longrightarrow}$. This means that behaviour B can perform the trace σ. We can define this formally as follows:

$$B \stackrel{\sigma}{\Longrightarrow}\ \Leftrightarrow \exists\, B' \bullet B \stackrel{\sigma}{\Longrightarrow} B'$$

As described above, $\mathrm{Tr}(P)$ is the set of traces which process P can perform. We can now define it formally as follows:

$$\mathrm{Tr}(P) = \{\sigma \mid P \stackrel{\sigma}{\Longrightarrow}\}$$

Given a process and a trace, there need not be only a single behaviour to which the process can evolve by performing the trace. For example, let us define R as follows:

$$R := a;\ b;\ \mathit{stop}\ [\,]\ a;\ c;\ \mathit{stop}$$

If R performs the trace $\langle a\rangle$, it could evolve to either $(b;\ \mathit{stop})$ or $(c;\ \mathit{stop})$. On the other hand, R cannot perform the trace $\langle b\rangle$ at all. We define P *after* σ to be the set containing those behaviours to which P can evolve by performing trace σ. So, in our example, we get the following results:

$$R\ \mathit{after}\ \langle a\rangle = \{(b;\ \mathit{stop}), (c;\ \mathit{stop})\}$$
$$R\ \mathit{after}\ \langle b\rangle = \{\}$$

We can define P *after* σ as follows:

$$P\ \mathit{after}\ \sigma = \{P' \mid P \stackrel{\sigma}{\Longrightarrow} P'\}$$

In our example, R cannot perform the action b. We say that it *refuses* b. If a process can perform an action, we say that it *offers* the action. This term emphasises that, even if an action can be performed, it need not necessarily be performed; however, it cannot be performed unless it was first offered.

The term *refuse* is a little misleading, however. A process refuses a given action if it is *possible* for it to deadlock when synchronised with an environment offering just that one action. It is therefore possible for a process to both offer and refuse an action at the same time. For example, the following behaviour both offers and refuses a:

$$i;\ a;\ stop\ []\ i;\ stop$$

So, R initially offers only one action, a. The set of actions which a process offers before it has performed any observable actions is called the *initials* of the process.

If R performs a, it reaches a state where it offers b, or a state where it offers c. However, it is often more useful to know that, after performing a, R could refuse b, or it could refuse c. Since we do not know which state R has evolved to, we cannot tell which of b or c it will refuse; on the other hand, we know that it cannot refuse both. Therefore, we define refusals like this: we imagine that the process first performs a given trace, then we offer a set of actions to the resulting process. If it is possible for the process to refuse all of the actions in the set, we say that it refuses that set of actions, after the given trace.

We can now define the *refusals* of a process. $\mathrm{Ref}(P, \sigma)$ is a set containing zero or more sets of actions. A set of actions is in $\mathrm{Ref}(P, \sigma)$ exactly if P can refuse all members of the set after performing σ. We can define $\mathrm{Ref}(P, \sigma)$ more formally as follows:

$$\mathrm{Ref}(P, \sigma) = \{S \in \mathbb{P}\, Act \mid \exists\, P' \in (P\ \mathit{after}\ \sigma) \bullet \forall\, a \in S \bullet P' \overset{a}{\not\Longrightarrow}\}$$

If we take $Act = \{a, b, c\}$, we get the following results using the process R defined above:

$$\begin{array}{l}\mathrm{Ref}(R, \langle\rangle) = \{\{\}, \{b\}, \{c\}, \{b, c\}\} \\ \mathrm{Ref}(R, \langle a\rangle) = \{\{\}, \{a\}, \{b\}, \{a, b\}, \{c\}, \{a, c\}\} \\ \mathrm{Ref}(R, \langle b\rangle) = \{\} \\ \mathrm{Ref}(R, \langle a, b\rangle) = \mathrm{Ref}(R, \langle a, c\rangle) = \mathbb{P}\, Act\end{array}$$

The third of these examples needs some explanation. Due to the way Ref is defined, it makes sense to refer to a process's refusals after a trace which the process cannot perform. The process can refuse a set of actions after a trace exactly if it can perform the trace and then refuse all the actions in the set. If it cannot perform the trace, then by definition, it cannot refuse the actions after that trace, regardless of which actions are in the set, or even if the set is empty. This leads to the following property:

$$\sigma \in \mathrm{Tr}(P) \Leftrightarrow \mathrm{Ref}(P, \sigma) \neq \{\}$$

(This property is, in fact, implied by the next property but one given below.)

If a process can refuse a given set of actions after a given trace, it follows that it can also refuse any subset of that set of actions after the same trace. For example, since R can perform $\langle a\rangle$, then refuse $\{a, b\}$, it follows that it must also be able to refuse the sets $\{a\}$, $\{b\}$ and $\{\}$ after $\langle a\rangle$. We can state this property as follows:

$$\forall\, S, T \bullet S \in \mathrm{Ref}(P, \sigma) \wedge T \subseteq S \Rightarrow T \in \mathrm{Ref}(P, \sigma)$$

It necessarily follows from this property that, if $\mathrm{Ref}(P, \sigma)$ contains any sets at all, it must also contain the empty set. That is, if P can perform σ, then P must be able to refuse the empty set after performing σ:

$$\sigma \in \mathrm{Tr}(P) \Leftrightarrow \{\} \in \mathrm{Ref}(P, \sigma)$$

A.13 Conclusion

This concludes our tour of LOTOS. We have presented a subset of the notation and its semantics which should be sufficient for understanding the discussions in this dissertation.

Bibliography

[AvdL90] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. *SIGPLAN Notices (ECOOP/OOPSLA '90 Proceedings)*, 25(10):161–168, October 1990.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.

[BBDS96] Howard Bowman, Eerke Boiten, John Derrick, and Maarten Steen. Strategies for consistency checking, the choice of unification. Technical Report 5-96*, University of Kent, Computing Laboratory, University of Kent, Canterbury, UK, February 1996.

[BBSDS97] Howard Bowman, Charles Briscoe-Smith, John Derrick, and Ben Strulo. On behavioural subtyping in LOTOS. In *FMOODS '97: Formal Methods for Open Object-based Distributed Systems*, volume 2, pages 335–351, Canterbury, UK, July 1997. Chapman & Hall.

[BCC+95] Kim Bruce, Luca Cardelli, Giuseppe Castagna, et al. On binary methods. *Theory and Practice of Object Systems*, 1(3), May 1995.

[BD99] Howard Bowman and John Derrick. A junction between state based and behavioural specification. In A. Fantechi P. Ciancarini and R. Gorrieri, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 213–239. Kluwer, February 1999. Invited Paper.

[BDLS95] Howard Bowman, John Derrick, Peter Linington, and Maarten Steen. FDTs for ODP. *Computer Standards & Interfaces*, 17:457–479, 1995.

[BDLS96] Howard Bowman, John Derrick, Peter Linington, and Maarten Steen. Cross-viewpoint consistency in open distributed processing. *Software Engineering Journal*, pages 44–57, January 1996.

[BJR+00a] Grady Booch, Ivar Jacobson, Jim Rumbaugh, et al. Object constraint language specification. In *OMG Unified Modeling Language Specification* [BJR+00b], chapter 7.

[BJR+00b] Grady Booch, Ivar Jacobson, Jim Rumbaugh, et al. *OMG Unified Modeling Language Specification*. Object Management Group, 1.3 edition, 2000.

[Bow96] Howard Bowman. Formal specification and modelling. Technical Report TR 15-96, Computing Laboratory, University of Kent at Canterbury, Canterbury, UK, 1996.

[Bri88] Ed Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing and Verification*, volume VIII, pages 63–74, North-Holland, 1988. Elsevier Science Publishers B. V.

[Bro75] Frederick P. Brooks, Jr. *The Mythical Man-month: Essays on software engineering*. Addison-Wesley, 1975.

[BS86] Ed Brinksma and Giuseppe Scollo. Formal notions of implementation and conformance in lotos. Technical Report Memorandum INF-86-13, Twente University of Technology, Department of Informatics, Enschede, The Netherlands, December 1986.

[BSS87] Ed Brinksma, Giuseppe Scollo, and Chris Steenbergen. LOTOS specifications, their implementations and their tests. In *Proceedings IFIP Protocol Specification, Testing and Verification VI*, pages 349–360, 1987.

[CDD+90] David Carrington, David Duke, Roger Duke, Paul King, Gordon Rose, and Graeme Smith. Object-Z: An object-oriented extension to Z. In Son T. Vuong, editor, *Formal Description Techniques II: Proceedings of the IFIP TC/WG6.1 Second International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE '89*, pages 281–296, North-Holland, 1990. Elsevier Science Publishers B. V.

[CHC90] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *ACM Symposium on Principles of Programming Languages*, volume 17, pages 125–135, San Francisco, California, 1990. ACM Press.

[CL90] Elspeth Cusack and Michael Lai. Object-oriented specification in LOTOS and Z, or my cat really is object-oriented! In *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[CR92] E. Cusack and G. H. B. Rafsanjani. ZEST. In Susan Stepney, Rosalind Barden, and David Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 113–126. Springer-Verlag, 1992.

[CRS90] Elspeth Cusack, Steve Rudkin, and Chris Smith. An object-oriented interpretation of LOTOS. In Son T. Vuong, editor, *Formal Description Techniques II: Proceedings of the IFIP TC/WG6.1 Second International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE '89*, North-Holland, 1990. Elsevier Science Publishers B. V.

[DBBS96] John Derrick, Eerke Boiten, Howard Bowman, and Maarten Steen. Supporting ODP — translating LOTOS into Z. In E. Najm and J.-B. Stefani, editors, *First IFIP International Workshop on Formal Methods for Open Object-Based Distributed Systems*, pages 399–406, Paris, March 1996. Chapman & Hall.

[DBBS99] John Derrick, Eerke Boiten, Howard Bowman, and Maarten Steen. Viewpoints and consistency: Translating LOTOS to Object-Z. *Computer Standards & Interfaces*, 21:251–272, 1999.

[Fis97] Clemens Fischer. CSP-OZ: A combination of Object-Z and CSP. In *FMOODS '97: Formal Methods for Open Object-based Distributed Systems*, volume 2, pages 423–438, Canterbury, UK, July 1997. Chapman & Hall.

[FM94] Kathleen Fisher and John C. Mitchell. Notes on typed object-oriented programming. In *Proceedings of Theoretical Aspects of Computer Software (TACS '94), Sendai, Japan*, volume 789 of *LNCS*, pages 844–886. Springer, 1994.

[FM96] Kathleen Fisher and John C. Mitchell. The development of type systems for object-oriented languages. Technical Report STAN-CS-TN-96-30, Stanford, November 1996.

[Goo95] Andrew Goodchild. An evaluation scheme for trader user interfaces. In Kerry Raymond and Liz Armstrong, editors, *Open Distributed Processing, Experiences with*

Distributed Environments: Proceedings of the Third IFIP TC 6/WG 6.1 International Conference on Open Distributed Processing, pages 145–156, Brisbane, Australia, February 1995. Chapman & Hall on behalf of the International Federation for Information Processing (IFIP).

[HH90] J. He and C. A. R. Hoare. Prespecification and data refinement. In *Data Refinement in a Categorical Setting.* Technical Report PRG–PRG–90, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, November 1990.

[HHS86] J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In Bernard Robinet and Reinhard Willhelm, editors, *ESOP 86: European Symposium on Programming*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196, Saarbrücken, March 1986. Springer-Verlag.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[ISO89] ISO—Informations Processing Systems—Open Systems Interconnection. LOTOS—a formal description technique based on the temporal ordering of observational behaviour. ISO/IEC 8807, 1989. International Organisation for Standardisation, Geneva.

[ISO93] ISO/IEC JTC1/SC21/WG7. Basic reference model of open distributed processing—parts 1–4. ITU-T X.901 and ISO/IEC 10746-1, July 1993. International Organisation for Standardisation, Geneva.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 1986.

[Lam94] Leslie Lamport. Introduction to TLA. Technical Report 1994-001, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, USA, December 1994.

[Led92] Guy Leduc. A framework based on implementation relations for implementing LOTOS specifications. *Computer Networks and ISDN Systems*, 25:23–41, 1992.

[Lin91] Peter F. Linington. Introduction to the open distributed processing basic reference model. In J. de Meer, V. Heymer, and R. Roth, editors, *IFIP TC6 International Workshop on Open Distributed Processing*, pages 3–13, Berlin, September 1991.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989. Also available as MIT Technical Memo MIT/LCS/TM-373.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, November 1994.

[Mey92] Bertrand Meyer. *Eiffel: the language.* Prentice Hall, 1992.

[Mil89] Robin Milner. *Communications and Concurrency.* Prentice Hall, 1989.

[Nie95] Oscar Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995. Earlier version in proceedings of *OOPSLA '93*, in *ACM Sigplan Notices*, 28(10):1–15, October 1993.

[PST96] Ben Potter, Jane Sinclair, and David Till. *An Introduction to Formal Specification and Z.* Prentice-Hall, 2 edition, 1996.

[Ros92] Gordon A. Rose. Object-Z. In Susan Stepney, Rosalind Barden, and David Cooper, editors, *Object Orientation in Z*, Workshops in Computing, pages 59–77. Springer-Verlag, 1992.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.

[Rud92] Steve Rudkin. Inheritance in LOTOS. *Formal Description Techniques*, IV, 1992.

[Sch00] Steve Schneider. *Concurrent and Real-time Systems.* Wiley, 2000.

[Seg92] Roberto Segala. A process algebraic view of I/O automata. Master's thesis, Massachusetts Institute of Technology, 1992.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall International, second edition, 1992.

[ST95] Richard O. Sinnott and Kenneth J. Turner. Applying formal methods to standard development: The Open Distributed Processing experience. *Computer Standards & Interfaces*, 17:615–630, 1995.

[ST97] Richard O. Sinnott and Kenneth J. Turner. Type checking in open distributed systems: a complete model and its Z specification. In J. Rolia, J. Slonim, and J. Botsford, editors, *Open Distributed Processing and Distributed Platforms*, pages 85–98. Chapman & Hall, May 1997.

[Str95] Ben Strulo. How firing conditions help inheritance. In *ZUM '95: The Z Formal Specification Notation. 9th International Conference of Z Users*, Lecture Notes in Computer Science, pages 145–156, Limerick, September 1995. Springer.

[Tur93] Kenneth J. Turner, editor. *Using Formal Description Techniques.* Wiley, Chichester, 1993.

[UP97] Irek Ulidowski and Iain Phillips. Formats of ordered SOS rules with silent actions. In *TAPSOFT '97: Theory and Practice of Software Development. 7th International Joint Conference CAAP/FASE*, volume 1214 of *Lecture Notes in Computer Science*, pages 297–308, Lille, April 1997. Springer-Verlag.

[vG90] R. J. van Glabbeek. The linear time-branching time spectrum. In CONCUR *'90: Theories of Concurrency—Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[WC90] J-P. Wu and S. Chanson. Translation from LOTOS and Estelle specifications to extended transition system and its verification. In Son T. Vuong, editor, *Formal Description Techniques II: Proceedings of the IFIP TC/WG6.1 Second International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE '89*, pages 533–549, North-Holland, 1990. Elsevier Science Publishers B. V.

[WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement and Proof.* Prentice-Hall, 1996.