Chapter 4

The provision of safety requirements from fault trees and their validation in formal models

In this chapter we address the problem of how to link directly standard safety analysis techniques with formal software models that are being refined towards implementation. Here, we focus on one of the most popular techniques – fault tree analysis – and detail within a formal framework the derivation of requirements from fault trees and their incorporation in system models. A common semantics basis of labelled transition systems is chosen, thereby avoiding extra difficulties involved in translations.

To come closer to industrial practice, we provide a procedure that allows much independence in the activities of building the fault tree, generating requirements and then relating them to models. Further scope is provided through a number of *consistency relations* to reflect varying levels of satisfaction of requirements needed at different stages in development. This overall flexibility allows the system builder to derive a series of meaningful results, based upon a gradual and rigorous refinement of a formal model.

As described in the previous chapter, there has been ongoing work in requirements capture, with the use of formal methods to sharpen the analysis, but the problem of 'threading through' the results to formal models has hardly been treated.

4.1 The use of FTA for software

Fault Tree Analysis (FTA) has been accepted as a very useful part of safety analysis for the engineering of safety-critical systems in general – they are part of 'best practice', a stock-in-trade tool. A definitive guide to FTA, to which we refer frequently, is the Fault Tree Handbook [VGRH81].

During the past decade or so, these techniques have also been found useful in

analysing critical aspects of systems to assist in the production of safety-related software. In particular, a technique called Software Fault Tree Analysis (SFTA) has been developed to analyse critical code. In [LH83], there is a general discussion of this approach and then experiences are reported of its application to Intel 8080 Assembly code that was used for controlling the flight and telemetry of a space craft; this project is described in more detail in [Har82]. SFTA has also been applied to ADA programs [LCS91].

In common with the production of most software of a critical nature, such techniques may be enhanced by the use of formal methods which can remove ambiguity and inconsistency and generally increase confidence in the safety of the design and subsequent implementation. In the following sections we provide a summary of the technique and then review the use of FTA for software, discussing how formal methods can help assess (and then justify) its suitability.

4.1.1 Summary of the technique

In the safety lifecycle model, the initial stage consists of identifying hazards at a system-wide level. FTA is a technique that provides an iterative means to determine the events or fault (hazard) causes that lead to a *top-level fault*, an event which is identified as posing some serious risk to the safety of the system. These events are deduced using whatever brainstorming techniques are available, so FTA may in the process prompt the discovery of further hazards at a wider level. Depending upon interpretation, a problem which is treated in section 4.2, FTA's may encompass a number of aspects. The three main ones are: causality, where one or more hazards may be sufficient or necessary to cause another one; temporality (*event sequences*, where a particular chain of events can lead to a hazard); and composition (a hazardous item may be hazardous due to one of its components).

FTA takes a top-down view, starting with those hazards which are visible to users and successively decomposes and/or works backwards to previous causes. In so doing a tree structure is formed – a *fault tree* – whose nodes are events. It is possible for this to be an infinite sequence, so at some stage, one must decide to halt the process, thereby arriving at *primary events*, which for the purposes of a particular fault tree are the root causes (temporal and structural). These root causes are represented as *leaf nodes*. For each branch in the tree, the type of choice is indicated by a logical combinator such as `AND` or `OR` depending upon the relationship between an event and its contributing causes. Fault trees may be constructed in stages, so an initial tree may include some behaviour of the software without incorporating any methods of control. As the model is refined, the fault tree may

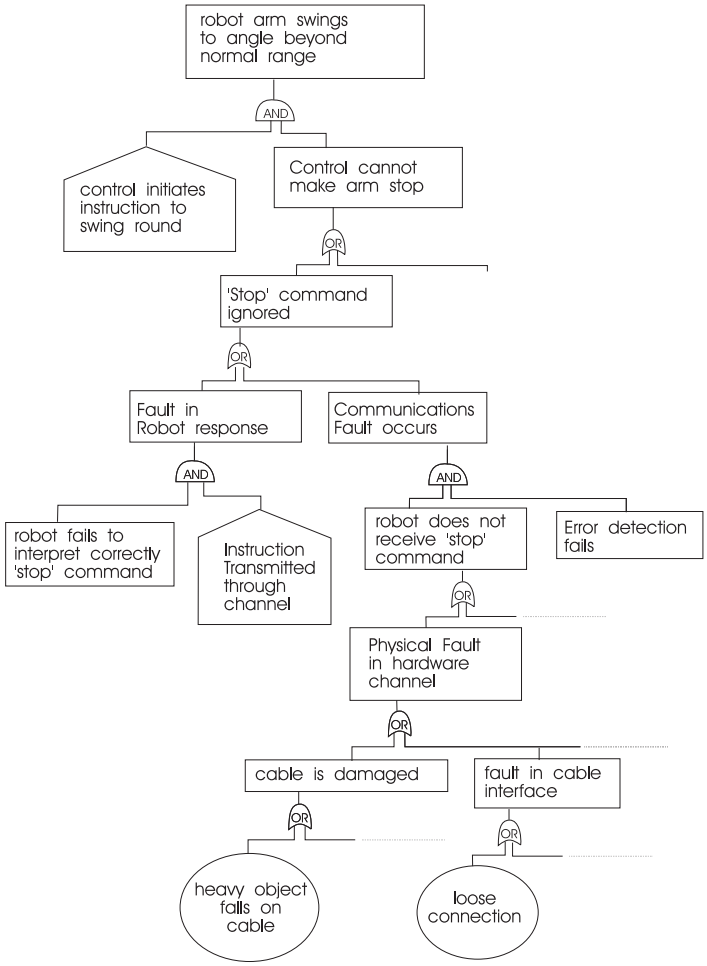

Figure 4.1: Part of Fault Tree Analysis for a remotely controlled robot

be expanded to account for the controlling mechanisms.

We now proceed to describe in more detail the process of constructing trees. There are a number of useful concepts that have emerged from industrial experience over two or three decades in the process of enlarging (or elaborating) the fault tree. We summarise the ones given in section V of the fault tree hand book and illustrate some of these concepts using an FTA for a robot controlled remotely via some cable, as shown in Figure 4.1. We also make a few observations concerning interpretation issues, which we discuss further in section 4.2.

The method starts at the top level fault and recursively seeks to generate causes as far as is deemed necessary. Thus the proper selection and definition of events is essential to the construction of an appropriate tree.

Thus the first ground rule (section V.7 of [VGRH81]) is:

Write the statements that are entered in the event boxes as faults; state precisely

what the fault is and when it occurs.

The events are the building blocks for the trees and may be variously classified into different types. Below is the classification in [VGRH81] (where events are termed *faults* if they are initiated by other events and *failures* if they are basic initiating events).

- The *rectangle* indicates an event to be analysed further
- The *circle* indicates a basic fault event or primary failure of a component. It requires no further development.
- The *house* is used for events which normally occur in the system. It represents the continued operation of the component, and its probabilty is the reliability of the component.
- The *diamond* is used for non-primal events which are not developed further for lack of information or insufficient consequence
- The *oval* is used to indicate a condition. It defines the state of the system that permits a fault sequence to occur. It may be normal or result from failure.
- The *AND* gate serves to indicate that all *input* events are required in order to cause the *output* event.
- The *OR* gate indicates that one or more of the input events are required to produce the gated event.

Note that not all events in the tree are faulty by themselves – those of the type 'house' may act simply as propogation media under normal operation.

The second ground rule helps the elaboration of the tree through determining the appropriate type of gate from information about the state of a fault:

> If the answer to the question "Can this fault consists consist of a component failure?" is "Yes", then classify the event as a 'state-of-component fault'. If the answer is "No", then classify the event as a 'state-of-system' fault.

For a 'state-of-component fault', the handbook advises adding an 'OR'-gate with inputs being components that can cause the fault.

For a 'state-of-system fault', the handbook advises looking for minimum necessary, sufficient and immediate causes using any of the gates.

In order to state precisely what a fault is, there are a number of aspects and issues which are described in [Ves81], sections V.1–V.6:

(i) Fault Existence vs. Fault Occurrence

Some faults may be transient, in which case a distinction should be made between fault existence and fault occurrence since, in particular, this has a bearing on determining probabilities of faults. For instance, in Figure 4.1, the event 'cable is damaged' will remain a fault unless dealt with; however; 'Loose connection' may be a transient fault.

However, for the construction itself, the handbook advises "we need concern ourselves only with the phenomenon of occurrence." That is to say, we explore what events may occur and treat likelihoods later.

(ii) Passive vs. Active Components

In most cases it is convenient to view components as either 'active' or 'passive':

- 'active' components trigger events or event sequences (examples include relays, resistors, pumps, and various control mechanisms)
- 'passive' components act as propagation media (examples include pipes, wires, bearings, etc..)

Passive components may lie between two active ones, though this ordering may or may not be reflected in the tree: for example, in Figure 4.1, there is the event " 'stop' command ignored". The fault may have been caused by the robot, in which case, the occurrence of the fault is actually dependent upon the transmission media, which acts a passive component. Here it is included as a house event "Instruction transmitted through channel", which site next to "robot fails to interpret correctly 'stop' command".

(iii) Component Failure Categories Fault occurrences may be categorised as:

- *Primary*: fault occurs in the environment for which the component is intended to operate normally, e.g. the top-level event of Figure 4.1.
- *Secondary*: fault occurs outside the environment for which the component is intended
- *Control*: fault occurs because a component operates properly but at the wrong time or wrong place etc.

(iv) Failure mechanism, Failure mode and Failure Effect

These concepts are important in determining the inter-relationship between events

- "effects" - why the particular failure is of interest - what are the effects on the system?
- "modes" - what aspects of component failure are of concern.
- "mechanisms" - how a particular failure mode can occur. (the above mode has mechanism 'loss of data across link' and 'lossy channel').

For example, in Figure 4.1 the event "cable is damaged" is a failure mechanism for the mode "Physical Fault in hardware channel".

(v) The 'immediate cause' concept

This is a general view on relationship within gates between inputs and outputs and has been subject to criticism when formally analysed.

Once a top level event (in Figure 4.1, viz: 'robot arm swings to angle beyond normal range') has been selected, the immediate, necessary and sufficient causes for its occurrence are determined. The immediate causes are those events which occupy a place just in front of the top level event - whether physically along side (or part of) or immediately preceding in time.

These cover the essentials of fault tree construction, as recommended in the Fault Tree Handbook. A few further rules are mentioned as the result of the experience of safety analysts:

1. 'No Miracles Rule': If the normal functioning of a component propagates a fault sequence, then it assumes that the component functions normally.
2. 'Complete the Gate Rule': All inputs to a particular gate should be completely defined before further analysis of any one of them is undertaken.
3. 'No Gate-to-Gate Rule': Gate inputs should be properly defined fault events, and gates should not be directly connected to any other gates.

4.1.2 Using formal methods to assess the suitability of FTA for software

We start this section with some motivating discussion for the use of FTA for software (the same considerations may also be applied to other safety analysis techniques). We may ask, especially with respect to the delivery of systems with safety-related software: How effective or faithful is a fault tree at capturing hazards or faults affecting software? Does FTA support adequate analysis?

Since computers generally operate according to deterministic physical laws, much depends on two factors: the ability of the safety analysis to be able to account for such deterministic behaviour and a facility for safety requirements to be consequently generated and reliably implemented in the software. Experiences as reported in [LH83] indicate that FTA can prompt the discovery of hazards and faults that escape other techniques. As part of the conclusions in that paper, a number of salient qualities of FTA emerge including:

- the encouragement of a focus on catastrophic events
- the provision of a systematic approach for the consideration of the causes of these events
- the convenient storage of information.

Even so, FTA may not be the most suitable technique for software. Indeed, researchers at the High Integrity Systems Group at the University of York, have focused attention on software structures and found that fault trees may become very large and unweildy. They have concluded that some adaptation of FTA and FMECA is required, resulting in the suggested Failure Propogation and Transformation Notation (FPTN) to overcome some limitations [FJMP94]. However, it may be noted with regard to the problem of fault trees burgeoning in size, that targeting specifications more than program code and making use of abstraction can be of considerable help.

Although some alternative to traditional FTA may well be more effective, it is worth considering beforehand the issue of what we mean by fault trees. Only then do we really know the extent to which FTA is effective. A precise understanding of fault trees requires the use of formal methods; only recently have they been used to address this issue, yet there have already been some significant findings. It should be noted their use depends upon capturing "as best as we can tell" the informal tree, i.e. a process that requires validation.

When FTA has been put under the spotlight of formal methods, there have been revealed ambiguities and inconsistencies in possible interpretations that were previously not known. It is shown in [BA93] that a given tree can reasonably be expected not to have the required properties according to the interpretation recommended even in [VGRH81]. In [BCG91], it is mentioned that there is an inconsistency between definitive sources regarding interpretation of gates. As an answer to these two problems, in both papers it is suggested that the user has to choose carefully between a number of interpretations depending upon the context. In this way, the formal analysis enables one to make informed suggestions

regarding the conventional interpretation of trees and the consequences for software. A more detailed discusssion of interpretation problems is given in Section 3 of [G94].

Formal reasoning is also desirable beyond the locality of individual events or gates, to clarify, for instance, what are common failure modes and component structures. This is important when requiring some quantification of risk through the calculation of probabilities for e.g. *minimal cutset forms* (or some derivative). This work is even more in its infancy, with one or two notable exception such as [GMW95], where formal analysis is conducted mainly to establish under what timing constraints a top-level hazard can actually occur from the lower level events. This is but a small step to determining the effect of a given hazard on the reliability of a system, in which it may be difficult to know to what extent events at different parts of the tree are independent.

However, putting aside problems of interpretation, the use of formal methods in these papers have supported the case that wide-ranging safety-related aspects of software systems can be captured using FTA as a safety analysis technique and, further, the output of the analysis can be formalised. There is still plenty of scope for establishing the useful extent of such analysis with the need to provide more examples. This may well confirm the need for flexibility in how a Fault Tree can be interpreted (and in turn refine the current approaches to formalisation, reflecting the flexibility). In general, as we hope to show in this chapter, fault trees serve as a very useful basis for generating requirements.

Since a fault tree has such wide scope, a further important question arises: "What are the criteria that make a formal fault tree faithful to the original FTA? How do we ensure faithfulness?" As hinted above, this is a subjective matter, open to personal interpretation and so requiring validation. The criteria should include some notion of completeness – i.e. every part of the informal analysis should be accounted for and, preferably, there should also be simplicity. This may be facilitated by having procedures both for formalising the elements of a tree (the events) and for constructing trees. An algorithm for event formalisation is given in [GW95], but there has not yet been stated a procedure for trees. We provide such a procedure in this chapter. We then need to derive software requirements based on the formal FTA and show that they can be usefully applied to a formal model. This is achieved using a common semantics basis on which are defined some relation(s) between faults and system models, interpreted over transition systems. In this chapter we use the semantic framework provided in [BA93]) that are to underpin our new relations defined in section 4.4.

4.2 Semantics of fault trees

This section reviews briefly the key elements for interpreting fault trees.

There are two fundamental semantic notions which are used to make up a traditional fault tree:

1. an EVENT
2. a GATE which relates events.

Instances of events represent faults or hazards and are the building blocks for the trees. The first issue to address is how we regard events: either we regard them as *atomic*, i.e. indivisible in some sense, in which case we can give them a simple denotation; or as non-atomic, i.e. they may be expressed in terms of simpler elements.

In the case that events are not atomic, we need to be able to reason about them. Doing this formally requires the task of *event modelling*. Work based on the CSDM model [BCG91, G94] has provided a formal semantics with explicit timing notions; an extension of CSDM, ECSDM adds a state-based system representation for events in the manner of VDM [Jon90] and an algorithm for event construction [GW95] (which provides some examples of the procedure). A supporting task is the classification of events into different types, for which one approach is based on the role of an event in a particular tree, as cited above, following the Fault Tree Handbook. Other classifications are based on the position of the event in a tree [G94]; and on the structure of an event [GW95].

In this chapter we wish to provide a very general framework and thus do not make explicit any particular event model. In some cases we may model an event at a low level of granularity and then use appropriate abstraction to follow [BA93], in which events are formalised as atomic propositions which are interpreted as sets of states. The view chosen should reflect the context, but generally we do not impose the requirements of atomicity and this is reflected in some of the definitions which omit mention of atomicity.

Relating the events to each other formally requires a precise notion of the meaning of a gate. As a general rule, one chooses semantics appropriate to the real-life situation being analysed and modelled. As a starting point, one can look to informal notions: one traditional interpretation of gate semantics is *propositional*, i.e. at any gate, event causes are immediate, sufficient and necessary (stipulated as the 'immediate cause concept' in section V.6 of [VSGH81]). In this case, duration may be incorporated in the events themselves and temporal sequences modelled by a chain of events at successively deeper levels in the fault

tree. However, it is far more natural to allow temporality between events. Using Figure 4.1 as an illustration, we may draw out the following sequence of events that lead to the hazard of a robot arm swinging out of control and possibly hitting someone:

1. The communications wire is frayed as a heavy object falls on it [this latter is itself an environmental fault cause to be part of the fault tree]
2. A communications fault develops [due to physical degradation] and goes undetected
3. A certain instruction given to the robot arm is initiated and subsequently cannot be stopped since there is a fault in the communications.
4. The robot arm swings without control and hits someone

We may observe that event (2) may or may not occur immediately after event (1). However, it is likely that there is duration between events (2) and (3). Further, in the tree, event (2) may be modelled by an `AND` gate with two input events (viz 'communication fault develops' and 'communication fault is undetected'). In reality, the detection of the fault must occur after the fault arises, so these events do not happen simultaneously. Thus, a propositional semantics applied to the tree derived from this fault sequence would have to use a number of unnatural squeezes, but may still be inadequate since some events *must* satisfy temporal relationships with others – which cannot be cleanly encapsulated in any one event. The Fault Tree Handbook does define 'conditioning events' which tack on conditions to gates, including the 'Priority AND' gates which would be able to specify the temporal constraint of one input occurring before another. However, realistically this should carry information about duration, which would imply temporal semantics.

In response to the awkwardness of propositional semantics in such scenarios, Bruns and Anderson have introduced both a temporal logic and a propositional logic gate semantics; the former has a simple structure for which "We assume only that a system model can be represented as a transition system or as a set of sequences of states." This approach is readily applicable to a variety of application domains since it does not impose big restrictions on the event structures. However, in view of the fact that some guides such as the Fault Tree Handbook have become somewhat definitive, the introduction of temporal semantics to supplement or replace the propositional semantics needs more justification, e.g. through case studies with specific examples which can only be modelled in a temporal semantics.

Górski et al provide in [BCG91] a detailed event model and temporal gate semantics as part of a Common Safety Description Model (CSDM). The former provides useful

information on how safety analysis techniques can be formalised in a way that is true to the original intention. This is further supported by an algorithm to generate formalised events based on the CSDM model and provides a classification of typical event classes [GW95]. But it is only one model – there are bound to be alternatives whose relative faithfulness may be assessed. As the range of effective applications of FTA is large, then it is likely that a number of distinct event models are required to cater for each of them.

Once event and gate semantics have been established, the fault trees themselves are interpreted in terms of the gate conditions – for instance, if the conditions are propositions, then the tree can be interpreted as their conjunction.

4.3 Constructing Fault trees and deriving safety requirements

In this section we combine the activities of constructing fault trees with the derivation of requirements.

4.3.1 From FTA to safety requirements

Assuming the foundations for development, as provided in Chapter **??**, section 3.4.1, safety requirements may be systematically derived from FTA. To recap, once the risks have been determined for the various hazards, requirements are generated to provide integrity, which in turn is achieved by methods of control. The requirements will usually be directed at a selection of hazards, which may be tackled directly or indirectly. Hence, decisions have to be made regarding for which hazards one plans to introduce methods of control and where – trying to prevent a hazard and all its causes would be superfluous. As an example, in data communications, error correction could in theory take place at many different points, but usually it is targeted for specific stages or levels.

A solution of this problem requires understanding through various analyses how events may combine to lead to a hazard. It must appeal to the context of the particular events, informed by the representations of the hazard structures and system models respectively. As described in section 3.5.2.2, the main role of fault trees is to provide a useful basis to reason about hazards. In particular, they enable the analyst to determine where to introduce or apply methods of control. Here we present some guidelines for this activity, based upon consideration of the fault tree, which we assume has some formal representation.

First, one may observe that in any fault tree it suffices to cut a line of hazards

right across all branches (or their trunks) in the subtree that stem from the node for once sufficient hazard causes are removed, the hazard ceases to arise. In deciding where to cut, the next step is to consider the hazards themselves. Given the informal identification of a real hazard, how do we find the appropriate method(s) of control in the formal domain?

In a formal model, we need to account for two aspects concerning the tree of hazards. The first is *static* analysis in which one examines the structure of the tree's representation, where one of the main techniques is to try to use decomposition in such a way that proving a property of a number of components enables proof of a property about a greater system. If an individual component's behaviour is too varied, then a higher level of abstraction should be used that takes into account it's interaction with other components. Decomposition may be attempted with respect two aspects: the system and the properties. In our example fault tree, we have that there may be a data corruption fault in the system. Decomposing the system reveals that it lies in the upper level link interface; further, the actual property which makes this system hazardous may be stated as an $\vee$ (or) composition – whose components are properties about where and how it is corrupted.

The second aspect is *dynamic* analysis and concerns the temporal ordering or sequence of events that may lead to a certain hazard – it is possible that an error in the data link layer may lead to the corruption of patient data, yet operate reliably again before the corruption is detected. This indicates the delicate interplay between actions and states, which formal models need to capture. Ideally, there should be tools that simulate any behaviour possible in the tree.

As a general rule of thumb, following the philosophy that prevention is better than cure, one may choose (as here) that the hazards initially selected are leaf nodes; then successively move up the tree only when an appropriate method of control cannot be found for a given hazard.

4.3.2 Verification and Validation

Formalisation of safety analysis, to reveal the ambiguity and inconsistency to which – like any other informal process – it is prone, is only established through the activities of verification and validation. Safety requirements also benefit from formalisation, and can be treated as objects which form part of our design activities. The formal models which incorporate these requirements in the context of their refinement towards a full software implementation are also objects in this process.

The tasks of verification and validation are both central to our formal treatment

	verification	**validation**
safety analysis	*prerequisite: formalise safety analysis (e.g., fault trees)*	
	check consistency of the internal components in the analysis	check that the formalisation is a true reflection of informal analysis analyse, using information from the verification, to cast light on the safety analysis
safety-related formal model	*prerequisite: derive safety requirements from the formal safety analysis*	
	check consistency of the internal components of the model	check that the model is a true reflection of the requirements in the requirements definition, including conformance to safety requirements
	check that the refinements during the design process preserve behaviour and properties	analyse the model to cast light on the requirements definition

Table 4.1: Verification and Validation with respect to Safety analysis and Models

of the safety analysis and the model in respect of its satisfaction of the safety requirements. Again, they are conducted separately: Table 4.1 indicates where verification and validation activities feature for the respective activities.

4.3.3 Motivation for an iterative approach to constructing fault trees

In this section we discuss the construction of fault trees and argue the need for an iterative approach – a novel method that is described as procedure **FTBuild** in the next section.

It is desirable that the overall process of constructing fault trees should not impose any restrictions which may prevent the discovery of hazards or faults. Accordingly, initially we do not impose any (formal) constraints on the relationships between events whilst constructing the tree (c.f the discussion of FMECA in [VGRH81]).

It is, however, useful to examine formally at intervals what one has done so far:

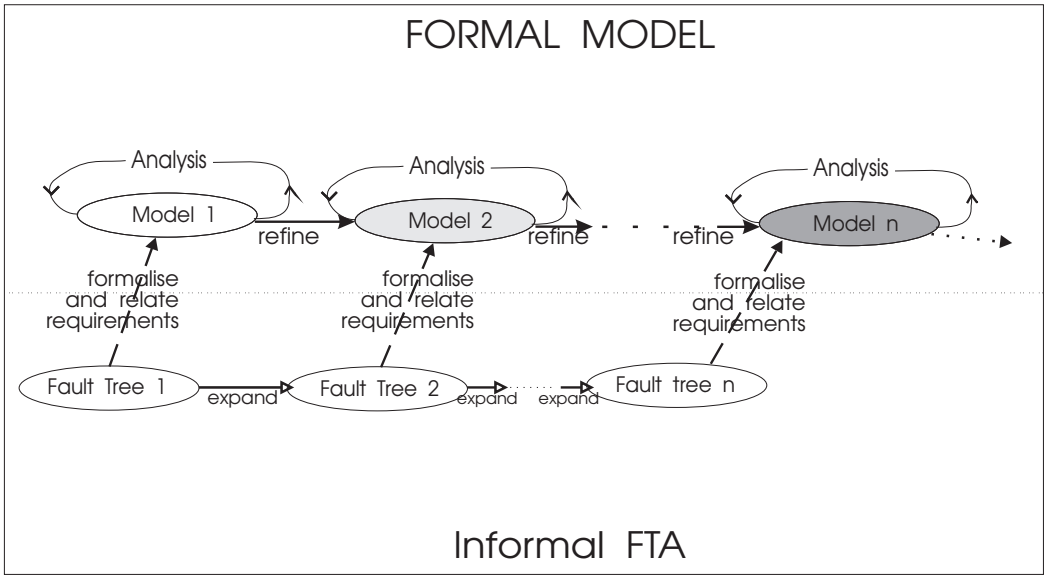

Figure 4.2: An incremental model for concurrent FTA and model refinement

the longer a fault tree is built up with no formal consideration, the greater is the potential weakness of events and their interactions being subject to ambiguity. Determining the appropriate semantics at just one gate can be a non-trivial matter, so if one extends the consideration to events not immediately related, then the situation is potentially much more complicated. Leaving the formal analysis to later may be uneconomical: an extreme case would be the generation of many complex fault trees, where the formal examination of the first one reveals some inherent weakness in the system conception, necessarily requiring its alteration and the modification of all the other fault trees.

In view of the potential pitfalls we suggest the construction of fault trees as an iterative procedure of constructing together informal and formalised trees, tied to the development of a model of a system or some part. This is illustrated by Figure 4.2, which is an instantiation of the generic incremental model illustrated in figure 2.3.

In this way, the formal analysis of the model plays a more integrated role in the system design. The procedure retains the freedom to explore faults informally, with no restrictions on the causes. However, once some particular gate condition is developed, it may be formalised, validated with respect to user requirements and these requirements further analysed in light of the formalisation, before continuing the brainstorming activity of searching further down the tree. The advantage of this method is that before a large tree is developed, it may be quickly realised that some faults are intractible (cannot be avoided or tolerated without adverse effects) and may thus cast light on some deficiency or

over-expectation in the design. (For instance, some part of the system might require more automation instead of a human operator). If this eventuality is reached, then the conceived system may be changed and the fault tree modified accordingly.

Note also that this procedure allows for a model to be developed separately from the fault tree analysis – one simply keeps skipping step 6 given below. It has been standard practice for models to have been built separately from such safety analysis, largely to meet prescribed functionality. Only then have they been validated *a posteriori* for safety-related properties, which in many cases have not been derived from any systematic safety analysis. In any case, if a model is fully developed without incorporating faults, it can be so complex that the late addition of faults may be difficult to treat, no matter how good the safety analysis. We have argued elsewhere that such a *post hoc* approach is inadequate for ensuring safety [TN96].

As indication of the generality of the method, we indicate how the work of Górski and Wardziński fits in the framework.

4.3.4 Procedure FTBuild for fault tree construction

This procedure **FTBuild** builds the fault tree in broad and unrestricted fashion, allowing its selective formal analysis both as a tree and in terms of safety requirements incorporated in one or more models.

Given an informal statement of user safety requirements, part of USER-REQS, together with a requirements definition SYS-REQS for the conceived system plus some model(s) of the system or part, **FTBuild** is defined as follows:

1. Select an informal fault (event) E.

2. Establish (informally) all [or some of] E's event causes – $E_1, E_2, ..., E_n$. If there are no causes then E is a leaf node and **FTBuild** is completed. This is perhaps the main brain-storming activity, really an exercise in HAZOP.

   Attempting to find all the causes is advised, to be in accordance with the 'Complete the Gate' Rule.

3. Formalise E – choose an event model and semantics; express this as some predicate and validate.

4. Consider the relationship between the events $E_1, E_2, ..., E_n$ as regards their causing the fault. (Note that not until all causes have been found will a relationship be complete).

5. Formalise the relationship as a gate condition, a predicate formulated in some logic, with semantics chosen accordingly (and validate).

6. (optional step)

   (a) formalise the whole tree and perform analysis (optional)

   - If the formalised fault tree implies some poor informal safety analysis (e.g. some inconsistency, as discussed in the next section), then this analysis needs to be re-addressed, whence the procedures involving all those parts of the tree's construction affected by the changes have to be terminated and re-invoked.

   (b) derive safety-related requirements for the given model based on the formalised gate conditions and events.

   (c) incorporate the safety requirements in the model and analyse both the suitability of the tree and to what extent the model takes account of the requirements – which we call *property conformance* (abbreviated henceforth as *conformance*) of the model. Assuming that the FTA has been validated formally, or is simply taken as OK, them the following scenarios are possible as a result of the analysis:

      i. the design appears okay, but the model is inadequate, so the model needs to be modified – the next step taken depends upon how the model is changed. (If the model cannot be modified in the desired manner, then it may be that the modelling language is not appropriate).

      ii. the model indicates that the conceived system is inadequate (so **FTBuild** may have to be abandoned until this is modified)

      iii. all is okay, so carry on with step 6d

   (d) (optional) Re-iterate step 6c for other model(s).

   These may be alternative models, perhaps prototypes, to be compared; or a refinement, which can provide information for the expansion of the tree.

7. For each E_i, perform **FTBuild** from step 2 and then END.

The benefit of this approach is that the fault tree is built up in a rigorous fashion. One may at any time halt the construction of the tree and concentrate on deriving requirements and/or mapping them to the model.

As indicated, reasoning about the model may lead to reconsideration of the system conception, without need to examine the fault tree further. This interactive approach shows

the need for the formal analysis as part of the requirements analysis for system design. Also, the analysis of the model in the light of the safety requirements may uncover new faults. Further contributions to the expansion of the fault tree may be expected as the model is refined towards implementation.

It is probably best to do the analysis of the fault tree in two stages: first, analyse the tree for its own sake (step 6a), not least because of computational concerns; next, analyse further when incorporating in the model (step 6c). The requirements for these are different: for instance, we may expect the formal examination of the safety analysis *per se* to treat all the events and gates found in the tree. This may mean evaluating the conjunction of predicates of events and gates, sometimes called the *characteristic predicate*, to check for consistency. This is likely to be not too heavy on resources. In contrast, we may not expect to perform so easily the same evaluation for a model, with a very much larger structure, so we would need to be selective here about the events and gates we focus on.

Work that has been undertaken in the CSDM can be seen to be contained within **FTBuild** as follows. An algorithm for the formalisation of an event (step 3) has been given in [GW95], steps 4– 6a have been treated in [BCG91, G94]. Some analysis of a formalised tree (step 6a) has been covered in [GMW95] and the derivation of safety requirements has been examined in [GW96].

4.3.5 Issues in the analysis of formalised fault trees

The analysis of formalised fault trees raises many issues, some general, others to be considered in the light of a specific tree developed as far as step 6a of **FTBuild** .

1. *consistency*: a tree has traditionally been defined inductively through gate conditions (a procedure we follow), which treat relations only between inputs and their immediate output. However, we require in addition to such local conditions the overall consistency of the system. A simple form of consistency relation has already been mentioned: just take the conjunction of the propositions corresponding to gate conditions. If such a predicate is always false then we regard that as an inconsistency. Such inconsistency may arise if, for example, one gate condition imposes certain constraints on the location of some hazardous object X, whilst another gate condition may impose other constraints which imply that X is in two places at the same time. In this case, the safety analyst should use, if available, past experience to clarify if and when such events can occur.

Thus the formalisation of just such informal conditions can be valuable in highlighting inconsistencies in the process of constructing an informal Fault Tree. One may use a partitioning scheme according to a particular viewpoint – e.g., using abstraction, one can examine both horizontal and vertical consistency in addition to notions of completeness defined in section 3.5.2.1.

2. *uniqueness*: for a given top-level fault or failure mode, how do we distinguish between possible fault trees? Given a tree containing events that are specifed as known plus token events, in those cases where there is lack of information about causes, is there some unique maximal tree?

3. *completeness*: to what extent can we say a tree is complete? Can we say that for some event, there are no further causes?

   Using the notation of the Fault Tree Handbook, we may say that a tree is *syntactically complete* if all its leaf nodes are primary events; it is *syntactically complete 'up to the level of information'* if instead, the leaf nodes may also be diamond events. In general, we cannot say whether we have a complete set of causes for a given event.

4. *abstraction*: some faults are visible to users in the environment, but others lie hidden, deep in some component. Handling all these together would be complex, so it is desirable to use abstraction to distinguish between (possibly classes of) events. If so, how? What are to be the bases for abstraction?

5. *computation of risk*: for propositional semantics risk is computed using minimal cutset forms but, as we have seen, this obscures durational aspects. Assuming a more realistic temporal semantics means that one can no longer in general apply this technique, as certain laws in Boolean Algebra no longer apply (see, e.g., CSDM examples in [BCG91]). In such cases one must determine alternative methods for calculating risk.

In [GMW95], formal analysis has been performed by mapping trees expressed in CSDM to corresponding representations in Time Petri Nets. In particular, reachability analysis is performed to show whether or not a hazard may arise from the given fault tree. It is reasonable to expect that a similar mapping could be performed to a language based on transition systems.

4.3.6 Generating safety requirements from fault trees

The procedure **FTBuild** generates in steps 2–5 events and gates as input to the safety requirements. Having formalised and analysed the fault tree, at step 6b we have to

formalise requirements for events and gates, which we view as a two stage process:

1. determine which events and gates are to be used in conditions
2. determine as maps with respect to the chosen events and gates the (nature of) the conditions which we require that the model should satisfy

For substep 2, we may choose as a mapping, one that simply maps events and gates to either themselves or to their negation, reflecting the following statement in [GW96] (part of STEP 4 in the Method Description):

> " ... the hazard can be prevented if, throughout the system operation, the software maintains the negation of the whole condition"

We may use the following simple decision mechanism to implement this:

> For those events (faults) and/or gates as specified which lie outside the control of the software being developed from our model, we stipulate that these conditions should be satisfied by the model as they stand (i.e. that the model is fault tolerant). For instance, such is the case for events of the kind, 'physical component wears out'.
>
> For those events and/or gate conditions which lie within control of the software being developed from our model, we stipulate that the negation of these conditions should be satisfied by the model. Such is the case for certain kinds of software elements: for instance, we would wish to have the negation of the event 'software routine Z engages in infinite loop'.

A more realistic mapping would take into account information about assumptions which can be made about those faults which lie outside the control of the model. Typically, certain other reliabilty measures can prevent some of the events that occur in a fault tree.

Another consideration for the requirements is distinguishing between local and global faults. If developing a component of a system, one should distinguish between faults of the overall system, which one is obliged to model if they impinge on the subsystem, and faults of the subsystem. In particular, a distinction should be made between software and hardware faults: some of the former faults are ones we expect to avoid, not tolerate.

We now elaborate the two phases below:

1. (a) Which events (faults and others) do we wish the model to take account of?
   (b) Which gates do we wish the model to take account of?

- Which events among those we've chosen above do we wish to be involved in the gate condition?
- type of gate condition? (may assume that gates may be classified into two types)
  - causal
  - generalisation

Whatever selection mechanism is used, we wish to ensure that requirements with respect to coverage of the tree are complete. We could indiscriminately go through each event and gate and derive requirements. However, it is more efficient to start with the leaf nodes. If we were to ensure the negation of each of these events (or just one from each 'AND' component) then that would ensure that the top-level hazard does not occur (removing a necessary cause removes an effect). In the case that we are not able to treat some of the hazards, we move up the tree and examine gate conditions involving higher level events.

If it is not possible to systematically address all events and gates, or if one wishes to differentiate between events, then one may apply some selection crieria as specified in section 4.4.1.

The denotation of requirements for the real-life situation may have several alternatives, all of which appear suitable candidates. Not until some formal analysis has been undertaken, can we discover which is most appropriate. Accordingly (for non-atomic events), we can establish a one-to-many correspondence between their informal denotation and a set of labels, each denoting a formalisation of the event. Unless stated otherwise, we take the convention of using upper case when we are referring to the real-life event, and using lower case when we are referring to a particular formalisation. Normally, we choose to consider just one formal representation at a time.

Similarly, it has already been shown that the semantics of a gate may need more than one formal representation before it is deemed suitable: an example is given in Section 5 of [GW95], where there is fine adjustment of the semantics of a causal gate.

2. (a) For each event selected, we determine event requirements for the model. These are generally logical formulae which explicitly mention some denotation of the event. Sometimes, however, it may be easier to formulate some indirect requirement, using some other methodology.

Thus the details of the requirements will depend on the formalism used; in temporal logic for instance, given an event E, formalised as e, if we are using the modal mu-calculus, we may define a function of e to represent the statement "E is to occur eventually"; or, if we use of a logic with explicit time, such as CSDM, we could specify that "E is to occur within t_E seconds".

Let $Events$ denote the set of events e in the tree, where $Events \subseteq \Phi$ with Φ denoting the set of formulae of some chosen logic. We specify requirements using a mapping $\psi : Events \rightarrow \Phi$. If we use the simple specialisation above, we have for any e, either $\psi(e) = e$ or $\psi(e) = \neg e$.

(b) For the gates selected as above, apply a procedure similar to that for events, though here we must pay additional attention to the semantics chosen. We define an enumerated *semantics type* whose instantiation determines how we choose to interpret the relations between events at a gate.

- **Gate semantics**

  Let Ω denote the set of logical connective such as 'AND', 'OR', ... etc which correspond to the gate types in the informal tree (perhaps as given in [VGRH81]). Let $Events$ denote the set of a events in a tree. Then the set $Gates$ of gates may be denoted as $Gates = \Omega \times Events^*$. Thus, a gate g may be given as a tuple $(\omega, out, in_1, ..., in_n)$, where $\omega \in \Omega$, out denotes the output event, and the in_i, $(i = 1, ..., n)$ denote input events, with all events belonging to $Events$. Each such event is a proposition belonging to Φ.

  Let Λ denote the set of semantics types for gates (covering generalisation and causal types etc.) Then we may define a function $[[\ ]] : Gates \times \Lambda \rightarrow \Phi$, which we call the *semantic function*. In the case that n is made explicit, $[[\ ]]$ may be regarded as a function which maps from $\Omega \times Events^{n+1} \times \Lambda$ to Φ. For convenience, we write $[[g]]_\lambda$ for $[[\ ]](g, \lambda)$. (In practice, the domain for events may be a strict subset of Φ.)

  In determining the semantics of a tree, we restrict the domain to $\mathcal{G} \times \Lambda$, where $\mathcal{G} \subseteq Gates$ denotes the set of gates in a tree.

  We allow for the case when λ is understood or otherwise not specified, where we write $[[g]]$ to denote the semantics of a gate g.

- **Requirements relation**

  We stipulate that requirements for gates should themselves be gate conditions. As befitting standard practice with other requirements, we regard

these as being originally conceived informally: even though we have at our disposal formalised objects, deciding what we require for the software may need some informal reasoning. Hence, the definition of a requirements function for gates is in two parts: a mapping that establishes one or more requirements for each gate, and the application of the semantics function above that determines a unique interpretation for each image of this mapping.

A *requirements relation* is any mapping σ such that $\sigma : \mathcal{G} \rightarrow Gates$. For a gate g, we say that $\sigma(g)$ denotes the *requirements of* g. A *requirements function* is the application of the semantics function [[ ]] to σ. Hence, it has signature $\mathcal{G} \times \Lambda \rightarrow \Phi$.

We have allowed the requirements derived from gate conditions to be very general. In particular, it allows complete freedom to choose which events are involved in requirements for gates – some requirements might not even mention any of the events of g. We have indicated earlier how formalisation of trees is useful and adviseable to reduce ambiguity. It is arguable then that a formalisation of FTA is rendered much weakened by such great flexibility. However this can be justified when considering that the specification of safety requirements is a design decision, and as such is a creative process. Also, this procedure does encourage the use of the given formulae as components in generating requirements, so the formalisations can be retained directly. These procedures have been designed to be general, whilst allowing for stricter domain-specific measures where deemed appropriate, so they should be in a form that leads to validation with confidence.

As an example to support this contention, suppose that we have the gate condition $g(AND, e, e_1, e_2)$ where e denotes 'Error in Patient prescription' with e_1 denoting 'data corruption' in some communications layer. In determining the fault tree semantics, we may choose a temporal interpretation that states that input events e_1 and e_2 imply that eventually output event e occurs. However, from the requirements analysis we may wish that the requirements relation σ for this gate specifies that if both input events occur, then some method of control e_c, say, is invoked immediately, a condition that has no reference to e and has semantics different from g. Such a very general relation gives much scope to the requirements specifier. Hence, there is a special need for care. In many cases, we may wish to preserve semantics as would be the case if we were to use the simple specialisation above, where we may define that for any g, either $\sigma(g) = g$ or $\sigma(g) = \neg g$.

4.3.6.1 Example of Gate Semantics and Requirements Derivation

We revisit the example of the robot on the assembly line, whose fault tree is given in Figure reffig:ftarobot, to illustrate some particular aspects of formalisation within the procedure FTBuild.

For this example, an instantiation of FTBuild may proceed as follows:

1. Select E to be 'robot arm swings to angle beyond normal range'.

2. We determine the causes of E to be:

E_1: "Control initiates instruction to swing arm round"

E_2: "Control cannot make arm stop swinging"

3. Simply denote E by e as its meaning can be rewritten in terms of its inputs. Similarly denote E_1 and E_2 by e_1 and e_2 respectively.

4,5. We now form a gate G_1. At this stage the Fault Tree Handbook would insist that E_1 and E_2 are necessary, sufficient and immediate causes, but, as already argued in section 4.2, this is restrictive. Our formalisation for gate semantics allows much greater flexibility.

For instance, suppose $\Omega = \{'OR', 'AND'\}$. Then we may choose that the set of semantic types Λ contains as a start λ_1 and λ_2 to denote generalisation 'OR' and generalisation 'AND':

$$[[('OR', out, in_1, in_2, ..., in_n)]]_{\lambda_1} \stackrel{def}{=} out \iff in_1 \vee in_2 \vee ... \vee in_n$$

and

$$[[('AND', out, in_1, in_2, ..., in_n)]]_{\lambda_2} \stackrel{def}{=} out \iff in_1 \vee in_2 \vee ... \vee in_n$$

However, many other kinds of relations can be formulated, especially causal ones. Thus we define a causal 'AND':

$$[[('AND', out, in_1, in_2, ..., in_n)]]_{\lambda_3} \stackrel{def}{=} out \iff [in_1]even(in_1 \wedge in_2 \wedge ... \wedge in_n)$$

where for some formula ϕ, $even(\phi)$ is a formula in temporal that represents the property "eventually ϕ holds".

We apply λ_3 here and stipulate that G_1 should be given by:

$$[[('AND', e, e_1, e_2)]]_{\lambda_3} \stackrel{def}{=} e \iff [e_1]even(e_2)$$

We take 6b) as the next step and just consider the derivation of a safety requirement generated from G_1. For one requirement we may stipulate a protocol feature that ensures that at least we can ascertain the connection is up. This is achieved by requiring:

> After the control sends an instruction to the robot, an acknowledgement is received from the robot" (which we denote as the event e_c).

Such a requirement requires the definition of several other events: the issuing of any command by E_i may be denoted *control.send.instr*, for the sending of the acknowledgement, we refer to E_a and choose a corresponding denotation to be *rob.send.ack*; for the receipt of the acknowlegement, E_b, we use the denotation, *control.rec.ack*.

For the requirement, we can then formulate the requirement as:

$$[[\sigma_1(g_1)]]_{\lambda_3} \stackrel{def}{=} e_c <=> [control.send.instr]even(rob.send.ack \wedge control.rec.ack)$$

In words, this means that once an instruction of any kind is sent to the robot, eventually the robot sends an acknowledgement and the control receives the acknowledgement.

Further requirements $\sigma_1, \sigma_2, ...$ may be derived in a similar manner.

4.3.7 Evaluating safety requirements

Once the requirements have been generated, they need to be evaluated in models – step 6c which consists of determining whether or not, or to what extent, the requirements are satisfied – as the evaluation of some general formula relating the safety requirements and the model. We stipulate that the formula is in terms of predicates, which may themselves be of any kind.

We give below an algorithm that performs this task.

4.3.7.1 An algorithm for evaluating a predicate for a particular safety requirement

A tree and its associated safety requirements may represent simultaneously any fault in great detail from various perspectives of viewpoints, for instance several levels of abstraction. In contrast, the corresponding representation in a model is more restricted – typically choices have to be made between levels of abstraction for a given fault.

For a model to be valid, requires therefore some means for events and gates to be evaluated to take this into account. We give here an algorithm **FTEVAL** which evaluates a predicate for a safety requirement of an event or gate. The algorithm is applicable to any system on which predicates may be evaluated, including labelled transition systems. Recall that for gate requirements, the events defined may or may not co-incide with those in a fault tree.

Notation Let $\mathcal{E}$ be a set of events and let $\mathcal{G}$ be a set of gates. Let $\mathcal{G}'$ denote the set of gates which constitute the image of $\mathcal{G}$ under σ. Let $\mathcal{C}$ denote the context of evalutation, being the evaluation according to the set $\{\mathcal{E}, \mathcal{G}, \mathcal{G}'\}$. Let $G(e; e_1, ..., e_n)$ denote a gate with output e and inputs $e_1, ..., e_n$. For distinct propositions $Q_1, ..., Q_n$, let $\phi[\phi_1/Q_1, ..., \phi_n/Q_n]$ be the formula ϕ with occurrences of $Q_1, ..., Q_n$ in ϕ replaced simultaneously by $\phi_1, ..., \phi_n$. Note that

FTEVAL may then be given as:

- Let ψ be a requirement relation on events and let $\psi_{\mathcal{C}}$ denote its evaluation under the context $\mathcal{C}$. Then for an event $e \in \mathcal{E}$, $\psi_{\mathcal{C}}(e)$ is evaluated as the predicate $(\psi(e))[EVENT(e)/e]$, where $EVENT(e)$ is defined as:

  e IF e is a leaf node or an output to a causal gate.

  ELSE

  $(\bigvee_{j \in \mathcal{J}} EVENT(e_j)$ IF e is an output to a gate of type Generalisation-OR,

  $$g(OR, e, e_1, ..., e_n) \in \mathcal{G}'$$

  ELSE

  $\bigwedge_{j \in \mathcal{J}} EVENT(e_j)$, IF e is an output to a gate of type Generalisation-AND,

  $$g(AND, e, e_1, ..., e_n) \in \mathcal{G}'$$

  ),

  where $\mathcal{J}$ is the index set for the set of inputs for output e in the generalisation gate restricted by $\mathcal{E}$, i.e. where $\mathcal{J} = \{i | e_i \in \mathcal{E}\}$.

- Let σ be a requirements relation on gates and let $\sigma_{\mathcal{C}}$ denote its evaluation under the context $\mathcal{C}$. For a gate $g \in \mathcal{G}$, suppose the requirements relation is: $\sigma_{\mathcal{C}}(g) = G(e; e_1, ..., e_n) \in \mathcal{G}'$. Then $\sigma_{\mathcal{C}}(g)$ is evaluated as

$$[[\sigma(g)]][EVENT(e)/e; EVENT(e_1)/e_1, ..., EVENT(e_n)/e_n] \qquad (4.1)$$

Provided predicates may be evaluated in a finite number of steps, then this algorithm terminates for finite trees. Further, if ψ and σ are defined for all events and gates, then this algorithm terminates with a value. One may view the algorithm as defining rewrite rules; and this could be the impetus for defining new semantics for 'generalisation', especially for gates.

The rationale behind this definition of evaluation is that we wish to allow that an event evaluates to `true` either by itself or, in the case of generalisation, through its inputs. Similarly for gates, we wish that a gate condition holds if the relationship holds between an output and any input event as evaluated above.

We specify that the gate conditions should be in terms of events as they stand rather than in terms of conditions on the maps defined by ψ to allow two levels of requirements. First, we may prescribe that $\psi(e)$ holds in order to prevent the occurrence of event e. But what if e does happen (i.e. $\psi(e)$ doesn't hold)? Then we may use the gate conditions defined by σ to prevent it causing problems. In this way, even if a model has undesired faults, recovery mechanisms may be stipulated.

4.4 Defining relations between models and fault trees

In this section we aim to provide relations which reflect practical needs regarding incorporating requirements in the model. A model may be checked for conformance to the set of requirements derived using the procedures 2a) and 2b) in section 4.3.6. Conformance may be defined (and evaluated) according to various levels of strength through the definition of one or more *conformance relations* which prescribe essentially the breadth (or completeness) of requirements coverage. A second degree of flexibility is provided by the introduction of a weaker notion, *consistency*, which, in addition, takes account of the fact that the model may be relatively embryonic in its refinement; and similarly *consistency relations* may be defined.

Bruns and Anderson have laid the semantic groundwork for relating fault trees and models in [BA93]. This has included the provision of three relations between fault trees and models which are termed 'consistency' relations. Note that in our work we term them more specifically conformance relations (such that conformance implies consistency). However, the paper being just a start, has the following drawbacks.

First, no requirements stage is made explicit – the relations that are given simply insist that models satisfy gate conditions without modification. We have addressed this gap in sections 4.3 and in terms of the algorithm defined there, this omission means that ψ and σ are defined to be the identity mapping in all cases.

Second, as two of the relations are very strong conditions and the other a minimal condition, they are unlikely to be much use for most models in practice. In general, a given model is unlikely to satisfy all the conditions in a tree, so we need to have the option of being selective about parts of the tree we wish to examine. Yet, we would like as much as the whole tree to be consistent in some way with the given model. This requires some leeway on the model itself in terms of how it may be transformed to yield the required conformance.

We attempt to address the second issue by providing first of all some criteria that enable us to select those parts of the tree that contribute to the requirements. This background helps to motivate the subsequent generalisation of the results that have been previously derived, gradually building up towards the definition of several relations that are more readily applicable, taking account of the criteria. We concentrate initially on gate requirements only, and then we expand to incorporate event requirements. We start by introducing a little terminology (from [BA93]) which we first use in the definition of a tree, on which we may conduct various analyses.

4.4.1 Establishing criteria for relations between fault trees and models

In this section we provide some criteria for the selection of events and gates to be used in requirements and for the requirements themselves. Underlying the selection of requirements is the need to ensure safety which will have to be determined qualitatively and quantitatively, covering severity and likelihood of occurrence. Assuming that there has been more than one fault tree constructed, a further important issue is the selection of top-level faults which one chooses that the model should incorporate.

Some criteria and related issues for a given relation are:

- *criticality of fault*: ideally, we would like for each event both its severity and probability of occurrence. Although, for a given fault tree, it is the probability of a sequence or combination of events that may happen which is the prime factor, it should be noted that some events may cause faults in many other trees. In practice, to provide some completeness, the severity of any fault would require reliabilty analysis plus various inductive analyses such as FMECA to determine a fault's consequences. Estimates

have to be made where information is incomplete.

- *levels of abstraction in fault tree*: how much of the tree should be abstracted out at what stage in the refinement? Although fault tree analysis is nominally 'top-down', the causal nature of hazard sequences allows that faults have propogated not only from lower levels, but also from higher levels, especially via interfaces. For example, consider an 'Out of Range' error on a prescription that is indicated on the display of a bedside medical device that is part of a network of devices controlled remotely by a central console. FTA may reveal that the error can have a fault cause in the 'low level' physical layer, which itself may relate to some 'high level' environmental factors such as location of objects (c.f. robot example in section 4.2). In this instance, the 'high level' fault has merely been propogated in transparent fashion through reliable channels – higher layers in the communications system. This kind of error also prompts FMECA since corrupted data supplied to the applications software may initiate further faults.

- *level of abstraction in model*: a system fault may be represented at various levels of abstraction – how do we match this up with the model? If a model is very abstract it will model few faults and the search down the fault tree may be only shallow and/or include few primary events. Alternatively, at a low level of abstraction, the model may cover a lot of leaf nodes which are disconnected.

4.4.2 A common semantics for fault trees and models

We use *labelled transition systems* as the semantic model underlying both fault trees and models/specifications defined in section 2.5.1.

In seeking to establish that such systems satisfy certain properties, we introduce for a transition system $\mathcal{T}$ valuations, through a map $\mathcal{V}$ that maps variables to set of states. A formula ϕ is then interpreted as the set $||\phi||^{\mathcal{T}}_{\mathcal{V}}$ of states, which is defined inductively on the structure of the formulae through a set of rules. A model $\mathcal{M}$ may then be defined as the pair $(\mathcal{T}, \mathcal{V})$; and a state s satisfies a formula relative to a given $\mathcal{M}$, written $s \models ||\phi||^{\mathcal{T}}_{\mathcal{V}}$ if $s \in ||\phi||^{\mathcal{T}}_{\mathcal{V}}$. In the case that we are applying the formulae to a model $\mathcal{M}$ in its initial state s_0, i.e. where $s_0 \models_{\mathcal{M}} \phi$, we call this a *rooted transition system.*

We add one more notation following [BA93]: for a property ϕ to hold for all states, we use the **always** temporal operator defined by:

$$\mathbf{always}(\phi) \stackrel{def}{=} \nu Z.\phi \wedge [-]Z \tag{4.2}$$

The fault tree's meaning is in terms of its gates which are predicates in some

temporal logic. As we are to interpret these formulae over transition systems, we require that they hold for all states of the transition systems. Hence, the semantics of the tree is defined as the conjunction of all gate conditions:

$$[[t]] \stackrel{def}{=} \textbf{always}(\bigwedge_{g \in gates(t)} [[g]]) \tag{4.3}$$

where t is a tree and $gates(t)$ denotes the set of gates of a tree.

We can be explicit about gate semantics by replacing the $[[g]]$ term by $[[g]]_\lambda$, where $\lambda \in \Lambda$ (as defined in section 4.3.6).

4.4.3 General conformance relations

Conformance relations between requirements and models are also evaluated as a conjunction of predicates, but here the predicates are more general, as allowed in the definition of σ. In particular, in deriving our requirements, we may not wish to insist on the use of the **always** operator, so we do not include it outside the conjunction. (Thus part of the requirements stage involves deciding the temporal scope of the requirements.)

The first relation allows the user to specify which (top-level) faults and any selection of gates of the fault tree. We use the following notation: Let S denote the safety-related system, $ft(F)$ denote the set of fault trees of F. Then in choosing various subsets of $ft(F)$, for coverage it suffices to select the *maximal tree*, $t_{max}(F)$ (where $\forall t \in ft(F).t \sqsubseteq t_{max}(F)$, with $\sqsubseteq$ denoting a subtree relation) and choose subsets of that. A function δ is defined to give any selection of those trees that are subtrees of $t_{max}(F)$.

Definition 4.4.1 *General Conformance Relation I*

Let $\mathcal{F}' \subseteq \mathcal{F}$, where $\mathcal{F}$ is the set of faults for S. For any fault $F \in \mathcal{F}'$, let $\rho(t) = \{\tau | \tau \sqsubseteq t\}$ and let $\delta(t) \subseteq \rho(t)$. Also, let ζ be a function mapping fault trees to sets of gates, such that for any fault tree Γ, $\zeta(\Gamma) \subseteq gates(\Gamma)$. Then we say that a model $\mathcal{M}$ conforms to S with respect to the requirements defined by σ applied to $\mathcal{F}', \delta$ and ζ if:

$$s_0 \models_{\mathcal{M}} \bigwedge_{F \in \mathcal{F}'} \bigwedge_{\Gamma \in \delta(t_{max}(F))} \bigwedge_{g \in \zeta(\Gamma)} [[\sigma(g)]]$$

We write $M\textbf{conf}\ S(\mathcal{F}', \delta, \zeta)$.

The requirements analysis may typically put forward:

- choose $\mathcal{F}'$ to be the (top-level) faults considered most important by the safety analysts, perhaps according to *criticality of fault*.
- choose δ to pick out those fault trees which are of interest to us. (E.g., referring to Figure 4.1, F could be 'robot arm swings to angle beyond normal range' and if we wanted to concentrate on the communications system, $\delta(T_{max}(F))$ could be accordingly a selection of subtrees that have root node 'communication fault occurs'.
- choose ζ to pick out those events in a given fault tree that we wish the model to address. Here we may use the criterion of *abstraction*. Such events may or may not be adjacent to each other in the tree (see the note on abstraction and interfaces).
- choose σ according to how we wish the relationships between events to be reflected in the requirements for the model.

We now define example relations. The predicates themselves may be checked independently of the model to determine the consistency of the tree. When evaluating the predicates on the model, they reflect a variety of specific requirements. The first two show that this definition generalises the first two notions of consistency given in [BA93]:

1. For all gates g, put $\sigma(g) = \mathbf{always}(g)$, $\mathcal{F}' = \mathcal{F}$, $\delta(t_{max}(F)) = ft(F)$ and $\zeta(\Gamma) = gates(\Gamma)$.

   (i.e. when evaluated with respect to a model M, M is consistent with the fault tree if and only if it satisfies all the gate conditions in the tree.)

2. For all gates g, put $\sigma(g) = \mathbf{always}(g)$, $\mathcal{F}' = \mathcal{F} \cap faults(\mathcal{M})$, $\delta(t_{max}(F)) = ft(F)$ and $\zeta(\Gamma) = gates(\Gamma)$.

   This is a kind of minimal condition which says: "M is consistent with the fault tree iff it satisfies all the gate conditions that involve faults of the tree that are in the model". Thus, e.g., if a model has one fault in common with a fault tree, but none of the tree's events to cause the fault, then this model will satisfy the minimality criterion.

4.4.4 Further generalisation of conformance

We provide a further generalisation of the above to the level of event in order to allow for truncated gate conditions where either information about some faults is not available or we deliberately choose to ignore some faults. We transfer notions in [BA93],

where an interpretation of a *tree* is given in the absence of events, to the interpretation of a *gate* in the absence of events.

For a gate g, let $events(g)$ denote the set of events specified in g. Let $Bool$ be the set $\{true, false\}$. Then the interpretation of a gate $g \in Gates$, in the absence of a set of events $\varepsilon = \{a_1, a_2, ..., a_n\} \subseteq events(g)$ is defined to be

$$[[g - \varepsilon]] \stackrel{def}{=} \bigvee_{(b_1,...,b_n) \in Bool^n} [[g]][b_1/a_1, \ldots, b_n/a_n] \tag{4.4}$$

Note that there is redundancy in this definition: if g is an 'AND'-gate, we can have the same meaning as above simply by putting $(b_1, \ldots, b_n) = True^n$.

Let ϵ be a function that maps from gates to events specified in the gate such that $\epsilon(g) \subseteq events(g)$. Now we define a second generalised conformance relation which specifies conformance in terms of a set of gate conditions restricted according to the selection of fault trees, plus a set of event conditions, restricted according to the selection of gates.

Definition 4.4.2 *Conformance II*

*A model $\mathcal{M}$ conforms to S with respect to the requirements defined by ψ and σ applied to $\mathcal{F}', \delta, \zeta$ and ϵ (write M***conf** *$S(\mathcal{F}', \delta, \zeta, \epsilon)$) if:*

$$s_0 \models_{\mathcal{M}} \bigwedge_{F \in \mathcal{F}'} \bigwedge_{\Gamma \in \delta(ft(F))} \bigwedge_{g \in \zeta(\Gamma)} \left([[\sigma(g) - \epsilon(g)]] \bigwedge_{e \in g} \psi(e) \right)$$

Putting $\epsilon(g) = \emptyset$ and $\psi(x) =$ `true` for all events x gives rise to the definition 4.1.

This definition leaves open whether or not we distinguish between the same event occurring in different parts of the tree. This is worthwhile if we wish, say, to analyse just a single gate. In [BA93] there is no such distinction made, since the consistency relation is defined on events over the whole tree.

Suppose t is a tree. Let $\eta \in events(t)$ be a set of events in a tree T. Suppose ε_F is a set of events to be ignored for tree F. Now the third consistency relation of Bruns and Anderson is subsumed by putting:

$\mathcal{F}' = \mathcal{F} \cap faults(M)$, $\delta(t_{max}(F)) = ft(F)$, $\zeta(\Gamma) = gates(\Gamma)$, for all gates g, $\sigma(g) =$ **always**(g) and $E \in \epsilon(g)$ iff $E \in \varepsilon_F$.

4.4.5 Consistency relations for models undergoing refinement

In this section, we consider briefly how greater flexibility may be introduced to take account of the fact that the requirements and the model may be at first glance far

apart. In Chapter 3, section 3.6.2.2, the notion of consistency between specifications (or models) that has been defined by Bowman *et al* was introduced into the CM framework. That notion defines specifications to be consistent with each other if each can be refined in such a way that they can merge into the same implementation. Here we treat the complementary problem of consistency between models and requirements. Unlike the definition for consistency between models, the notion introduced below keeps one side fixed.

Let us refer again to the incremental model given in figure 4.2. So far we have in effect concentrated on developing relations which match side by side fault trees and models, as indicated explicitly by the 'mapping' arrows. A more flexible view is to realise that any object can implicitly be related to any image. Thus we can define relations for a model that is relatively behind, or level with or in advance of a set of requirements.

In particular, the conformance relations above insist that requirements are satisfied for every event and gate and thus are aimed at models at the same level as the requirements. This can be restrictive: consider the case where we are just starting to build a model using stepwise refinement. It is likely that the initial versions of this model will conform to only some of the specified requirements. Now consider the case where a model is relatively more developed than the set of requirements. There we expect that certain simplifications of the model would be necessary before a model could satisfy all these requirements.

Another series of relations may then be defined, all placed in the context of refinement. We consider a *trajectory* (or refinement path) to be a sequence of models, whose initial member is the most abstract model, with all other models being some refinement of the previous one. In our general definition we do not specify what we regard as refinement – that can be made explicit when the modelling context is chosen (e.g process algebra).

Thus, we may consider relations in which a model belongs to some trajectory in which one of the models does conform. We call such relations *consistency relations*, with consistency defined in an existential manner. Then we define:

Definition 4.4.3 *General Consistency Relation*

A model M is 'consistent' with the requirements defined by ψ and σ applied to $\mathcal{F}'$, δ, ζ and ϵ (write M **cons** *$S(\mathcal{F}', \delta, \zeta, \epsilon)$) if it belongs to a refinement trajectory $\mathcal{T}$ containing some model M' such that M'* **conf** *$S(\mathcal{F}', \delta, \zeta, \epsilon)$.*

We may be more explicit by following the definition of consistency in the previous chapter: let $\mathcal{SPEC}$ denote the set of all models (or specifications) and let $\mathbf{ref} \subseteq \mathcal{SPEC} \times \mathcal{SPEC}$ denote a refinement relation. Then the statement "belongs to a refinement trajectory $\mathcal{T}$ containing some model M'" may be replaced by "$\exists M' \in \mathcal{SPEC}$ such that $M \mathbf{ref} M'$ or

$M'\mathbf{ref}M$)". That is to say, given a set of requirements, Req, then there exists a set $\mathcal{M}_{Req}$ of models which conform to the requirements such that: if M is a model that is at a relatively early stage in development, then we expect to refine it towards some M' belonging to $\mathcal{M}_{Req}$. On the other hand, if M is at a relatively late stage in development, then we expect to show that it has been refined from some such M'. The definition can be specialised if, say, M is always intended to be a refinement of M'. Then M is "consistent" with respect to Req if and only if $MrefM'$.

The problem of existence, if tractible, is probably solvable in most cases by running some algorithm to actually determine in some exploratory and computationally expensive manner a model that conforms. This approach is thus unlikely to be very practical. Further, such an approach does not respect the essentially creative aspect of the design process involved in building a model

A weaker approach to consistency is to see if one may simplify the requirements using one set of operations or the model using another set of operations so that there is conformance at some specified level. The kind of simplifying operations that may be defined will depend upon the languages used for formalising the fault tree and the model respectively.

The task of simplifying the requirements consists of redefining conditions in terms of a reduced sets of events and gates, and the way we reduce the selection depends upon the structure of the tree. Suppose we know which events ε_M and gates $\mathcal{G}_M$ the model is supposed to incorporate thus far. Let ε_S and $\mathcal{G}_S$ denote those events and gates for which safety requirements have been established. Then one can check for consistency by testing conformance on the requirements of $\varepsilon_M \cap \varepsilon_S$ and $\mathcal{G}_M \cap \mathcal{G}_S$. This is in the spirit of the second consistency relation of [BA93].

However, if one is not sure of the events and gates treated by the model (which may especially be the case if the safety analysis and modelling are conducted by two separate people or teams), then we need some other approach. A very general method is to treat this not as a 2 valued decision problem which reduces to terms of yes/no for a specific conformance, but rather a function which gives a status of the model, telling us to what extent a model conforms (or is consistent). Ideally, we'd like this deductive process to determine the largest subset of requirements to which the model conforms. This may be achieved by appropriate selection of simplification operations on the requirements.

4.5 Conclusions

We have provided in this chapter a methodology for generating requirements for formal models from making appropriate use of fault tree analysis, one of the traditional safety analysis techniques. There are two main pillars to this: a procedure which spans formal fault tree analysis through to requirements derivation and incorporation into a model; and the extension of existing theory to support the validation of requirements. The work has shown, amongst other things, that safety-related properties really can and do have a system basis. Also, as an indication of its viability, many of the usual theoretical issues in formal methods are raised quite naturally.

Even for simple examples, it is evident that determining where a model has failed a requirement is not always straightforward. In order to be able to develop a fault tree in a helpful manner, with penetrating hazard analysis, it becomes important to have a well-structured specification in order to isolate causes of problems. The success of the procedure is also largely dependent upon the proof techniques for verification and validation – conformance relations fail to hold where proofs cannot be shown using a given set of computational resources.

As this chapter is principally concerned with establishing relations between fault trees and models, issues surrounding the safety requirements analysis irrespective of models were not developed further. However this would be useful: in particular, it would be interesting to map fault trees to a transition system model for validation, analagous to the process that has been carried out to Petri Nets. Assuming that this can be done, then state reachability analysis may be performed in one or more of a number of tools available. For complex trees, many of the tools support techniques for simplifying the system, whilst preserving properties, techniques which have been commonly applied to the analysis of system models. One might also wish to perform 'what-if' analysis by altering some of the event or gate conditions to supply information towards appropriate methods of control.

Chapter 5

A Theory of Robust Conformance Testing

5.1 Introduction

A standard technique for contributing towards the assurance of integrity for safety critical systems is the use of testing. In industrial practice, testing procedures have become well developed through experience. They can have distinct roles: verifying that an implementation conforms to a specification or validation with respect to user requirements.

For safety, a key issue is *robustness*, the ability for a system to operate dependably in all operational circumstances. It may be possible to determine this through exhaustive testing so that all eventualities are accounted for, but generally this is not the case since testing is a time consuming activity. However, selected aspects may be more amenable. Perhaps the most penetrating aspect of testing is its ability to target certain modes of operation to uncover faults, which can certainly contribute to the ascertaining of robustness or otherwise.

In this chapter we start with a general introduction to testing, illustrating some of the ideas in an informal manner. We then proceed to treat testing in a formal software context, whilst retaining an engineering-style perspective. This view acts as a backdrop to the context of the specific Formal Description Technique (FDT), LOTOS, so that testing in the formal context is also related to testing in the physical world. On the other hand, there is also some discussion as to how testing in the formal setting can serve as a basis for a testing strategy for physical realizations.

The foundations of the chapter are: a testing framework due to Brinksma, Tretmans et al., a notion of what a test or experiment actually is, due to Hennessy and De Nicola, notions of conformance and robustness, as characterised by Brinksma, and the careful consideration of their relation to testing relations of Hennessy and De Nicola, culminating in

a proof that the reduction preorder may be characterised in terms of *may* and *must* tests (defined later).

After this backbone has been established, the main investigation is conducted. This concerns the derivation for a restricted class of LOTOS behaviour expressions of a proposed single (*canonical*) tester for the *reduction* pre-order (a well-established relation). This is designed in such a way as to allow a method for implementing it as a LOTOS process definition. Further, this tester is *unified* in the sense that it is designed to test simultaneously for robustness and conformance, in contrast to, e.g., [BAL+89], where testing for each is done separately.

This chapter assumes some knowledge of LOTOS, for which a brief introduction was given in chapter 2 and a proper tutorial given in [BB87].

5.2 Background to Testing in the Formal Context

In this section we consider standard engineering views on testing and describe the formal analogue, contrasting the two as regards environments for testing.

Testing consists simply of supplying certain input(s) into a system during its execution and observing the response of the system - the *outcome*. This is normally with the intention of comparing the response with some given expectation in such a way that some definite statement may be made about the system. Test outcomes are thus assigned valuations or *verdicts* to indicate this. Typically, there may be three verdicts – 'success' (or 'pass'); 'failure'; and 'inconclusive'. In such a scheme, either of the first two occur when there is some output from which one can deduce some property of the system, whilst the last occurs when either there is no output or the given output is insufficient to determine an aspect of a system's behaviour.

The notion of testing as an input–output function is standard for programming and physical devices which both receive input from and pass output to an external environment (such as a human operator) via some interface. However, formal *modelling* languages such as process algebras are not programming languages and do not 'compute' functions in this way. Nevertheless, even though 'input' does not come directly from an external environment, we can certainly simulate this through the *modelling* of tests. In this context we refer to testing *implementations* with respect to certain properties we wish them to satisfy. We refer to physical systems that are to satisfy such properties as *realizations*.

Many railway stations have automatic ticket vending machines which accept coins and notes and issue change. Suppose you test a prototype machine by buying on several

different occasions a ticket for 70 pence by feeding in a pound coin. The tester may know nothing about the internal behaviour of the machine, so it is quite possible that the change given will be, on one occasion, 3 10p's, on another 10p and 20p, and on yet another nothing (because there's no change left)! This is an instance of *non-determinism*.

The execution of the test may yield either *deterministic* or non-deterministic behaviour. In the former, when a test is applied the response of the system will always be the same. For a non-deterministic system, at least one of the actions performed will be completely up to the system being tested, lying outside the control of the tester. In this case, the response may vary: applying a test may yield some satisfactory behaviour on one occasion, but deadlock on another.

Testing depends upon taking a viewpoint. Here we treat the system under test as a 'black box' whose responses to tests may be observed only externally. Responses may then be characterised as having two aspects: sequences that are accepted (information collated 'off-line' as it were) and responsiveness at a particular instance to extending the test ('on-line' information). Hence, the system's behaviour can be described completely in terms of the response to tests of two types – acceptance tests and rejection tests. An *acceptance test* seeks to determine if the system accepts a given set of interactions with the environment. A *rejection test* seeks to determine if the system rejects a set of events in a given state.

Testing may be made comprehensive through the use of a suitably complete set (or *suite*) of tests. Further, it may be possible to design a single test to have the same power to discriminate as a test suite – as is the case for canonical testers.

Formal methods provide the following benefits as regards the use of testing:

- The use of formal methods prompts the careful planning of a testing strategy for the realization, well *in advance*.
- Formal testing can reveal important information about subtle aspects of the behaviour of system being tested.
- Further (and hence), the analysis of formalised tests contributes information towards the *kinds of tests* that should be carried out.

For example, the use of formal methods can be very useful in planning the verification of the correct installation and performance of hardware components and their configurations on which the embedded software depends.

5.2.1 Testing as an alternative validation and verification activity

Some observations can be made as regards the use of testing as an alternative verification and validation activity to other methods. Regarding the formal setting, testing has a number of advantages over other traditional verification techniques, which check properties of implementations:

- Testing allows analysis of an implementation without the tester having to know its structure; though, in current practice, most automated tools require some representation of the Implementation Under Test (IUT) in order to subsequently check testing-based relations.

- Consider the task of realizing a formal model as a physical implementation. Some non-testing based relations for the models cannot be applied analogously to physical entities, so these cannot help in verifying that the behaviour of the physical implementation meets its specification. However, one can use testing strategies that have been applied earlier for the models.

- Testing offers flexibility in the strength of properties that are to be shown – from demonstrating that an implementation can possibly execute a certain sequence of events to the proof that useful pre-order relations hold between it and the specification.

  Testing allows partial verification and validation in the cases of large implementations which are subject to state explosion.

- The use of observation composition (specifically parallel composition in process algebras) offers the potential to yield a good deal of information through appropriate design of the test. For process algebra, the composition is a specification itself which may thus be simulated, thereby revealing much more diagnostic information than many model checking tools.

However, there are some difficult practical problems associated with testing of the physical implementation, which the formal developer needs to bear in mind. A couple are:

- How does one know that every outcome has resulted? Even though, one has applied every possible test in the test suite, it may never be established conclusively that all the behaviour of the physical realization has been accounted for – e.g., in a vending machine, you offer 20p on one occasion and got a chocolate; but how do you know that next time you will get a chocolate also? In practice, some non-deterministic

behaviour may be masked. A compromise solution to this is to conclude after n duplicate responses from applying the same test sequence, that there is no further (non-deterministic) behaviour possible.

- How does one know that deadlock has arisen? (Consider that you put in a coin and it disappears from view and gets stuck). One way around this problem is to assume after a certain delay that deadlock has arisen. In this case, divergence caused by an infinite internal loop is considered tantamount to deadlock.

5.3 Some testing notions illustrated formally in LOTOS

The theory of testing for process algebras was originally developed in a very general theoretical setting, as covered fully in [Hen88]. Within the LOTOS community attention has been focused on formalising intuitive notions of what constitute 'valid implementations' with respect to testing and to construct appropriate test suites derived from specifications. It is an approach which is generally used to demonstrate selective aspects – partial verification and the satisfaction of particular properties – with generally less expense on resources. Hence relations between successive implementations may be verified, though not as strong as those which may be shown by working directly with the specification.

The main thrust of research in testing LOTOS specifications has been to implement formally the methodology of *conformance testing* [ISO89c]. Conformance is one notion of valid implementation which, in small cases, may be enhanced to ensure robustness, as discussed in this chapter. Here we employ a formal notion of testing, which fits within a general framework of formal testing presented in [ABe$^+$90, BAL$^+$89], and for which a more recent presentation is to be found in [Tre94].

In LOTOS we represent an input (or environment) simply as another process – a *test process* (or *tester*) T. A *test* consists of composing T in parallel with the IUT). A *trace*, being a sequence of events as a result of executing a process – being here the composition – constitutes *interaction* in this process algebra context. Outcomes are either finite computable traces of a certain form, with verdicts 'success' or 'fail', or traces which provide no useful information, possibly incomputable, with verdict 'inconclusive'.

5.3.1 Test Requirements

For tests to be effective, requires the consideration of two parts: the design of the tester and the amount of synchronisation – on a set $\mathcal{A}$, say – stipulated in the composi-

tion. The latter is an important factor that contributes to the *strength*, being the level of constraint on the behaviour that results in executing the test. Regarding the design of the tester, the first decision is its set of events (sometimes called *label set*). Testers typically reflect some user requirements and can vary from containing just a few events to all those in IUT plus some others. If IUT is some implementation with respect to a specification S, then we denote by $T(S)$ (or just T where S is known) the tester has been derived from S. IUT may have extra events which are implementation details that are not in T's label set.

We use some extra events, $\mathcal{F}$ say, which we call *flags*, used for determining verdicts. The tests are designed with flags being triggered on reaching certain states, typically after certain traces are performed, from which we may draw conclusions. IUT is not obliged to co-perform events in T unless they are stipulated in the synchronisation. Thus, in order to minimise inconclusive verdicts through the synchronisation, the general practice is to require that *all* observable actions are in the synchronisation set $\mathcal{A}$. Note that this makes the assumption that we know *a priori* what are the actions possible for IUT. In our work we make the slight refinement of defining $\mathcal{A}$ to be the set of all actions in both IUT and T, but not in $\mathcal{F}$.

For LOTOS processes in general, non-determinism arises when there is a choice with offers of identical actions and/or of the internal **i** action, the latter we call *compulsory*, and possibly arising through the hiding of some observable action(s). When a test possesses such behaviour, there may be a wide variety of possible execution paths, so there needs to be a way of ensuring that each path is accounted for.

In summary non-determinism may arise if a node contains the following choices (a is an event, **i** is the internal action, P and Q are processes):

Case (i)	(a;P) [] (**i**; Q)
Case (ii)	(a;P) [] (a;Q)
Case (iii)	(**i**;P) [] (**i**;Q)

Otherwise the behaviour is deterministic (insofar as guards and predicates associated with actions may be resolved) and we say that any choices are *benign* or *mutually optional*. In any choice, all those branches that do not have an internal action prefix are termed *optional*.

It is often the case that when the structure of a formal implementation (model) is known – it may well be given as a process definition. This is where testing a model is easier that testing a physical implementation since the only way to know completely all possible behaviour is to open it up – often impractical for a device, but tractable for a model. This

also emphasises the importance of working out a testing strategy at an early stage: the conformance of the physical implementation can then be carried out based on the formal testing strategy.

We list more systematically these ideas for the design of test processes. In particular, we include diagnostics for cases of failure.

Notation Let the set of flags $\mathcal{F}$ be partitioned into two: $\mathcal{F}_{succ}$ indications of success and some others $\mathcal{F}_{diag}$ which are used for diagnostics when there are failures. Let S be a specification and $\mathcal{A}ct(S)$ denote the set of actions belonging to S.

Through a test process T, say, we wish typically to test whether certain behaviour is viable. In order to glean as much information as possible, we may construct T to be:

1. *Applicable* — we require that apart from flag events, T can only perform actions belonging to S, i.e. $\mathcal{A}ct(T) \subseteq \mathcal{A}ct(S) \cup \mathcal{F}$

2. *Discriminating* (where necessary) — if a tester is to test for multiple behaviour, it needs to deduce the choices possible in IUT for this determines what transitions are optional and what are compulsory.

3. *Terminating* – to avoid superfluous non-terminating behaviour, where possible, every trace in T is either of finite length or infinite due to one or more recursive loops of finite length (this is acceptable for testing some kinds of liveness when it is not known beforehand how many iterations of the loop are required for IUT to eventually perform certain behaviour). We say that T *terminates* when its execution sequences (or traces) cannot be extended. We call such traces *maximal* and say that T has reached a *terminal state*.

4. *Conclusive* — we construct T such that the last action of all maximal traces is in $\mathcal{F}$ and denotes either a successful or unsuccessful test execution. We call a maximal trace with last action in $\mathcal{F}_{succ}$ a *successful maximal run* and say that T *terminates successfully*. We allow failures to be included in maximal traces in order to design testers which provide diagnoses. (If the test composition terminates before a flag is raised, this denotes failure).

We apply these principles later in the design of the unified tester for the reduction preorder.

5.3.2 Test analysis

To draw out worthwhile conclusions about the specifications being tested, the analysis of the tests categorises test executions.

The semantics of LOTOS dictate that in the test composition there must be complete multiway synchronisation on all actions specified within the parallel operator. This enables us to infer information about the behaviour of IUT by selecting an appropriate T to act as an 'environment'. With $\mathcal{A}$ and T constructed as above, we have that the observable traces of the resulting test composition are traces of IUT's external behaviour. If a trace of IUT cannot be extended by a further observable action to match one in T, then the composition will subsequently deadlock.

This mechanism of synchronisation enables us to design T to show whether or not IUT possesses a particular trace, typically some desirable or undesirable behaviour. If we apply points 3) and 4) above, we simply look for indications when deadlock arises (indicated by `stop`) – if T has reached a successul terminal state, then we draw a verdict 'success'; otherwise, we draw a verdict of 'fail'.

Definition

i. Given a specification S and a test T, T has a *may response* when applied to S if it terminates successfully for at least one execution of the test composition.

ii. Given a specification S and a test T, T has a *must response* when applied to S if it terminates successfully for every execution of the test composition.

For T and $\mathcal{A}$ as above and IUT representing S this implies:

- if T has a may response, then there exists a trace σ in IUT, which is some maximal trace t in T with the last action (flag) removed.
- if T has a must response, then every trace σ in IUT which is a subtrace of some t in T, may be extended to t, except for the flag action.

If T does not have a 'must' response, then provided it is computable it will fail for at least one of its executions: either it will deadlock or it will terminate, without raising a flag in $\mathcal{F}_{succ}$, i.e. in an unsuccessful state.

Definition MAY and MUST tests

For a specification S and a test T, a *may (must) test* is defined to be the evaluation of the predicate *T has a may (must) response when applied to S* with valuations 'success' if this predicate is true; 'fail' if this predicate is false; and 'inconclusive' otherwise.

The 'may' and 'must' tests are generic – they may be applied to any range of test constructions, some of which are defined below. Note that in the case that T is just a trace, where IUT is deterministic, there is no difference between a 'may' and 'must' test.

5.3.3 Some example testers

We give below some examples of testers, following the design criteria above. These are only a subset of possible LOTOS behaviour expressions and are expressed in a particularly simple form.

1. Sequential Tests

Sequential tests are those tests where T is defined to be a single trace. This is also called *trace testing*.

```
process SequentialTest [ ... < gates > ..., success] : exit :=

   <action_1>;
   <action_2>;
      .
      .
      .
   <action_n>;
   success; exit

endproc (* SequentialTest *)
```

2. Property Tests

Property testing consists in testing the satisfaction of more general behaviour, where T is a general process, usually more than just a trace. The four criteria above are applied for these kinds of tests, which vary as much as the variety of LOTOS behaviour expressions.

One kind of property test is a *refusal set test*, which checks if a set of events $\mathcal{E}$ is rejected in the state where it is applied [Bri87]. It is a process which consists of a number of choices, one of which offers initially the 'success' event, whilst the others offer initially

those events to be rejected. It may be used in conjunction with a trace test which consists of a trace which leads up to the state where a refusal set test is to be applied.

The use is maximised if $\mathcal{E}$ is specified to be the complement of allowable actions in the set of all observable actions.

Applying the definitions above, we have, e.g., a *may sequential test* evaluates to 'success' if T terminates successfully, having executed a desired trace of S. Larger tests may be regarded as consisting of a suite of such tests.

A 'may' test (of the refusal set test) is unhelpful as it is always satisfied, but a 'must' test is useful for it indicates whether or not the set Γ is rejected whatever the executions in IUT.

There is an informative paper on trace testing and property testing in [CG93], where safety and liveness properties are specified and tested using LOLA[Lla91] for the service definition of the ISO Association Control Service Element. This is an interesting alternative to the use of temporal logic, which is a common technique for validating such properties, generally favoured due to its expressiveness. However, an advantage of testing drawn out in that paper is the great flexibility in the choice of testing scenario, particularly the handling of data values. A similar testing approach is used in [Tho94] to highlight how process algebras revealed errors in the design of a safety-critical medical application.

5.4 A generic formal framework for Testing

In this section we put testing on a sound theoretical basis and show how it can tackle more comprehensive validation, called in this context *conformance*.

A special framework needs to be set up for the formal theory and methodology of testing and conformance. The nature of testing is such that we cannot work directly with the specification but can only deduce properties of the specification (and its formal description) through observing its *behaviour* in a given environment. Thus a formal framework has to operate at two levels: when analysing an implementation under test (IUT) for conformance to a specification S, we seek, at the principal level, to establish *implementation relations* between IUT and S. We achieve this through performing at another level analysis based on the *observation* of tests to deduce whether or not an implementation relation holds.

Underlying our approach is the *observation framework* due to Brinksma et al. [ABe$^+$90, BAL$^+$89] which formalises testing in a general way so as to be suitable for a wide range of formal languages, including the Formal Description Techniques Estelle, LOTOS

and SDL [Tur93]. Within the observation framework we fit a testing system for process algebras – the *Experimental System* due to Hennessy and De Nicola [Hen88]. In the latter, there are some useful points as regards the kinds of behaviour any testing theory needs to consider, especially non-determinism – where, over a number of executions, the IUT may respond in various ways to the same test.

In order to establish formally the link between observations of a specification's behaviour and its actual formal expression, the framework involves constructing an *observation relation* to *operationalise* the corresponding implementation relation.

5.4.1 Notions of conformance and refinement

Given a specification S, what do we consider is a valid *implementation* I? Can we test this and, if so, how? What do we mean by testing? How can we formalise this and what formalisms would lend themselves realistically to proof of validity? What are the methods of proof? The consideration of such questions has led to a theory of *conformance* for 'valid implementations' and a theory of *conformance testing* for demonstrating conformance. The work has been extensive: the formalisation of testing has required the consideration of formal relations between designs and a variety of different notions of implementation (e.g.s [LOT92b, BSS86]). This has led to assorted testing relations, notably pre-orders and equivalences for process algebras in general [NH84, Hen88] and LOTOS in particular [Bri87, BSS86]. Further on, work has been done on the derivation of tests [Bri89, ABe$^+$90] and various methodologies for derivation (e.g.s [Wez89, HvB94]); an application of one such method (CO-OP) to a case study is reported in [WBL91]. An overview that covers most of these developments is presented in [Tre94].

We follow largely the material in [Hen88] and [ABe$^+$90, BAL$^+$89]; in the former Hennessy provides some useful elements to be considered in any formal analysis. A formal notion of testing is presented for process algebras, using their framework (recapitulated in sections 5.4.2 and 5.4.3) which is based on the principle of observation of the behaviour of a specification – for this is fundamental to the nature of testing.

The set of conforming implementations for a given specification may be infinite, so to be practical this set may be specified indirectly using a formal relation and a *behaviour specification* (model-based specification), or a *requirement specification* (logical/axiomatic specification) [Tre94]. The former is a set of behaviour expressions determined by a relation between behaviour expressions, whilst the latter is the set of behaviour expressions such that a given set of properties, formulated in some language of logic, are satisfied. The work

conducted in Chapter 4 dealt with safety requirement specifications; the material below deals with behaviour specifications.

The theory that has been developed supports a stepwise refinement which may be regarded as starting off with a (potentially infinite) set of implementations conforming to S and gradually reducing this set by imposing extra conditions related to behaviour and/or properties. By defining appropriate relations, consistency between successive refinements may be demonstrated by testing.

Here, we choose that specification and implementation are relative notions in a hierarchy of system descriptions, where we define that one description is viewed as an implementation of another description, the specification, if the former may be observed to result from the latter by essentially resolving choices that were left open in the specification. This notion of implementation appears fundamental; it has been formalised in a wide variety of settings, having been first described for processes in [BHR84], and subsequently treated in the context of LOTOS (as the *reduction* relation) in [BSS86]. Implementations in this sense may be characterised by two intuitive notions of what is a valid implementation I of a specification S:

CONF1 Everything prescribed by S should be implemented in I,

CONF2 Everything that I does must be allowed by S

Both of these are subject to interpretation. For the first condition, which is one of conformance, we take the contrapositive: "whenever I can refuse something then S must also be able to refuse it". The second adds robustness, for which one may provide the interpretation that I cannot engage in (extra) behaviour which is not specified in S. This can be thought of as analogous to requiring 'clearance' for any action.

5.4.2 A formalisation of behavioural conformance

In this subsection we quote from the work in [ABe+90] to define the requirements for a formal behavioural conformance, widely applicable, though with process algebras especially in mind.

We refer to specifications as *behaviour expressions*. For two expressions, B_1 and B_2, to have (formally) the same behaviour we write $B_1 \approx_R B_2$, where $\approx_R$ is some equivalence relation. We stipulate that such a relation may be factorised into pre-orders (i.e. relations that are reflexive and transitive) $\leq_R$:

$$B_1 \approx_R B_2 \iff B_1 \leq_R B_2 \wedge B_2 \leq_R B_1. \tag{5.1}$$

The pre-orders $\leq_R$ may be used to express 'is an implementation of", which then gives equation 5.1 the intuitively pleasing meaning that B_1 and B_2 are equivalent if and only if they have the same class of implementations. Henceforth we refer to $\leq_R$ as an *implementation relation* when the above interpretation applies; in that case we define

$$Impl_R(B) = \{C | C \leq_R B\} \tag{5.2}$$

Thus, given a behaviour expression S that specifies some system, the *conformance problem* is to determine whether the behaviour B_I of a given implementation I of S is valid, i.e. whether $B_I \in Impl_R(S)$.

5.4.3 Observers and Tests

Testing, unlike standard verification, does not work directly on behaviour expressions themselves, only behaviours. Testing only allows us to *deduce* information about the behaviour expression of I itself. In this section, a language of testing is set up (quoting from [ABe+90]) concerned with observations of behaviour and which is then tied in with behaviour expressions themselves. A framework is constructed which enables external validation of implementations satisfying the relation $\leq_R$. To achieve this requires some operational procedure to demonstrate satisfaction on the basis of *observations* of the behaviour of implementations. Hence the relation $\leq_R$ should be understood not only as a relation between behaviour expressions, but also as a relation between *black box processes* whose behaviour (to be revealed) can in principle be described by behaviour expressions.

We define an *observation framework* as a triple $(\Omega, \Sigma, |\!\vdash)$, where Ω is a set of *observers*, Σ is a set of *observations*, and $|\!\vdash$ is an *observation composition*. The behaviour in this composition, like the observer O and/or observed B, can be non-deterministic. The set of observations that result from O regarding B is defined by $|\!\vdash$, which constitutes interconnection between behaviour expressions. It can be interpreted as a mapping: if Beh_{Proc} is the set of behaviour expressions of observed processes, and Beh_Ω is the set of behaviour expressions of the observer processes, then $|\!\vdash$ is of the type

$$|\!\vdash : Beh_{Proc} \times Beh_\Omega \rightarrow \mathcal{P}(\Sigma), \tag{5.3}$$

where $\mathcal{P}$ denotes the power set.

Having introduced these definitions for testing in the observation framework, the operationalization of an implementation relation $\leq_R$ is determined by an *observation relation* $\lhd_R \subseteq \mathcal{P}(\Sigma) \times \mathcal{P}(\Sigma)$, which we intend to be the means for actually showing through observation of external behaviour that an implementation relation holds. At this point, if we start with $\leq_R$ already in mind, then we need to consider:

- Is $\leq_R$ testable, i.e. does there exist a $\lhd_R$ such that the implementation relation holds?
- Is $\leq_R$ computable through $\lhd_R$, i.e. does the observation relation lend itself to establishing relations subject to current computing resources?

Alternatively, $\leq_R$ can be defined by starting with $\lhd_R$, in which case one needs to look at the strength of $\lhd_R$. Historically, the testing theory has followed somewhat later than other non-testing based theories, where already many $\leq_R$'s had been defined. Perhaps the first theory of testing, based on observation, was developed by Hennessy and De Nicola – their Experimental System. It is this system which we re-examine later in the light of the testing framework.

In any case, we stipulate that $\lhd_R$ is an observation relation for $\leq_R$ when:

$$B_1 \leq_R B_2 \iff \forall O \in \Omega : B_1 \mid\vdash O \lhd_R B_2 \mid\vdash O. \tag{5.4}$$

Hence

$$B \in Impl_R(S) \iff B \leq_R S \iff \forall O \in \Omega : B \mid\vdash O \lhd_R S \mid\vdash O. \tag{5.5}$$

The problem of whether or not an implementation is valid according to the relation $\leq_R$ may thus be decided by using valuations of the composition $\mid\vdash$ for each observer, which correspond to whether or not the observation relation $\lhd_R$ holds. For this we introduce *verdicts* through the definition of a family of mappings

$$\{v_{O,R} : \mathcal{P}(\Sigma) \to \{pass, fail\}\}_{O \in \Omega} \tag{5.6}$$

with

$$v_{O,R}(V) = \begin{cases} pass & \text{if } V \lhd_R S \mid\vdash O, \\ fail & \text{otherwise} \end{cases} \tag{5.7}$$

This allows us to reformulate 5.5 as:

$$B \in Impl_R(S) \iff \forall O \in \Omega : v_{O,R}(B \mid\vdash O) = pass \tag{5.8}$$

That is, under the implementation relation, B is an implementation of S if and only if all the results of every observation composition have verdict *pass*.

Note that where R is understood, we may the notation $v_{O,R}$ may be simplified to v_O (an assumption made in [BAL$^+$89, ABe$^+$90]).

Finally for this section, we define what it means for an implementation relation to be testable:

Definition (testability)

Given an observation framework, $(\Omega, \Sigma, \mid\vdash)$, an implementation relation $\leq_R$ is *testable* if for all behaviour expressions B there exists a set of observers Ω, an observation relation $\lhd_R$ and a verdict function v such that for any behaviour expression B',

$$B' \in Impl_R(B) \iff \forall O \in \Omega : v_{O,R}(B' \mid\vdash O) = pass$$

It may not be necessary to use the entire universe of observers to determine whether or not an implementation relation holds. If a subset exists which can do this, we call this set a *test suite*, which is formally defined as:

Definition (test suite)

Given a specification S, a *test suite* for $Impl_R(S)$ is a family of pairs $\{< T, v_T >\}_{T \in \Pi}$ with $\Pi \subseteq \Omega$ and $\{v_T\}_{T \in \Pi}$ a family of verdicts such that:

$$B' \in Impl_R(S) \iff \forall O \in \Pi : v_{O,R}(B' \mid\vdash O) = pass$$

Where the v_T is understood, we denote a test suite by its index set, Π.

5.4.4 Incorporating an Experimental System due to Hennessy and de Nicola

In this section we show how the observation framework can incorporate the testing system called the *Experimental System*, due to Hennessy and De Nicola, and which is described in detail in Sections 2.1 and 2.2 of [Hen88]. This system provides the basis for operationalising the implementation relations in which we are interested and which are treated afterwards. It characterises testing in terms of 'may' and 'must' tests. To recap, our work is on the development of observation relations as the basis for notions

of implementation, i.e. on the RHS of (5.4), focusing in particular on the nature of the observation composition.

The link has already been shown briefly by Brinksma et al in [BAL$^+$89], using a set of observers defined to be all processes with states marked as either *succ* or *unsucc*. Here we provide fuller coverage, showing where everything fits in. We also define a few extra terms to anticipate later developments.

In the framework, we regard Ω as a set of processes – *testers* (or *experimenters*), and $|\vdash$ as the result of executions of (parallel) compositions using an interconnector, denoted $||$. The behaviour arising from $||$ is built up from *transitions* which may be represented by relations $\rightarrow$ written as:

$$\rightarrow \subseteq (Beh_{Proc} \times Beh_{\Omega}) \times (Beh_{Proc} \times Beh_{\Omega})$$

Let $P \in Beh_{Proc}$, and let $O \in Beh_{\Omega}$. Then writing the relation $\rightarrow$ in infix notation we have

$$P \,||\, O \rightarrow P' \,||\, O'$$

for some behaviour expressions P' and O'.

If both processes participate during the transition, then we term this *interaction*.

A *test* between processes P and O is simply the composition $P||O$. A *test run* (or *experiment*) is a sequence of transitions (possibly interactions) conducted in the composition of the tester and the process. It is represented by:

$$P_0 \,||\, O_0 \rightarrow P_1 \,||\, O_1 \rightarrow \ldots \rightarrow P_n \,||\, O_n \rightarrow \ldots$$

We say that such a sequence is a *computation* if it is maximal, i.e. it is infinite or it is finite with terminal element $P_n \,||\, O_n$ $(n \geq 0)$ which has the property, $P_n \,||\, O_n \rightarrow P' \,||\, O'$ for no pair P', O'.

The set Σ of *observations* is in terms of the transitions, which are typically labelled.

In equations 5.6 to 5.8 there was introduced the notion of an observation being given a verdict 'pass' or 'fail' depending upon whether or not an observation relation was satisfied. Here we use an 'experimental system' to make explicit when such a relation holds. We provide a mechanism for awarding verdicts according to whether tests have been 'successful' or 'unsuccessful'.

Let $\mathcal{S} \subseteq \Omega$ be a set, denoting 'successful' states (processes). We stipulate that the *computation is successful* if $O_k \in \mathcal{S}$ for some $k \geq 0$, i.e. if the tester passes through some successful state. We do not specify what constitutes an unsuccessful computation, leaving this open to definition in the respective contexts.

We are now able to define an Experimental System.

Definition Given an observation framework $(\Omega, \Sigma, \vdash)$, an *Experimental System* $\mathcal{ES}$ is a collection of the form $< \mathcal{P}, \mathcal{O}, \mathcal{R}, \mathcal{S} >$, where

i) $\mathcal{P}$ is an arbitrary set of processes

ii) $\mathcal{O} \subseteq \Omega$ is an arbitrary set of observers/testers

iii) $\mathcal{R} = \{\stackrel{a}{\rightarrow} \subseteq (\mathcal{P} \times \mathcal{O}) \times (\mathcal{P} \times \mathcal{O}) \mid a \in \mathcal{L}\}$ is a set of binary interacting relations.

iv) $\mathcal{S} \subseteq \mathcal{O}$ is the success set.

For such an $\mathcal{ES}$, and P in $\mathcal{P}$, and O in $\mathcal{O}$, we let $Comp(P,O)$ be the set of computations whose initial element is $P \mid\mid O$. Let *succ* denote a successful computation and *unsucc* an unsuccessful computation. Let $Result(P,O) \subseteq \{succ,\ unsucc\}$ be defined by:

$succ \in Result(P,O)$ if $Comp(P,O)$ contains a successful computation.

$unsucc \in Result(P,O)$ if $Comp(P,O)$ contains an unsuccessful computation.

5.4.4.1 Testing relations

We are able to formalise notions of 'may' satisfy and 'must' satisfy a test through defining 2 relations *may* and *must*:

$P\ may\ O$ if $succ \in Result(P,O)$

$P\ must\ O$ if $unsucc \notin Result(P,O)$

Now define corresponding observation relations $\lhd_{may}$ and $\lhd_{must}$:

$P \vdash O \lhd_{may} S \vdash O$ if $P\ may\ O \Rightarrow S\ may\ O$

$P \vdash O \lhd_{must} S \vdash O$ if $P\ must\ O \Rightarrow S\ must\ O$

We then define respective *verdicts* as:

$$v_{O,may}(V) = \begin{cases} pass & \text{if } V \lhd_{may} S \vdash\ O, \\ fail & \text{otherwise} \end{cases} \tag{5.9}$$

$$v_{O,must}(V) = \begin{cases} pass & \text{if } V \lhd_{must} S \vdash\ O, \\ fail & \text{otherwise} \end{cases} \tag{5.10}$$

The corresponding implementation relations $\leq_{may}$ and $\leq_{must}$ respectively are pre-orders, and in 5.8, we have:

$$I \in Impl_{may}(S) \iff \forall O \in \Omega : v_{O,may}(I \mid\vdash O) = pass \tag{5.11}$$

$$I \in Impl_{must}(S) \iff \forall O \in \Omega : v_{O,must}(I \mid\vdash O) = pass \tag{5.12}$$

We can now define a testing-based pre-order which reflects the notions of robust conformance described at the end of section 5.4.1.

- *conformance*: whenever, for a given tester, S has no unsuccessful computations, then neither must I.
- *robustness*: whenever, for a given tester, I has a successful computation, then so must S.

These notions motivate the following definition of a *robust conformance testing* preorder, denoted $\leq_{testrc}$:

Definition $I \leq_{testrc} S \iff I \in Impl_{may}(S) \wedge S \in Impl_{must}(I)$.

A second preorder relation (as defined in [Hen88]), which we call the *testing* pre-order is as above, except for reversing the $\leq_{must}$ relation in the conjunction:

Definition $I \in Impl_{testing}(S) \iff I \in Impl_{may}(S) \wedge I \in Impl_{must}(S)$.

We may also define testing equivalence:

Definition Two processes are *testing equivalent*, written $P \sim Q$ if $P \in Impl_{testing}(Q)$ and $Q \in Impl_{testing}(P)$

Similarly, we may define the equivalence relation corresponding to the 'testrc' preorder, and see that it co-incides with testing equivalence.

The testing equivalence relation may be viewed as the formalisation of the system design concept of a black box. Two systems are *testing equivalent* (equivalent black boxes) if they cannot be distinguished by testing. Thus we may define two specifications S_1 and S_2 to be *testing equivalent* if every may (must) test of S_1 is also a may (must) test of S_2.

In [Lan90] there is some related work which defines a more powerful set of observers through the introduction of a slightly extended version of LOTOS (TLOTOS) designed to

have the special facility of deadlock detection; the paper also covers the relationship between the reduction and *failures* pre-orders as described here, but for finite processes only and not within the observation framework.

5.4.4.2 Instantiating the Experimental System with LTS Operational Semantics

We now specialise $\mathcal{ES}$ by specifying that operational semantics be defined through the use of LTS, using the definitions given for LOTOS in figure A.1 of Appendix A. Note that τ is a 'silent' unobservable action, which we also call an *internal action*. We stipulate that the *empty string* belongs to L^*, and denote it by ε and that for any $s \in \mathcal{L}^*$, we define $s^0 = \epsilon$. Given $a \in L$, in any state P, whenever $P \stackrel{a}{\Rightarrow} P'$, for some P', we have an *observable transition*. A sequence $< a_0, a_1, ..., a_n >$ of observable transitions is called a *trace*. For a process P, the set of traces is denoted $Tr(P)$.

We now instantiate in the Experimental System, and employ labels for the transitions. Hence, these are denoted as parameterised relations with signature:

$$\stackrel{\mu}{\rightarrow} \subseteq (Beh_{Proc} \times Beh_{\Omega}) \times (Beh_{Proc} \times Beh_{\Omega})$$

As before let $P \in Beh_{Proc}$, and let $O \in Beh_{\Omega}$. Then writing the relation $\stackrel{\mu}{\rightarrow}$ in infix notation we have

$$P \,||\, O \stackrel{\mu}{\rightarrow} P' \,||\, O'$$

for some behaviour expressions P' and O' which satisfy the relevant transition rules.

A test run is now represented by a sequence of the form

$$P_0 \,||\, O_0 \stackrel{\mu_0}{\rightarrow} P_1 \,||\, O_1 \stackrel{\mu_1}{\rightarrow} \ldots \stackrel{\mu_{n-1}}{\rightarrow} P_n \,||\, O_n \stackrel{\mu_n}{\rightarrow} \ldots \text{ where } \mu_i \in \mathcal{L}.$$

The set Σ of possible *observations* is the set of traces, $\Sigma = \{< a_0, a_1, ... >: \forall i \in \{0, 1, ...\}.a_i \in L\}$.

5.5 Establishing Robust Conformance as a testing relation

We now approach testing from the conformance viewpoint; by appropriate choice of definitions, we show that conformance-based notions are closely related to the testing notions we examined above; this coverage follows that in e.g. [ABe+90], but provides more detail, concentrating on the one relation. In our treatment, which is geared towards

refinement, we seek to show implementation relations and thus view pre-order relations as appropriate. Hence, we concentrate on showing that we may solve through testing (in theory, at least) the problem of determining for a given S whether or not some I satisifies $I \leq_R S$ for some implementation relation $\leq_R$. The main result of this section is that, subject to minor restrictions, one such relation, called *reduction* (or *failures* preorder) co-incides with a testing relation 'testrc' defined within the Experimental System.

5.5.1 Preliminary Definitions and Results

First, we need to introduce a couple of definitions which describe the immediate capability to perform actions in a given state.

Definition (Refusal function)

The *refusal function* of a process B, $R_B : L^* \rightarrow \mathcal{P}(\mathcal{P}(L))$ is defined for each $\sigma \in L^*$ by:

$$R_B(\sigma) = \{A \subseteq L \mid \exists B' : B \stackrel{\sigma}{\Rightarrow} B' : \forall a \in A.B' \stackrel{a}{\not\Rightarrow})\}$$

The set $R_B(\sigma)$ is called a *refusal set*.

Note that this differs slightly from the definition in [ABe$^+$90] in that a second colon is used (between last two terms) instead of a conjunction ($\wedge$). This is to aid logical clarity.

We may also define a function that is the complement of the refusals in $\mathcal{P}(L)$ which we call the *acceptance function*. Note that this is different from the function defined in [ABe$^+$90]:

Definition (Acceptance function)

The *acceptance function* of a process B, $A_B : L^* \rightarrow \mathcal{P}(\mathcal{P}(L))$ is defined for each $\sigma \in L^*$ by:

$$A_B(\sigma) = \mathcal{P}(L) \setminus R_B(\sigma).$$

The set $A_B(\sigma)$ is called an *acceptance set*.

The following result is immediate consequence of the properties of sets:

Lemma 5.5.1 *Suppose* $B_1, B_2 \in Beh_{Proc}$ *and that* $\sigma \in L^*$, *then* $R_{B_2}(\sigma) \subseteq R_{B_1}(\sigma) \iff A_{B_2}(\sigma) \supseteq A_{B_1}(\sigma)$.

Some simple manipulation gives an alternative expression of the Acceptance function:

Lemma 5.5.2 *Let* $B \in Beh_{Proc}$ *and* $\sigma \in L^*$. *Then* $A_B(\sigma) = \{A \subseteq L \mid \forall B' \in Beh_{Proc} : B \stackrel{\sigma}{\Rightarrow} B' : \exists x \in A.B' \stackrel{x}{\Rightarrow}\}$

Proof We have $R_B(\sigma) = \{R \subseteq L \mid \exists B' \in Beh_{Proc} : B \stackrel{\sigma}{\Rightarrow} B' : \forall x \in R.B' \stackrel{x}{\not\Rightarrow}\}$ and $A_B(\sigma) = \mathcal{P}(L) \setminus R_B(\sigma)$. Therefore $A_B(\sigma) = \{A \subseteq L \mid \neg(\exists B' : B \stackrel{\sigma}{\Rightarrow} B' : \forall x \in A.B' \stackrel{x}{\not\Rightarrow})\}$. This implies $A_B(\sigma) = \{A \subseteq L \mid \forall B' \in Beh_{Proc} : B \stackrel{\sigma}{\Rightarrow} B' : \neg(\forall x \in A.B' \stackrel{x}{\not\Rightarrow})\}$ if and only if $A_B(\sigma) = \{A \subseteq L \mid \forall B' \in Beh_{Proc} : B \stackrel{\sigma}{\Rightarrow} B' : \exists x \in A.B' \stackrel{x}{\Rightarrow})\}$ □

The following simple Lemma shows that a given acceptance set can absorb any other members of the label set.

Lemma 5.5.3 *Let* $B \in Beh_{Proc}$ *and* $\sigma \in L^*$. *If* $A \in A_B(\sigma)$ *then* $\forall y \in L.A \cup \{y\} \in A_B(\sigma)$.

Proof From Lemma 5.5.2, clearly if $A \in A_B(\sigma)$ then for any $y \in L$ we have: $\forall B' \in Beh_{Proc} : (B \stackrel{\sigma}{\Rightarrow} B').\exists x \in A \cup \{y\} : B' \stackrel{x}{\rightarrow}$. So $A \cup \{y\} \in A_B(\sigma)$ □.

We now define a relation *reduction* for robust conformance, providing another formalisation, being a conjunction of each of the two notions CONF1 and CONF2 given informally.

Definition (Reduction Relation)

I is a *reduction* of a specification, written $\leq_{red}$, if

$$I \leq_{red} S \iff I \textbf{ conf } S \wedge Tr(I) \subseteq Tr(S)$$

where **conf** is an implementation relation for conformance and is defined as:

$$I \textbf{ conf } S \iff \forall \sigma \in Tr(S) : R_I(\sigma) \subseteq R_S(\sigma)$$

Note that another relation given in the literature is *extension* [BSS86], which is defined as for reduction except that the relation on trace inclusion is reversed. Hence, the equivalence relation for extension also co-incides with testing equivalence. The extension relation, in the way it allows extra behaviour does not check for robustness, so we do not use it here.

The following Lemma shows that the reduction relation may be characterised entirely in terms of refusals:

Lemma 5.5.4 *Let* $I, S \in Beh_{Proc}$. $I \leq_{red} S \iff \forall \sigma \in L^* : R_I(\sigma) \subseteq R_S(\sigma)$

Proof First note that from the definition, we have: $\forall \sigma \in L^*, P \in Beh_{Proc}$ if $\sigma \in Tr(P)$ then $\emptyset \in R_P(\sigma)$ (since for all behaviour expressions $B' : \forall a \in \emptyset : B' \stackrel{a}{\not\Rightarrow}$); if $\sigma \notin Tr(P)$ then $R_P(\sigma) = \emptyset$

($\Rightarrow$)

(i) $Tr(S) \subseteq L^*$ so the **conf** relation holds.

(ii) Suppose $Tr(I) \not\subseteq Tr(S)$, so $\exists \sigma \in Tr(I)$ with $\sigma \notin Tr(S)$. Note σ is not ε since $\varepsilon \in Tr(B)$ for all behaviour expressions B. Therefore $R_S(\sigma) = \emptyset$, whilst $\{\emptyset\} \subseteq R_I(\sigma)$, whence $R_S(\sigma) \subset R_I(\sigma)$ which is a contradiction.

($\Leftarrow$) We have already that $\forall \sigma \in Tr(I) : R_I(\sigma) \subseteq R_S(\sigma)$. It remains only to show that the containment of refusals holds also for traces not in the traces of I.

Suppose $\sigma \notin Tr(I)$, then (as before), $R_I(\sigma) = \emptyset$. As $R_S(\sigma)$ is some set, the containment follows at once. □

From this we deduce immediately that the reduction relation is a preorder. The **conf** relation is not, since it may not always satisfy the transitive property. Suppose $P1, P2, P3$ are a sequence of processes with $P3$ **conf** $P2$ and $P2$ **conf** $P1$, then $\forall \sigma \in Tr(P1) \cap Tr(P2)$, we do have $R_{P3}(\sigma) \subseteq R_{P1}(\sigma)$, but transitivity can fail for $\sigma \in Tr(P1) \setminus Tr(P2)$. As an example, let `P1 = i;b;stop [] a;c;stop`, `P2 = b;stop`, `P3 = b;stop [] a;stop`. Then we have $P3$ **conf** $P2$ and $P2$ **conf** $P1$ but $\neg P3$ **conf** $P1$ (as $R_{P_3}(< a >) = \mathcal{P}(L)$ but $R_{P_1}(< a >) \subset \mathcal{P}(L)$ since, e.g., $\{c\} \notin R_{P_1}$).

5.5.2 Notes and Examples

Given this characterisation of the refusal function, we can observe more easily the following properties that describe the reduction preorder.

Regarding processes in terms of their depiction as trees, a process I is an implementation in the sense of reduction with respect to S if it is essentially derived from S by making certain kinds of choices at each node of the tree of S. These choices are made according to certain rules, that (up to some suitable notion of equivalence) either preserve branches or drop them. I may be observed as having basically a substructure of S (possibly with some extra duplication), built up in corresponding fashion node by node, starting from the root. The rules are (informally):

- If a node N of S has only deterministic choices, then these must be preserved in I, albeit the choice possibly prefixed by an internal action.

- If N has at least one non-deterministic choice, then the implementer has the option of incorporating as a choice a path whose initial(s) action corresponds to that in any one or more of those non-deterministic paths plus any number (zero or more) of the remaining other (deterministic) branches. The subsequent behaviour of these branches in I depend upon the behaviour of the corresponding path in S.
- In either case, no new paths may be introduced that have initial action different from those that may be performed (perhaps after internal actions) from N.

In summary, at each node the implementer has an option if and only if there is some non-deterministic choice, in which case, at least one of the non-deterministic paths must be incorporated in I.

We provide a few examples to illustrate the relation:

Example 1 `S = a;stop [] i;b;stop`, `I1 = b; stop`, `I2 =a;stop`, `I3 = a;stop [] b;stop`, `I4 = a;stop [] b;stop [] c;stop`.

Then we have:

$I1 \leq_{red} S$,
$I2 \not\leq_{red} S$, since $\{b\} \in R_{I2}(\varepsilon)$, but $\{b\} \notin R_S(\varepsilon)$
$I3 \leq_{red} S$
$I4 \not\leq_{red} S$, since $R_{I4}(< c >) = \mathcal{P}(L)$, but $R_S(< c >) = \emptyset$

Example 2 `S = a;stop [] b;stop`, `I=b;stop [] i; (a;stop [] b;stop)`

Then it may be seen that $I \leq_R S$.

This indicates that the reduction relation does not merely resolve choices that are left open in the specification – an implementation that is a reduction of a specification can exhibit more complex behaviour, though in general, this requires some duplication of initial actions.

Example 3 Allowing implementations to arise through resolving choices that arise in duplication of actions inserted by the specifier seems sensible as it reflects the intention of leaving it to the implementer to decide between such alternatives depending on external factors such as performance, cost etc. This is certainly incorporated in the reduction relation; however, the reduction relation also allows as valid implementations, other processes, which remove benign choices on a non-deterministic branch.

We provide several examples to illustrate these points (further discussion of the drawbacks of the reduction relation are given in [Lan90]).

3.1: `S = a;(b;stop [] c;stop) [] a;b;d;stop`, `I = a;b;stop [] a;b;d;stop` implies
$I \leq_{red} S$, but here we see that I has lopped off an action at a benign choice, contained in a non-deterministic branch.

3.2: `S = a;(b;stop [] c;stop) [] a;(b;d;stop [] c;stop)`,
`I = a;(b;stop [] c;stop) [] a;b;d;stop` implies that we do NOT have $I \leq_{red} S$, since the refusals of S after $< a >$ are smaller than that of I.

3.3: `S= i;(a;stop [] b;stop) [] c;stop`, and `I = a;stop [] b;stop`. Then I is a reduction of S - note that the initial choice involves non-determinism, and hence the benign action 'c' can be dropped in the implementation.

3.4: `S = i;(a;stop[] b;stop)[] i;(c;stop[] d;stop)[] a;stop[] c;stop`, `I1 = a;stop[] b;stop`, `I2 = a;stop[] c;stop`. Then $I1 \leq_{red} S$, but $I2 \not\leq_{red} S$ (since $\{b, d\} \notin R_S(\varepsilon)$). However, removing the 'b' action in the internalised branch that offers 'a' or 'b', reverses these valuations!

5.5.3 Some Guidelines for use of conformance in refinement

Some guidelines are required for effective use of this interpretation of conformance, to elaborate on the options open to the specifier/implementer in the process of moving a process definition along the refinement trajectory. In particular, the specifier has to take account of the rules in the previous section regarding the choice structures so that his/her intentions are retained in the production of conforming refinements.

If the specifier foresees more than one path of continuation from a given state, all of which are compulsory, then this may be implemented as a benign choice:

$$a_1; P_1 \ [] \ a_2; P_2 \ [] \ ... \ [] \ a_n; P_n$$

Implementations I will conform only if I has a corresponding node with a choice which includes each of these paths $a_i; P_i$.

However, if one requires a choice in which there is a mixture of compulsory and optional choices, then all the compulsory choices should be grouped together and prefixed by an internal action, whilst any options should be included as benign choices:

$$\mathbf{i}; (a_1; P_1 \ [] \ a_2; P_2 \ [] \ ... \ [] \ a_i; P_i) \ [] \ b_1; Q_1 \ [] \ b_2; Q_2 \ [] \ ... \ [] \ b_k; Q_k$$

Here, implementations I will conform only if I has a corresponding node with a choice which includes each of these paths $a_i; P_i$, possibly with internal action prefixes.

More generally, suppose that the specifier wishes that an implementation possesses at least one branch from several, some of which offer more than one choice. Then this is specified through the use of an internal action prefix before each such branch as follows:

$$\sum_{i=1}^{l} \mathbf{i}; (\sum_{j=1}^{m} a_{i_j}; P_{i_j}) \; [] \; \sum_{k=1}^{n} b_k; Q_k$$

Here there are l non-deterministic branches, one of which must be preserved in the implementation.

If the specifier wishes to specify that the implementation has some pre-emptive power at a certain state (such as a 'time out' facility), then, as noted in the previous section, under **conf**, if a node offers a choice between a mixture of benign and non-deterministic options, then valid implementations can drop any or all of the benign choices and, further, need only retain one of the non-deterministic branches. In this case, the specifier could define an observable action such as 'timeout' that does not belong to the synchronisation set $\mathcal{A}$, and which can later be hidden. However, once hidden, for the other (benign) branches to remain, some other notion of conformance would have to be used for subsequent stages in refinement.

Once it is decided for a node which paths to include, then some simplification can take place through the removal of internal action prefixes, still leaving conforming implementations. Once this has been done for all nodes, the implementer can seek to reify with details of 'how', and a new notion of refinement should be introduced, with the internal action assuming its more usual conotation of hiding some behaviour rather than being a mechanism for resolving choices.

Example

As an illustration, consider the design of a vending machine VM for maps of a local UK area that will each cost 2 Pounds Sterling. In our design, we wish to allow the machine to accept 50p and 1 Pound coins. Thus the initial state of a specification VM may be:

$\texttt{VM} = \texttt{50p}; \texttt{VM1} \; [] \; \texttt{1Pd}; \texttt{VM2}$, where $VM1, VM2 \in Beh_{Proc}$

However, suppose that the government announces that it is considering issuing a

new 2 Pound coin. In our specification we wish to allow for the possibility of accepting these new coins depending upon what the government decides, but we don't yet know the outcome, though news of the decision will be available in due course. By the reasoning above, we can't simply tack on a branch that accepts a 2Pd choice, instead we need to rewrite the initial definition of VM to:

$$\texttt{VM} = \texttt{i};(\texttt{50p};\texttt{VM1} \ [] \ \texttt{1Pd};\texttt{VM2}) \ [] \ \texttt{2Pd};\texttt{VM3}$$

and some comment is included to explain the choices, including the use of the internal action.

When subsequently, the decision not to go ahead with the new coin is announced, then a subsequent refinement is settled upon (dropping the internal action prefix in the process):

$$\texttt{VM} = \texttt{50p};\texttt{VM1} \ [] \ \texttt{1Pd};\texttt{VM2}.$$

and we can proceed to refine this model with detail.

Overall, the approach above seems workable, but care is needed in interpreting the role of the internal action under **conf**. Whatever relations are used, it is wise to provide comments to clarify the designers' intentions.

5.5.4 Proof that *reduction* is a testing relation

We now show that reduction is testable simply by demonstrating that it coincides with the 'testrc' preorder. As a consequence, reduction may be shown by using 'may' and 'must' tests. The following contributing result shows that for any parallel operator, compositions between I and O that lead the tester to pass through a certain state may be mirrored in the composition of S with O.

Lemma 5.5.5 *Let $I, S \in Beh_{Proc}$ and suppose $R_I(\sigma) \subseteq R_S(\sigma)$ and that $||$ denotes any parallel operator. Let $O \in \Omega$. Then $I||O \stackrel{\sigma}{\Rightarrow} I'||O'$ implies $S||O \stackrel{\sigma}{\Rightarrow} S'||O'$, for some processes O' and S'.*

Proof We do this by induction.

Suppose $\sigma =< a_0, a_1, ..., a_n > \in L^*$.

Base Case

There are three cases depending upon the ability of I and O to perform the initial action a_0.

1. $I||O \stackrel{a_0}{\Rightarrow} I||O_1$, some state 0_1, i.e. a_0 is only performed by O. Then, we have either:

   (a) $a_0 \in R_I(\varepsilon)$, whence $a_0 \in R_S(\varepsilon)$, since $\forall \sigma \in L^*.R_I(\sigma) \subseteq R_S(\sigma)$, and then $S||O \stackrel{a_0}{\Rightarrow} S||O_1$.

   or

   (b) $a_0 \notin R_I(\varepsilon)$, so $O \stackrel{a_0}{\Rightarrow} O_1$ independently of I and hence of any process that offers the action in parallel composition. Hence, $S||O \stackrel{a_0}{\Rightarrow} S||O_1$.

2. $I||O \stackrel{a_0}{\Rightarrow} I_1||O$, i.e. a_0 is only performed by I.

   Then, since $Tr(I) \subseteq Tr(S)$, we have $< a_0 > \in Tr(S)$ and hence $S||O \stackrel{a_0}{\Rightarrow} S_1||O$, for some process S_1.

3. $I||O \stackrel{a_0}{\Rightarrow} I_1||O_1$, i.e. a_0 is performed by both I and O. Again, since $Tr(I) \subseteq Tr(S)$, we have $< a_0 > \in Tr(S)$, so $S||O \stackrel{a_0}{\Rightarrow} S_1||O_1$.

The base case is done.

Inductive Case

A very similar argument is applied. We assume that the result is true for $\sigma' \in L^*$ of length h, where $h \in \mathbf{N}$. Thus, by this hypothesis, we have that $I||O \stackrel{\sigma'}{\Rightarrow} I'||O'$, some I', O'. and that $S||O \stackrel{\sigma'}{\Rightarrow} I'||O'$ with $I \stackrel{\sigma_1}{\Rightarrow}{}' I'$ and $O \stackrel{\sigma_2}{\Rightarrow}{}' O'$, say

Now suppose that the composition of I' and O' can perform the action a_h. As above, there are three cases, and the arguments are analogous.

1. $I'||O' \stackrel{a_h}{\Rightarrow} I'||O''$, some state $0''$, i.e. a_h is only performed by O'. Then, we have either:

   (a) $a_h \in R_I(\sigma')$, whence $a_h \in R_S(\sigma')$, since $\forall \sigma \in L^*.R_I(\sigma) \subseteq R_S(\sigma)$, and then $S'||O' \stackrel{a_h}{\Rightarrow} S'||O''$.

   or

   (b) $a_h \notin R_I(\sigma')$, so $O' \stackrel{a_h}{\Rightarrow} O''$ independently of I' and hence $S'||O' \stackrel{a_h}{\Rightarrow} S'||O''$.

2. $I'||O' \stackrel{a_h}{\Rightarrow} I''||O'$, for some process I'', i.e. a_h is only performed by I'.

   Then, since $Tr(I) \subseteq Tr(S)$, we have $\sigma' \frown < a_0 > \in Tr(S)$ and hence there exist processes S', S'' such that $S'||O' \stackrel{a_h}{\Rightarrow} S''||O'$, for some process S''.

3. $I'||O' \stackrel{a_h}{\Rightarrow} I''||O''$, i.e. a_h is performed by both I' and O'. Again, since $Tr(I) \subseteq Tr(S)$, we have $\sigma' \frown < a_0 > \in Tr(S)$, so there exist processes S', S'' such that $S'||O' \stackrel{a_h}{\Rightarrow} S''||O''$.

Hence the statement is true for traces of length $h+1$. Thus, as the statement is true for $h = 1$, it is true by induction for $h = 2, 3, ...$ and hence for traces of any length, so the result follows. □

Proposition 5.5.6 For a specification S and implementation under test, I, suppose that both I and every tester in Ω are *strongly convergent* (i.e. contain no infinite sequence of internal actions) and that in the Experimental System we define $||$ to be the parallel operator in which synchronisation is on the set L of all observable actions; and a *computation is unsuccessful* if it is not successful. Then,

$$I \leq_{red} S \iff I \leq_{testrc} S$$

Proof From Lemma 5.5.4 it suffices to show that $\forall \sigma \in L^*.R_I(\sigma) \subseteq R_S(\sigma)$ iff $I \leq_{testrc} S$.

($\Rightarrow$) We need to show:

1. $\forall O \in \Omega.\ I\ may\ O \Rightarrow S\ may\ O$
2. $\forall O \in \Omega.\ S\ must\ O \Rightarrow I\ must\ O$

1. Suppose for some $O \in \Omega.\ comp(I,O) \Rightarrow succ \in Result(I,O)$. Then we have:

$$I||O \stackrel{\sigma}{\Rightarrow} I'||O_k$$

for some $\sigma \in \Sigma$, $O_k \in \mathcal{S}$ and $I' \in Beh_{Proc}$, where $I \stackrel{\sigma_1}{\Rightarrow} I'$ and $O \stackrel{\sigma_2}{\Rightarrow} O_k$, say. Since $Tr(I) \subseteq Tr(S)$, we have $\sigma_1 \in Tr(S)$. It follows immediately from Lemma 5.5.5 that $S||O$ has a trace in which O passes through O_k. □

2. We prove the contrapositive, i.e. that $Result(I,O) \neq \{succ\} \Rightarrow Result(S,O) \neq \{succ\}$. There are two cases corresponding to the length of the sequence being finite and infinite. Case (i): Suppose $I||O \stackrel{\sigma_0}{\Rightarrow} I_0||O_0 \stackrel{a_0}{\Rightarrow} I_1||O_1 \stackrel{a_1}{\Rightarrow} \ldots \stackrel{a_{l-1}}{\Rightarrow} I_l||O_l$, and $I_l||O_l \rightarrow I'||O'$ for no pair I', O' and $\forall i \in \{0,1,...,l\}.O_i \notin \mathcal{S}$. From Lemma 5.5.5, we have that any path in $I||O$ can be matched in $S||O$, i.e. there exist processes $S_0, S_1, ..., S_l$ such that $S||O \stackrel{\sigma_0}{\Rightarrow} S_0||O_0 \stackrel{a_0}{\Rightarrow} S_1||O_1 \stackrel{a_1}{\Rightarrow} \ldots \stackrel{a_{l-1}}{\Rightarrow} S_l||O_l$. Let $\sigma_{dk} = \sigma_0 \frown < a_0, a_1, ..., a_l >$. Then since $R_I(\sigma_{dk}) \subseteq R_S(\sigma_{dk})$, we have that there exists an S_l such that $S||O \stackrel{\sigma_{dk}}{\Longrightarrow} S_l||O_l$ with $S_l||O_l \not\rightarrow S'||L'$ for any pair S', L' □

Case (ii): Suppose we have an infinite sequence $I||O \stackrel{\mu_0}{\rightarrow} I_0||O_0 \stackrel{\mu_1}{\rightarrow} I_1||O_1 \stackrel{\mu_2}{\rightarrow} \ldots$. If μ_i is observable, then (again by Lemma 5.5.5), it can be matched by the behaviour of $S||O$, an infinite sequence with $\forall j.O_j \notin \mathcal{S}$. Noting that $I_i||O_i \stackrel{\tau}{\rightarrow} I_{i+1}||O_i$ may or may not be matched by S, we invoke the hypothesis in the proposition that an infinite sequence cannot contain an infinite subsequence of internal actions, i.e. subsequences of internal actions must be finite. Therefore, we may apply inductively the argument above. So there exist processes $S_0, S_1, ...$ such that $I||O \stackrel{a_0}{\Rightarrow} I_0||O_0 \stackrel{a_1}{\Rightarrow} I_1||O_1 \stackrel{a_2}{\Rightarrow} \ldots$ (with observable actions $a_0, a_1, ...$) is matched by $S||O \stackrel{a_0}{\Rightarrow} S_0||O_0 \stackrel{a_1}{\Rightarrow} S_1||O_1 \stackrel{a_2}{\Rightarrow} \ldots$, where $\forall j.O_j \notin \mathcal{S}$. □

($\Leftarrow$) We prove $I \leq_{testrc} S$ implies $I \leq_{red} S$ by proving the contrapositive. Suppose then that $\exists \sigma \in L^*.R_I(\sigma) \not\subseteq R_S(\sigma)$. Then we need to show $\neg P \vee \neg Q$ where:

- P is the statement $\forall O \in \Omega.\ \ I\ may\ O \Rightarrow S\ may\ O$
- Q is the statement $\forall O \in \Omega.\ \ S\ must\ O \Rightarrow I\ must\ O$

i.e., show $P' \vee Q'$ where:

- P' is the statement $\exists O \in \Omega.\ \ I\ may\ O \wedge\ \neg(S\ may\ O)$
- Q' is the statement $\exists O \in \Omega.\ \ S\ must\ O \wedge\ \neg(I\ must\ O)$

Suppose we have $\sigma =< a_0, a_1, \ldots, a_k >$.

Case (i) $R_S(\sigma) = \emptyset$.

We have $R_S(\sigma) = \emptyset \iff \sigma \notin Tr(S)$. Since $R_I(\sigma) \not\subseteq R_S(\sigma)$, we have that $R_I(\sigma) \neq \emptyset$. Therefore $\sigma \in Tr(I)$. Now define as (the only) successful testers O those which just tack on a successful state after performing σ. That is, those O such that $O \stackrel{\text{def}}{=} a_0; O_0$ and $O_i \stackrel{\text{def}}{=} a_{i+1}; O_{i+1};\ 0 \leq i \leq k-1$, where $\forall i \in \{0, 1, \ldots, O_{k-1}\}.O_i \notin \mathcal{S}, O_k \in \mathcal{S}$, where ";" denotes action prefix, (see figure A.2 for a definition). Then $I||O \stackrel{\sigma}{\Rightarrow} I'||O_k$ for some I'. Hence $I\ may\ O$.

Let σ_0 be the longest initial subtrace of σ such that $\sigma_0 \in Tr(S)$. Then as $||$ requires synchronisation on all observable actions, $S||O \stackrel{\sigma_0}{\Rightarrow} S'||O'$ such that $S'||O' \stackrel{a}{\not\rightarrow}$ for no $a \in L$. Thus O cannot in its interaction with S reach a successful state after performing σ_0, so $\neg(S\ may\ O)$ □.

Case (ii) $R_S(\sigma) \neq \emptyset$

So $\sigma \in Tr(S) \cap Tr(I)$. In general, the behaviour of a process after σ is a tree. To prove this case, we look at the behaviour that is possible after σ. We construct a tester O as follows:

Following the notation above, let $O \stackrel{\text{def}}{=} a_0;O_0$ and let $O_i \stackrel{\text{def}}{=} a_{i+1};O_{i+1}$; $0 \leq i \leq k-1$, where $\forall i \in \{0,1,\ldots,O_k\}.O_i \notin \mathcal{S}$.

Let a_σ be any action belonging to some $A \neq \emptyset$ in $A_S(\sigma) \setminus A_I(\sigma)$ (some such action exists since $R_I(\sigma) \not\subseteq R_S(\sigma)$ and we have that σ is a trace of both). Now define $O_k \stackrel{\text{def}}{=} a_\sigma;O_{k+1}$ where $O_{k+1} \in \mathcal{S}$. Then we have that since in our proposition hypothesis neither S nor O contain no infinite sequences of internal actions and the synchronisation in $||$ is on L, then S *must* O. However, again through the definition of $||$, we have $I||O \stackrel{\sigma}{\Rightarrow} I'||O'$ such that $I'||O' \stackrel{a_\sigma}{\not\Rightarrow}$, so $\neg(I \; must \; O)$. We are done $\Box$.

Notes

1. To illustrate that the condition of strong convergence is necessary (for the first part), consider the following processes (in LOTOS shorthand): `S = a;(b;stop [] c;stop)`, `I = a; I'`, where `I' = b;stop [] c; stop [] i;I'`. Then it can be seen that $I \leq_{red} S$ but $I \not\leq_{testrc} S$ since, for example, if $O = a;succ;$`stop` with $O_k = succ;$`stop` $\in \mathcal{S}$ then S *must* O does not imply I *must* O owing to the possible infinite sequence of the internal `i` after the initial action `a`.

2. It is not the case that $\forall\sigma \in L^*.R_I(\sigma) \subseteq R_S(\sigma)$ implies that after any $\sigma \in Tr(I||O) \cap Tr(S||O)$, there is no transition which S can do in $S||O$ which I cannot do in $I||O$, for consider e.g:

```
S = a;S0 [] a;S1, S0 =b;stop,  S1=c;S2,
I = a;b;stop
O = a;O1, O1=c;Ok
```

   Then $\forall\sigma \in L^*.R_I(\sigma) \subseteq R_S(\sigma)$. But here $S||O \stackrel{<a>}{\Rightarrow} S_1||O_1$ with $S_1||O_1 \stackrel{<c>}{\Rightarrow} S_2||O_k$, whilst $I||O \stackrel{<a>}{\Rightarrow} I_1||O_1$, with $I_1||O_1 \stackrel{<c>}{\not\Rightarrow}$

The small examples used in this chapter may be examined by automated tools. In particular, those illustrating the reduction relation may be checked in the Concurrency Workbench [CPS89] which has the facility of testing the 'may' and 'must' preorder testing relations defined by Hennessy and De Nicola. (Hence the importance of establishing that these preorders constitute an alternative characterisation of this relation).

The next stage is to actually construct (or *derive*) a suite of tests Π, say, which may be used to show whether or not the testing preorder holds. A further consideration is the existence of a single tester, $T(S)$, derived from S, called a *canonical tester* which has the

same testing power as Π, which could save a lot of time and effort. In the next section we show for a certain context the existence and construction of $T(S)$, which provides complete test coverage, including a procedure which enables us to deduce the behaviour of S and I with all possible testers. In general, it may not be possible to do this where the testers are infinite.

5.6 A Canonical Tester for robust conformance in LOTOS

5.6.1 Introduction

In this section, we look at *robust* conformance testing for finite Basic LOTOS specifications. LOTOS allows a semantic interpretation in terms of labelled transition systems, so we may usefully apply the theory of the Observation Framework and the Experimental System, with all its results. The theory of test derivation as a whole has been the subject of much research, but most of the attention has been focused on conformance only. We are motivated by certain requirements that are important for safety-critical applications such as the Flexport protocol for medical devices. In such areas there is the need to insure against unexpected behaviour, i.e. robustness. Thus our approach is to show primarily how canonical testers can be built to provide (in certain circumstances) completeness in validation and, where possible, how detailed information may be obtained. Where overall completeness cannot be realised, smaller contexts may be chosen: for instance, a process may be checked for robustness after executing a given trace.

The work in this section consists of developing a canonical tester which is able to determine whether or not an implementation under test I is a reduction of a specification S. The approach constructs (or *derives*) a test process, guided by criteria similar to those mentioned in section 5.3.1, and which proceeds iteratively to examine the refusals at each state reachable by S. In order to use faithfully the 'may' and 'must' testing strategy on which is based the definition of the relation *testrc*, successful and unsuccessful computations are defined, and the tester designed accordingly.

There are already described in [Bri87, BAL$^+$89], procedures (due to Brinksma) for deriving a canonical tester for the reduction preorder, where it is split into two parts, which test separately for trace inclusion and the **conf** relation respectively. The tester for the **conf** relation is constructed following the observation first expressed by Brookes, Hoare and Roscoe in [BHR84] that processes can be characterised in terms of traces and refusals that satisfy certain properties. Hence, Brinksma's tester makes use of *failure trees* in the construction.

However, this is not the most efficient in the sense that some of its behaviour may be removed and it would still suffice as a tester, though not 'canonical' according to the definition given by Brinksma, whose particular notion of canonical imposes the condition that the traces of the tester for S must equal the traces of S. Such a constraint gives rise to some pleasant properties: any such tester must be unique up to testing equivalence, i.e., if $T1$ and $T2$ are such canonical testers for **conf**, then $T1 \sim T2$, and further, $T(T(S)) \sim S$, i.e the tester for the tester for S gives rise to S, modulo testing equivalence.

Relaxing the constraint on traces enables greater efficiency and larger classes of valid testers. Indeed, in [Led91], Leduc has derived another canonical tester for the **conf** relation that may be seen to start off with the tester tree derived as in [Bri87, BAL$^+$89] and then prunes it so that it becomes minimal in the sense that removing any traces from it would nullify the completeness property that is necessary for it to be a valid tester. A corresponding uniqueness property is expressed in terms of a new equivalence relation $conf\text{-}eq$.

The canonical tester developed here has a similar approach in that refusals are computed for the derivation. However, our presentation is more succinct in that we write down directly an algorithm for the construction of $T(S)$ which does not require any particular characterisation of S.

We also term our tester the *unified tester* since it is simultaneously meant as a tester for both the **conf** and trace preorder relations. It is a simple composition of two testers into one, thereby still allowing us to know in the case of failure whether there is lack of conformance or trace inclusion. Brinksma's canonical tester can be extended to test for $\leq_{red}$ in such a manner by simply adding in some extra branches that test for traces that extend beyond those belonging to S – discussed a little below.

Finally, in view of the wide-ranging checks that have to be carried out for robustness, here efficiency is more of an issue than uniqueness. Thus, we follow the approach of Leduc so that our tester for $\leq_{red}$ shares the same minimality property on traces.

5.6.2 Outline of Methodology

We sketch here the main steps involved in justifying the existence of a canonical tester, through the describing its construction. As an aide-mémoire, refer to Figure A.1 for the LTS notation for LOTOS and Figure A.2 for some axioms and transition rules which specify how a LOTOS behaviour expression may be unfolded action by action. Labelled Transition Systems for LOTOS are discussed in more detail in [BB89]. Note that given

certain assumptions (namely that guards and predicates may be resolved), we can extend the theory for Basic LOTOS to Full LOTOS expressions, where the LTS for LOTOS may be expressed as a 4-tuple as given in the definition of section 2.5.1, such that actions μ are implicitly gates parameterised with values. We note especially that synchronisation in parallel composition requires agreement on gates plus values. Indeed, by making these assumptions we are able to apply conformance testing to our Flexport case study.

Testing for the reduction preorder may be seen as a special instance of property testing as given in [CG93], where IUT is tested with respect to a property according to a *scenario* which makes available a certain selection of events for each transition through constraining the set of events; in our case we stipulate that the specification, implementation and tester all have a finite label set and that the scenario consists of all of these except for the flags. The construction of the unified tester $T(S)$ for a given specification S, has at its heart a directed graph structure (more general than a tree).

The methodology consists of the following steps. First there are preliminary definitions – of what it means for a canonical test to be 'satisfied'; and the scope of our tester. In order to make things computationally feasible, we focus on finite systems: to avoid non-terminating test expansions, we stipulate finite data sorts and bounded event traces. Thus we use a subset of Basic LOTOS for the behaviour expressions Beh_{Proc}, with the set Beh_{Ω} of testers T to be Beh_{Proc} together with a set $\mathcal{F}$ of flags.

Next, the construction is given – a single LOTOS behaviour expression $T(S)$, the canonical tester, is built according to an algorithm. This is defined recursively in terms of S's behaviour after a trace σ, starting from its root node and proceeding down its branches: at each node in $T(S)$, the behaviour is given by $\Gamma(\sigma)$, where:

- we create non-deterministic choices corresponding to each of the reduced acceptance sets. (In fact for each of these we use a kind of minimal subset called a *reduced acceptance set*, defined later.)
- we create a deterministic choice for every action that is not possible for S after σ. Such branches lead to a failed computation.
- to correspond to a state of deadlock in the original specification we create a non-deterministic choice that leads to flag 'success'.

Finally, we provide a number of results that should lead to a formal proof that the algorithm really does generate a canonical tester for the reduction preorder.

The tester may be seen as being constructed in two parts since in S after a given trace, σ, the behaviour may be characterized by the acceptance sets (which leads to testing for the **conf** relation), and the set of actions not possible (which leads to testing for the trace preorder).

Conformance

Consider a specification S and its possible behaviours after a trace σ. Let $N_1, ..., N_n$ be the set of $\tau-$stable nodes reachable after σ. In Lemma 5.6.3 it is shown that the behaviour at these nodes completely determines $R_S(\sigma)$, generating a set R of refusals where $\forall R_i \in R : R_i \subseteq L$. From the definition, we have that the acceptance set consists of component sets which we denote by $\{A_j : j = 1, ..., m\}$, some $m \in \mathbf{N}$. These have the property that $\forall j \in \{1, ..., m\}, \forall S' : S \stackrel{\sigma}{\rightarrow} S'.\exists a \in A_j : S' \stackrel{a}{\rightarrow}$ or $A_j = \emptyset$, i.e. after S performs a trace σ, in each acceptance set there is at least one action by which σ can be extended.

Thus the tester T for *conformance* may be constructed by generating a tree that mirrors in each of its nodes the acceptance sets reachable after any trace σ in S. Then I may be tested for whether or not it is able to synchronise after σ on at least one of the actions a offered by the acceptance set. This test may be forced by offering a choice at the corresponding node N_T in T that consists precisely of a choice of m non-deterministic branches, one for each acceptance set A_j, with each branch prefixed by an internal event before offering every action in A_j.

If there is an unsuccessful computation completed after the empty action at N_T, then, as mentioned above, all actions of some acceptance set are refused, i.e. $\exists j \in \{1, ..., m\}$, $A_j \in R_I(\sigma)$, whence $R_I(\sigma) \not\subseteq R_S(\sigma)$ and we do not have conformance. Conversely, if $\exists j \in \{1, ..., m\}, A_j \in R_I(\sigma)$ then it follows that that there will be deadlock.

The above accounts for compulsory choices – those paths that must be preserved. We need also to consider those actions that are 'optional'/'benign': if one of these is implemented in I, then the subsequent behaviour of this branch must conform. This is achieved simply by offering deterministic branches at the respective node N_T of T for each action that is possible after some trace σ in S, but which is not a member of any acceptance set.

Finally as regards conformance, if S has a termination after σ, then the corresponding acceptance set is empty. The tester captures this by creating a branch that executes a flag action (not in the label set $\mathcal{A}$) to indicate success before termination.

Note that if A_1 and A_2 are sets such that $A_1 \subseteq A_2$, then $\forall a \in A_2 : T||A_1 \stackrel{g}{\not\rightarrow} \Rightarrow \forall a \in A_1 : T||A_1 \stackrel{g}{\not\rightarrow}$, whence we deduce that in the above test for conformance, it suffices to select the smallest sets with respect to containment of (acceptance) sets. This is in

encapsulated in the definition of reduced acceptance sets given below.

Robustness

For robustness, we need to ensure that there is trace inclusion. Similar, to the above, we allow at a given node N_T to offer extra deterministic choices: for each action that would lead to a trace not in S, a branch is created consisting simply of that action followed by a flag to indicate failure, followed by the `stop` action. Hence there will be at least one 'fail' indication if I has such an extra branch.

These tests may be applied for traces of any length by the appropriate use of recursion. For instance, the traces that belong to S may be given recursively, starting with ε, the empty trace and considering in turn traces that are incremented by single actions. As ε belongs to all processes, we can then define for any trace $\sigma \in Tr(S)$, the following recursive procedure $Proc(\sigma)$ to test for trace inclusion, starting with ε:

> $Proc(\sigma)$ Suppose $\sigma \in L^*$. Then, for an action a, either $\sigma \frown < a > \in Tr(S)$ in which case we reiterate and perform $Proc(\sigma \frown < a >)$ or $\sigma \frown < a > \notin Tr(S)$, whence we deduce that all traces that contain $\sigma \frown < a >$ as a prefix are not in $Tr(S)$.

This approach for testing trace inclusion has already been expressed in [BAL+89], which is constructed via another line of reasoning that starts by considering *all* traces in the test suite and successively reducing to the same set as above. Although not stated, this tester for trace inclusion can be integrated within the canonical tester detailed in [Bri87] in a manner similar to that described here. The construction of the canonical tester for **conf** can be extended to test for $\leq_{red}$ by replacing the *standard invertable function* that is defined on failure tree projections T of processes (definition 2.5) by an extension of it that simply adds a summation offering choices of the form $a_j; fail; \texttt{stop}$ for all a_j that are not possible transitions for T.

5.6.3 Derivation of Unified tester

In this section we give the formal derivation of the unified tester for the reduction preorder.

5.6.3.1 Preliminaries

We start by defining what we mean by 'canonical' below for processes in general. This definition is simply characterised as a special case of a test suite and is more general than that in, e.g., [Bri87] which also stipulates a condition on traces.

Definition (Canonical Tester)

Given a specification S and an implementation relation $\leq_R$, a test suite $\{< T, v_T >\}_{T \in \Pi}$ for $Implr_R(S)$ is a *canonical tester* if $\Pi = \{T\}$ for some $T \in \Omega$ (i.e., it contains just one element). Where this is the case, we write $T(S)$ for the canonical tester.

Hence where such a $T(S)$ exists, determining whether an IUT is an implementation of a specification S may be decided simply by establishing whether or not a test is satisfied:

Definition (canonical test satisfaction)

Let S be a specification, from which a canonical tester $T(S)$ has been derived for an implementation relation $\leq_R$. Then we say that an implementation under test IUT *satisfies the test for* $\leq_R$ *with respect to* $T(S)$ iff $v(IUT||T(S)) = pass$, in which case we write IUT *sat* $T(S)$.

In order to construct a canonical tester for $\leq_{red}$ to make use of the testing theory above, we need to supply some definitions to make up the verdict function, making explicit what we mean by successful and unsuccessful computations, and the observation relation corresponding to $\leq_{red}$.

Notation

Let $\mathcal{F} = \mathcal{F}_{succ} \dot{\cup} \mathcal{F}_{diag} \subseteq L$, where $\mathcal{F}$ is a set of flags, with $\mathcal{F}_{succ}$ being indications of success and $\mathcal{F}_{diag}$ being diagnostics labels for failures. We choose to keep these actions reserved for the testers, so we define $\mathcal{A} = L \setminus \mathcal{F}$ and stipulate: $\forall P \in Beh_{Proc} : \mathcal{A}ct(P) \subseteq \mathcal{A} \cup \{\tau\}$, whilst $\forall T \in Beh_{\Omega} : \mathcal{A}ct(T) \subseteq \mathcal{L}$. Finally, let $||$ denote the LOTOS parallel operator where synchronisation must occur on all actions in $\mathcal{A}$.

Definition The *set of computations restricted by a label set* $\mathcal{A}$, denoted $Comp(I,T)_{\mathcal{A}}$ is the set of computations between I and T whose transitions have been constrained by synchronisation on the set of transitions given in $\mathcal{A}$.

Where $\mathcal{A}$ is understood, we choose to drop the suffix. Full LOTOS allows finer granularity in restricting the set of values permissible at gates.

Definition (successful and unsuccessful computations)

Let $x \in Comp(I,T)_{\mathcal{A}}$ be a computation represented by:

$$I \,||\, T \rightarrow I_1 \,||\, T_1 \rightarrow \ldots \rightarrow I_n \,||\, T_n \rightarrow \ldots$$

x is a *successful computation* if $\exists \delta \in \mathcal{F}_{succ}, k \in \mathbf{N} : I_k||T_k \stackrel{\delta}{\Rightarrow}$.

x is an *unsuccessful computation* if it is not successful.

For the sake of simplicity, we define here $\mathcal{F}_{succ} = \{success\}, \mathcal{F}_{diag} = \{fail\}$.

Recall that we have: $I \in Impl_{red}(S)$ if and only if $v(I||T(S)) = pass$ if and only if $I||T(S) \lhd_{red} S||T(S)$. We now define:

$$I||T(S) \lhd_{red} S||T(S) \iff (I\ may\ T(S) \Rightarrow S\ may\ T(S)) \wedge (S\ must\ T(S) \Rightarrow I\ must\ T(S)).$$

where, for any processes $P \in Beh_{Proc}$,

$P\ may\ T$ if $Comp(P,T)_{\mathcal{A}}$ contains a successful computation;

$P\ must\ T$ if $Comp(P,T)_{\mathcal{A}}$ does not contain an unsuccessful computation.

By construction, we will claim that $S\ must\ T(S)$ is always true and that the *may* observation relation always holds. Hence:

Definition (canonical test satisfaction for $\leq_{red}$)

$I\ sat\ T(S)$ for $\leq_{red}$ if and only if $I\ must\ T(S)$.

In words, this means that I is a reduction of S if and only if composing canonical tester $T(S)$ of S with I yields no unsuccessful computations.

Definition (Reduced acceptance sets)

Let $B \in Beh_{Proc}$ and $\sigma \in L^*$. Then a *reduced acceptance set* for B, denoted $\mathring{A}_B(\sigma)$ is one that satisfies:

1. $\mathring{A}_B(\sigma) \subseteq A_B(\sigma)$
2. $\forall A \in A_B(\sigma) : \exists A' \in \mathring{A}_B(\sigma) : A' \subseteq A$.
3. $\forall A_1, A_2 \in \mathring{A}_B(\sigma) : A_1 \not\subset A_2$.

This definition is similar in style to that in [ABe+90], except for the addition of the last condition which ensures the following property:

Lemma 5.6.1 *The reduced acceptance set is unique.*

Proof Suppose $\mathring{A}_1$ and $\mathring{A}_2$ are both distinct reduced acceptance sets for process B and trace σ. Therefore WLOG $\exists A \in \mathring{A}_1$ such that $A \notin \mathring{A}_2$. From condition 1), $A \in \mathring{A}_1 \Rightarrow A \in A_B(\sigma)$. From condition 2) it follows that $\exists A' \in \mathring{A}_2 : A' \subseteq A$. Since from the hypothesis, $A' \neq A$, this implies $A' \subset A$. Now from conditions 1) and 2) we require $\exists A'' \in \mathring{A}_1 : A'' \subseteq A'$. From

the transitive property of set inclusion this implies $A'' \subset A$ which contradicts condition 3) □

Lemma 5.6.2 *Let $P, Q \in Beh_{Proc}$. Let $\sigma \in L^*$. Then $A_P(\sigma) \subseteq A_Q(\sigma) \iff \mathring{A}_P(\sigma) \subseteq \mathring{A}_Q(\sigma)$*

Proof

($\Rightarrow$) Let $A \in \mathring{A}_P(\sigma)$. Then by condition 1) $A \in A_P(\sigma)$ and hence by the hypothesis $A \in A_Q(\sigma)$. Suppose $A \notin \mathring{A}_Q(\sigma)$. Since $A \in A_Q(\sigma)$ we have from condition 2) that $\exists A' \in \mathring{A}_Q(\sigma)$ such that $A' \subseteq A$ (and also with $A' \in A_Q(\sigma)$ – from condition 1). Therefore from our supposition it follows that $A' \subset A$. By condition 3), $A' \notin \mathring{A}_P(\sigma)$. Two cases arise:

1. $A' \in A_P(\sigma)$. This implies in condition 2) that $\exists A'' \in \mathring{A}_P(\sigma)$ such that $A'' \subseteq A' \subset A$ which contradicts condition 3).

2. $A' \notin A_P(\sigma)$. This implies $A_P(\sigma) \not\subseteq A_Q(\sigma)$ which contradicts the hypothesis of the Lemma.

In either case we reach a contradiction, and so the left implication holds □

($\Leftarrow$) Let $A \in A_P(\sigma)$. There are two cases:

1. $A \in \mathring{A}_P(\sigma)$. Therefore by the Lemma hypothesis, we have $A \in \mathring{A}_Q(\sigma)$ and then by set inclusion in the definition of reduced acceptance sets $A \in A_Q(\sigma)$.

2. $A \notin \mathring{A}_P(\sigma)$. Therefore $\exists A' \in \mathring{A}_P(\sigma) : A' \subset A$. By the Lemma hypothesis, $A' \in \mathring{A}_Q(\sigma)$, hence (by set inclusion) $A' \in A_Q(\sigma)$. Finally we apply Lemma 5.5.3 inductively on each element that is in $A \setminus A'$ to give the required result □

The following Lemma allows a simplification in the structure of the summations that make up the definition of the algorithm, stating that at any given node it is the *tau*-stable nodes that govern the refusal sets.

Lemma 5.6.3 *Let S be a state belonging to a process P which has no infinite internal loops. Let $N_1, N_2, ..., N_n$ be the set of τ-stable nodes reachable from S after σ. Then $R_S(\sigma) = \bigcup_{i=1}^{n} R_{N_i}(\varepsilon)$.*

Proof A state is either τ-stable or not, so we need only show that after σ, any refusals in states that offer τ are also refusals of some τ-stable node that is reached after σ.

If $\exists N_x$ such that $S \stackrel{\sigma}{\Rightarrow} N_x$ where $N_x \stackrel{\tau}{\rightarrow}$ then there exists a τ-stable N_y such that $N_x \stackrel{\tau^k}{\rightarrow} N_y$ for some $k \in \mathbf{N}$ since we have the assumption that there are no infinite internal loops. Then we have: $\forall a \in R_{N_x}(\varepsilon).a \in R_{N_y}(\varepsilon)$ since otherwise $N_x \stackrel{a}{\Rightarrow}$ which is a contradiction. This is true for all such N_x and the result follows$\Box$.

5.6.3.2 Construction, Properties and Examples

The algorithm

$T(S)$ is derived from S by defining $T(S)$ to be $\Gamma(\varepsilon)$ where $\forall \sigma \in L^*$:

$$\Gamma(\sigma) := \sum_{b \not\in out_\sigma(S)} b; fail; \mathtt{stop} \quad [] \sum_{\substack{c \in out_\sigma(S): \\ \nexists A \in \mathring{A}_S(\sigma): c \in A}} c; \Gamma(\sigma \frown < c >) \quad []$$

$$\sum_{\substack{A \in \mathring{A}_S(\sigma) \\ : A \neq \emptyset}} \mathbf{i}; \sum_{\substack{a \in L: \\ a \in A}} a; \Gamma(\sigma \frown < a >) \quad [] \quad [IF \quad \mathring{A}_S(\sigma) = \emptyset \quad THEN \quad success; \mathtt{stop}] \tag{5.13}$$

where for a behaviour expression B and trace σ, $out_\sigma(B) := \bigcup_{B \stackrel{\sigma}{\Rightarrow} B'} out(B')$ where $\sigma \in Tr(B)$.

In summary, the first summation term handles traces that lie outside those of S, the second summation handles 'benign' branches whose initial action does not match any initial action in a 'compulsory' branch, the third specifies branches corresponding to non-deterministic choices in S, whilst the last one specifies a successful termination corresponding to the case that S enters a terminal state.

The first Lemma for the algorithm simply states that whenever we have a trace of S, then the Γ function of this trace offers all actions:

Lemma 5.6.4 *Let $S \in Beh_{Proc}$. Then $\forall \sigma \in Tr(S).out(\Gamma(\sigma)) = L$.*

Proof We have that for any $x \in L$, either $x \in out_\sigma(S)$ or $x \not\in out_\sigma(S)$. The first summand of the algorithm offers all actions of the latter case. Noting that $x \in A : A \in \mathring{A}_S(\sigma) \Rightarrow x \in out_\sigma(S)$ by definition of the acceptance set, it also follows that the second and third summations together offer all actions for the former case$\Box$

The second Lemma shows that any trace of S is also a trace of the tester.

Lemma 5.6.5 *Let $\sigma \in Tr(S)$. Then $T(S) \stackrel{\sigma}{\Rightarrow} \Gamma(\sigma)$.*

Proof by induction on $len(\sigma)$

base case: $len(\sigma) = 0$

$\sigma = \varepsilon$. Therefore $T(S) = \Gamma(\varepsilon)$. Clearly $\Gamma(\varepsilon) \stackrel{\varepsilon}{\Rightarrow} \Gamma(\varepsilon)$ $\Box$

Inductive case

Suppose that the result is true for $\sigma \in L^* : len(\sigma) \leq k$, $k \in \mathbf{N}$. Let $\sigma' \in L^* : len(\sigma') = k + 1$. Therefore $\sigma' = \sigma_k \frown < x >$ for some $\sigma_k \in L^* : len(\sigma_k) = k$ and where $x \in L$. By the induction hypothesis, $T(S) \stackrel{\sigma_k}{\Rightarrow} \Gamma(\sigma_k)$. Since $\sigma_k \frown < x > \in Tr(S)$, then $x \in out_{\sigma_k}(S)$. Therefore $R_S(\sigma_k) \neq \mathcal{P}(L)$ so $\mathring{A}_S(\sigma) \neq \emptyset$. And hence $\Gamma(\sigma_k)$ can only perform the x action via one of the second or third summand since these partition $out_\sigma(S)$. In either case we have $\Gamma(\sigma_k) \stackrel{<x>}{\Rightarrow} \Gamma(\sigma_k \frown < x >)$ and the result follows$\Box$.

Thus the result is true for traces of length 0,1,2, ... by induction and we are done. $\Box$

An immediate corollary from the above two Lemmas is that if $\sigma \in Tr(S)$ then $out_\sigma(T(S)) = L$.

The next Lemma shows that whenever the test composition performs a trace of certain length, then it will reach a state in which the Gamma function of the trace is within an internal transition of this state.

Lemma 5.6.6 *Let $S, I \in Beh_{Proc}$. Suppose $\sigma \in Tr(I||T(S))$ and $out_\sigma(T(S)) \notin \mathcal{F}_{diag}$. Then $\forall I' \in Beh_{Proc}, \forall T' \in Beh_\Omega . (I||T(S) \stackrel{\sigma}{\Rightarrow} I'||T'$ implies $\Gamma(\sigma) \stackrel{\varepsilon}{\Rightarrow} T')$.*

Proof The proof is by induction on $len(\sigma)$.

Base case: $len(\sigma) = 0$

We have $\sigma = \varepsilon$. So $I||T(S) \stackrel{\varepsilon}{\Rightarrow} I'||T'$ which implies $T(S) \stackrel{\tau^k}{\rightarrow} T'$ for some $k \in \mathbf{N}$ and hence $\Gamma(\varepsilon) \stackrel{\varepsilon}{\Rightarrow} T'$ $\Box$

Inductive case

Suppose that the result is true for $\sigma \in L^* : len(\sigma) \leq k$, $k \in \mathbf{N}$. Let $\sigma' \in L^* : len(\sigma') = k + 1$. Therefore $\sigma' = \sigma_k \frown < x >$ for some $\sigma_k \in L^* : len(\sigma_k) = k$ and where $x \in L$.

Suppose $I||T(S) \stackrel{\sigma'}{\Rightarrow} I'||T'$. Then looking at intermediate states, we have $I||T(S) \stackrel{\sigma_k}{\Rightarrow} I_k||T_k$ for some I_k, T_k such that $I_k||T_k \stackrel{<x>}{\Rightarrow} I'||T'$. Thus $T_k \stackrel{<x>}{\Rightarrow} T'$, through the definition of $||$, and by the induction hypothesis we have: $\Gamma(\sigma_k) \stackrel{\varepsilon}{\Rightarrow} T_k$. Hence $\Gamma(\sigma_k) \stackrel{<x>}{\Rightarrow} T'$. Now instantiating in the definition of the algorithm gives,

$$\Gamma(\sigma_k) := \sum_{b \notin out_{\sigma_k}(S)} b; fail; \mathtt{stop} \quad [] \quad \sum_{\substack{c \in out_{\sigma_k}(S): \\ \nexists A \in \mathring{A}_S(\sigma_k): c \in A}} c; \Gamma(\sigma \frown < c >) \quad []$$

$$\sum_{\substack{A \in \mathring{A}_S(\sigma_k): \\ A \neq \emptyset}} \mathbf{i}; \sum_{\substack{a \in L: \\ a \in A}} a; \Gamma(\sigma_k \frown < a >) \quad [] \quad [IF \quad \mathring{A}_S(\sigma_k) = \emptyset \quad THEN \quad success; \mathtt{stop}]$$

Suppose the tester performs an action from the first summand. Then after σ the tester can perform a 'fail' action which contradicts the Lemma hypothesis.

We are left with either:

- $\Gamma(\sigma_k) \stackrel{<x>}{\Rightarrow} \Gamma(\sigma_k \frown < x >)$

  or

- $\Gamma(\sigma_k) \stackrel{<x>}{\Rightarrow} \Gamma'$ such that $\Gamma(\sigma_k \frown < x >) \stackrel{\tau}{\rightarrow} \Gamma'$.

Hence we deduce that $\Gamma(\sigma_k \frown < x >) \stackrel{\varepsilon}{\Rightarrow} T'$ as required □

Thus the result is true for traces of length 0,1,2, ... by induction and we are done

□

These Lemmas should be sufficient to show the main conjecture, i.e. that if S is a specification with finite behaviour and $T(S)$ constructed as above, then $T(S)$ is a canonical tester for the reduction preorder. Further, the construction should enable the simple deduction of the following:

1. (Decomposition property)

   - I satisfies trace inclusion iff $T(S)$ does not record a 'fail' in the test composition;
   - I satisfies the **conf** relation iff all terminations in the test composition record either 'fail' or 'success'.

2. (Minimality property with respect to traces)

   Let $S \in Beh_{Proc}$, with canonical tester $T(S)$ (for reduction preorder). Let T' be another canonical tester. Then $Tr^*(T') \supseteq Tr^*(T(S))$, where $\forall B \in Beh_{Proc}$, $Tr^*(B) := \{\sigma \in Tr(B) : \sigma \in \{L \setminus \mathcal{F}\}^*\}$.

Examples

Example 1 `S = i;a;stop [] b;stop`. $L = \{a, b, c\}$.

Then $R_S(\varepsilon) = \{\{b, c\}, \{b\}, \{c\}, \emptyset\}$

Therefore, $A_S(\varepsilon) = \{\{a, b, c\}, \{a, b\}, \{a, c\}, \{a\}\}$

Hence $\mathring{A}_S(\varepsilon) = \{\{a\}\}$.

$R_S(< a >) = R_S(< b >) = \mathcal{P}(L)$. Hence $\mathring{A}_S(< a >) = \mathring{A}_S(< b >) = \emptyset$

... hence ...

```
T(S) = i;a;TL [] b;TL [] c;fail;stop
```

where

```
TL = succ;stop [] a;fail;stop [] b;fail;stop [] c;fail;stop
```

Example 2 `S = i;a;stop [] i;(c;stop [] d;stop) [] a;stop [] c;stop`.

$L = \{a, b, c, d\}$.

Then $R_S(\varepsilon) = \{\{b, c, d\}, \{b, c\}, \{b, d\}, \{c, d\}, \{b\}, \{c\}, \{d\}, \emptyset, \{a, b\}, \{a\}\}$

Therefore, $A_S(\varepsilon) = \{\{a, b, c, d\}, \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{a, c\}, \{a, d\}\}$

Hence $\mathring{A}_S(\varepsilon) = \{\{a, c\}, \{a, d\}\}$.

$R_S(< a >) = R_S(< c >) = R_S(< d >) = \mathcal{P}(L)$. Hence $\mathring{A}_S(< a >) = \mathring{A}_S(< c >) = \mathring{A}_S(< d >) = \emptyset$

... hence ...

```
T(S) = i;(a; Tm [] c;Tm) [] i;(a;Tm [] d;Tm) [] b;fail;stop
```

where

```
Tm = succ;stop [] a;fail;stop [] b;fail;stop [] c;fail;stop [] d;fail;stop
```

Example 3 `S = i;(a;stop[] b;stop)[] i;(c;stop[] d;stop)[] a;stop[] c;stop`. $L = \{a, b, c, d\}$.

Then $R_S(\varepsilon) = \{\{c, d\}, \{c\}, \{d\}, \emptyset, \{a, b\}, \{a\}, \{b\}\}$

$\Rightarrow A_S(\varepsilon) = \{\{a, b, c, d\}, \{a, b, c\}, \{a, b, d\}, \{a, c, d\}, \{b, c, d\}, \{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}\}$

Hence $\mathring{A}_S(\varepsilon) = \{\{a, c\}, \{a, d\}, \{b, c\}, \{b, d\}\}$.

$\forall x \in L. R_S(< x >) = \mathcal{P}(L)$. Hence $\forall x \in L. \mathring{A}_S(< x >) = \emptyset$

... hence ...

```
T(S) = i;(a; Tm [] c;Tm) [] i;(a;Tm [] d;Tm) [] i;(b;Tm [] c;Tm) [] i;(b;Tm
[] d;Tm)
```

where, as above,

```
Tm = succ;stop [] a;fail;stop [] b;fail;stop [] c;fail;stop [] d;fail;stop
```

Example 4 To illustrate computations of $T(S)$ and I terminating with/without raising flags, consider: `S = a;b; stop`, `I1 = a; stop`, `I2 = a; c; stop` with $\mathcal{A} = \{a, b, c\}$. Then

```
T(S) = i; a;(i;b;TL [] a;fail;stop [] c;fail;stop) [] b;fail;stop [] c;fail;stop
```

where

```
TL = succ;stop [] a;fail;stop [] b;fail;stop [] c;fail;stop
```

In this case, $T(S)||I1 = a;$`stop` and $T(S)||I2 = a; fail;$`stop` – the former fails to conform, whilst the latter fails trace inclusion.

5.6.4 A Special Case

Where the refusals of S can be expressed simply, then this may lead to simplification in the tester expression, as is the case resulting from the following Lemma.

Lemma 5.6.7 *Let $S \in Beh_{Proc}$ and let $\sigma \in L^*$, where $\mathcal{I}$ is some non-empty index set. Suppose $R_S(\sigma) = \bigcup_{i\in\mathcal{I}} \mathcal{P}(L \setminus a_i)$. Then, we have:*

1. $\mathring{A}_S(\sigma) = \{\bigcup_{i\in\mathcal{I}}\{a_i\}\}$.
2. $\{out_S(\sigma)\} = \mathring{A}_S(\sigma)$

Proof Let $X = \bigcup_{i\in\mathcal{I}}\{a_i\}$.

1. We show first that $X \in A_S(\sigma)$:

   Suppose $X \in R_S(\sigma)$. Then since $R_S(\sigma) = \bigcup_{i\in\mathcal{I}} \mathcal{P}(L \setminus a_i)$, we have that $\exists a_i \in X : X \in \mathcal{P}(L \setminus a_i)$. This gives an immediate contradiction (since $a_i \in X$ but $a_i \notin \mathcal{P}(L \setminus a_i)$). Therefore $X \notin R_S(\sigma)$. However $X \in \mathcal{P}(L)$, so $X \in A_S(\sigma)$.

   We now show that $\forall A \in A_S(\sigma) : A \supseteq X$. Suppose this is not the case, so $\exists A \in A_S(\sigma), x \in X : x \notin A$. But then we have $A \in \mathcal{P}(L \setminus x)$. Hence $A \in R_S(\sigma)$ — contradiction.

   Now let $Z = \{X\}$. Then it can be seen that this is in fact $\mathring{A}_S(\sigma)$ since it satisfies all three of the requirements in the definition of reduced acceptance sets.□

2. From the first part we need only show that $out_S(\sigma) = \bigcup_{i\in\mathcal{I}}\{a_i\}$.

   Suppose that $a \in out_S(\sigma) \wedge a \notin X$. Then we have $\forall a_i : i \in \mathcal{I} : \{a\} \in \mathcal{P}(L \setminus a_i)$. Then since $R_S(\sigma) = \bigcup_{i\in\mathcal{I}} \mathcal{P}(L \setminus a_i)$any set in $\mathcal{P}(L)$ containing $'a'$ must be a refusal set in which case $a \notin out_\sigma(S)$, which is a contradiction. Therefore $out_\sigma(S) \subseteq X$. Hence it follows that $\forall x \in X : x \in out_\sigma(S)$ since otherwise $L\{x\} \in R_S(\sigma)$ which is a contradiction □

These Lemmas lead to considerable simplification in the expression for the canonical tester (equation 5.13) for this particular case:

$$\Gamma(\sigma) = \sum_{b \in (L \setminus out_\sigma(S))} b; fail; \texttt{stop} \quad [] \sum_{c \in (out_\sigma(S) \setminus X)} c; \Gamma(\sigma \frown < c >) \quad [] \quad \mathbf{i}; \sum_{a \in X} a; \Gamma(\sigma \frown < a >) \tag{5.14}$$

where $X = \bigcup_{i \in \mathcal{I}} \{a_i\}$

We now need to know structures which make use of the simplification. One such is as follows.

Suppose that $\forall N : S \stackrel{\sigma}{\Rightarrow} N : N = \sum_{i \in \mathcal{J}} \tau^k; a_i; N'$ for some $k \in \mathbf{N}, N' \in Beh_{Proc}$, where $\emptyset \neq \mathcal{J} \subseteq \mathcal{I}$. Then we do indeed have $R_S(\sigma) = \bigcup_{i \in \mathcal{I}} \mathcal{P}(L \setminus a_i)$. In the following section we use this particular case to help in the actual construction of simpler testers reflecting the simple tester expression 5.14.

5.7 Implementation in a subset of Full LOTOS

We provide an implementation of the algorithm in a restricted subset of Full LOTOS, in which actions are parameterised with finite data sets and where predicates are resolved. This allows for a simple injection between behaviour expressions given as LTSs and the corresponding LOTOS specification, preserving the tree structure and the relationships between nodes. The algorithm for this subset of Full LOTOS may then be expressed as follows.

The algorithm in Full LOTOS subset

$T(S)$ is derived from S by defining $T(S)$ to be $\Gamma(\varepsilon)$ where $\forall \sigma \in L^*$:

$$\Gamma(\sigma) := \sum_{i=1}^{n_1} \sum_{\substack{k=1 \\ g_i!v_{i,k} \notin out_\sigma(S)}}^{m_1} g_i!v_{i,k}; fail; \texttt{stop} \quad []$$

$$\sum_{i=1}^{n_2} \sum_{\substack{k=1: \\ g_i!v_{i,k} \in out_\sigma(S) : \neg \alpha(g_i!v_{i,k})}}^{m_2} g_i!v_{i,k}; \Gamma(\sigma \frown < g_i!v_{i,k} >) \quad []$$

$$\sum_{\substack{A \in \mathring{A}_S(\sigma) \\ : A \neq \emptyset}} \mathbf{i}; \sum_{i=1}^{n_3} \sum_{\substack{k=1 \\ g_i!v_{i,k} \in out_\sigma(S) : \{g_i!v_{i,k}\} \in A}}^{m_3} g_i!v_{i,k}; \Gamma(\sigma \frown < g_i!v_{i,k} >) \quad []$$

$$[IF \quad \mathring{A}_S(\sigma) = \emptyset \quad THEN \quad success; \texttt{stop}] \quad --(*)$$

where for $x \in L^*$, $\alpha(x)$ is the statement $\exists A \in \mathring{A}_S(\sigma) : \{x\} \in A$.

The algorithm is implemented directly following the shape of the algorithm, using iteratively process instantiation. An enhancement is made to the test for trace inclusion, whereby diagnosis on failure is given according to whether or not there is synchronisation on gates or values; the 'fail' action is effectively parameterised to reflect this.

5.7.1 Main procedure

1. First the algorithm must be expressed to account for parameterised actions in the form corresponding to $(*)$, so all predicates and guards must be resolved and the specification flattened. One may use a tool such as *SMILE* [EW93] to do this.

2. The data types of S are combined into one super data type, whose sort we give below as `Data`. This data type requires the sorts for the Natural Numbers, the Booleans and Set. We have used the pre-defined types given in an updated version of the ISO library. Equations are then defined fully for the operations lt, eq, and ne.

3. The tester itself may then be constructed as a LOTOS specification whose body is given by a single process `Tester` which consists of a structure corresponding to that in (*).

Now we define the tester as:

`specification CanonicalTesterReduction [`$< all\ gates >$`, fail_data, fail_gate, fail_both, success] : noexit`

`behaviour Tester[`$< all\ gates >$`,fail_data,fail_gate,fail_both,success](<>)`

`where`

`process Tester [ `$< all\ gates >$`, fail_data, fail_gate, fail_both, success] (`σ`:string): noexit: =`

$$\sum_{i=1}^{n_1} \sum_{\substack{k=1 \\ g_i!v_{i,k} \notin out_\sigma(S)}}^{m_1} ([\exists x : g_i!x \in out_\sigma(S)] \rightarrow \texttt{fail\_data;stop} \quad [] \quad [\exists h : h!v_{i,k} \in out_\sigma(S)] \rightarrow \texttt{fail\_gate;stop} \quad []$$

$$[\nexists h : h!v_{i,k} \in out_\sigma(S) AND\ \nexists x : g_i!x \in out_\sigma(S)] \rightarrow \texttt{fail\_both; stop}) \quad []$$

$$\sum_{i=1}^{n_2} \sum_{\substack{k=1: \\ g_i!v_{i,k} \in out_\sigma(S): \neg\alpha(g_i!v_{i,k})}}^{m_2} g_i!v_{i,k}; \texttt{Tester}[< all\ \ gates >, fail\_data, fail\_gate, fail\_both, success]$$

$$(\sigma + < g_i!v_{i,k} >) \quad []$$

$$\sum_{\substack{A \in \mathring{A}_S(\sigma) \\ : A \neq \emptyset}} \mathbf{i}; \sum_{i=1}^{n_3} \sum_{\substack{k=1 \\ g_i!v_{i,k} \in out_\sigma(S): \{g_i!v_{i,k}\} \in A}}^{m_3} g_i!v_{i,k}; \texttt{Tester}[< all\ \ gates >, fail\_data, fail\_gate, fail\_both, success]$$

$$(\sigma + < g_i!v_{i,k} >) \quad []$$

$$[\mathring{A}_S(\sigma) = \emptyset] \rightarrow success; \texttt{stop}]$$

```
        endproc (* Tester *)
endspec (* CanonicalTesterReduction *)
```

Notes

1. Within the body of the process `Tester`, there are recursive calls to `Tester`. It may be possible to encode the behaviour of S within the data types and take this as a process instantiation. Otherwise, when the behaviour of S is entered manually, this 'call' should be substitued by the body of the `Tester` process. Likewise for the preconditions.

2. For the diagnosis, there will be some superfluous flags: if there is a 'fail-both' indication, then there will also be a 'fail-data' or 'fail-gate' indication.

5.7.2 Special Case

We can make use of the special case expressed in section 5.6.4. This further simplification allows a straightforward application to the Flexport case study: in the next chapter it is described how the tester is actually derived from a specification based on a Message Sequence Chart.

We assume that the LTS for a specification S has a tree structure as given by:

$$S = \sum_{i=1}^{n} \sum_{k=1}^{m_i} \tau^{p_i}; g_i!v_{(i,k)}; S_{(i,k)} \qquad [] \qquad \sum_{j \in J} \tau_j^n; \texttt{stop}, \tag{5.15}$$

In this case, we implement the algorithm by an injection between behaviour expressions and the corresponding LOTOS specification, preserving the tree structure and the relationships between nodes. To implement the algorithm neatly, we use iteratively process instantiation of a process `TestEvent` within a newly defined process `tester`, which itself is not recursive. The `TestEvent` process gives diagnosis on failure of trace inclusion.

For the case that for all nodes N either $m_i = 1$ and $p_i = 0$ or $p_i > 0$, we may express the tester process as

```
process Tester [ < gates >, < failure gates >, success] :  noexit :  =
        < q(N) > (TestEvent[g1,< failure gates >,fail_data,fail_gate,fail_both,
success]
        (v(1,1), Insert(v(1,1), Insert v(1,2), ...  Insert_1(v(1,m1),{} ) ...  ))
        >>
             < q(N1 > TestEvent[ ...  ]  ( ...  )
        >> .  .  .  >> success; stop
        )
        []
        TestEvent[g2,< failure gates >,fail_data,fail_gate,fail_both,success]
        (v(2,1), Insert(v(2,1), Insert v(2,2), ...  Insert_1(v(2,m2),{} ) ...  ))
        >> .  .  .  >> success; stop
        []
        .
        .
        .
        []
        TestEvent[gn,< failure gates >,fail_data,fail_gate,fail_both, success]
        (v(n,1), Insert(v(n,1), Insert v(n,2), ...  Insert_1(v(n,mn) ,{}) ...  ))
        >> .  .  .  >> success; stop
             )
endproc (* Tester *)
```

where for a node N_x, $q(N_x)$ is a kind of rewrite operation: replace this by an empty

string if $\#out(N_x) = 1$ or else by internal action prefix '**i**;' if $\#out(N_x) > 1$. If $out(N_x) = \emptyset$ then write '`success;stop`'.

Note: `Insert`, an operation of the `set` data type, adds an element to a set.

5.7.2.1 LOTOS 'procedure' `TestEvent`

We define the process `TestEvent` which 'receives' as gate labels a correct gate g plus other (incorrect) gates $f1, f2, ...$ and as a value parameter the correct data z. This process offers for any value `zz:Data`, all actions possible of form $gg!zz$, where gg is any of the supplied gates. If the correct action (gate and data) is specified then the process successfully terminates via `exit`. Otherwise, depending on IUT, there will either be immediate deadlock arising when IUT lacks a branch, or undesirable synchronisation on some other action of IUT, not equal to $g!z$. After such an action one of three flags are raised, immediately before (unsuccessful) termination:

1. `fail_data` — gate was matched, but the data was not matched;
2. `fail_gate` — data was matched, but the gate was not matched;
3. `fail_both` — neither the data nor the gate were matched.

By choosing to supply to `TestEvent` all gates and by encapsulating all possible data values in the sort `data`, this process offers all actions possible. When the test process is specified in this way, it becomes a test of robustness.

Special care is required to ensure that the equations for sort `data` are completely defined otherwise there may be deadlock without diagnosis.

```
process TestEvent [g,f1,...,fail_data,fail_gate,fail_both](e:Data,
z:Pset) : exit :=

  choice zz: Data []

   [zz eq e] ->  (* Correct Data *)

    (
     g!e; exit  (* valid input  *)
    []
     f1!zz; fail_gate; stop
    .
    .    (* Incorrect gates *)
```

```
   .
   )
 []
  [zz NotIn z] -> (* Incorrect Data *)

   (
    g!zz; fail_data; stop  (* but Correct Gate *)
   .
    f1!zz; fail_both; stop
   .
   .   (* Incorrect data and gates *)
   .
   )

endproc (* TestSet *)
```

Note: It is possible and convenient to define another process simpler than TestEvent, which caters for the special case when there is no choice in S.

5.7.3 Observations

The complexity of the algorithm in terms of a simple LTS setting and BASIC LOTOS is not severe for finite transition systems. In general, the main structure of $T(S)$ is actually more simple than that of S: the iterative design of the algorithm generates a tree-like structure for $T(S)$ consisting essentially just of traces from S; further, $T(S)$ allows only one path for a given trace. The extra bits of $T(S)$ are the short failure branches that show lack of robustness (or trace inclusion).

The main computational concerns are the determination of the acceptance sets after a given trace. This depends upon the size of the label set and the intricacy of S in terms of the number of different ways of performing a certain trace. For our subset of Full LOTOS, additional concerns are in being able to reduce the specifications to the required form, resolving any predicates. In general, this may only be practicable if a tool such as SMILE's conversion to EFSM can perform the reduction automatically.

In the special case that S is a tree, then the computations for the acceptance set are very simple in nature and manual translation for at least small size specifications is realistic.

This has enabled the implementation of the algorithm described in sections 5.7.1

and 5.7.2, for which general expressions have been given." the special case which we have tested on one or two simple examples, which are illustrated for Flexport in the next chapter. At present, the tester is generated manually and it is evident that it soon becomes quite unwieldy. However, it should not be too difficult to generate the canonical tester specification automatically from the acceptance sets of S. As already noted, a further refinement would be to code the behaviour of S as a data type, in which case the tester would have a very neat expression.

The simplified case shows how the implementation of the unifed tester can be enhanced in structure by defining a special process that handles trace inclusion, effectively allowing this test to be abstracted out from the main tester process.

5.8 Discussion: Alternative notions of conformance

It is worth considering still other relations which provide different formal interpretations of the notion of implementation, conformance and reduction. Some experimental work has been conducted for a novel relation, *cut*, denoted $\leq_{cut}$, that we argue may serve as a useful alternative to $\leq_{red}$ as an implementation relation that can be used for reducing specifications, and which can be tested. In this setting, robust and conforming implementations are considered to be those processes constructed simply through reducing the specification by allowing non-deterministic branches to be optionally incorporated in the implementation, whilst preserving 'benign' choices. This is in accord with the notion that reduction decreases the amount of non-determinism. Hence, this relation keeps the same notion of robustness, but differs in the notion of conformance (for which a new relation $\mathbf{conf_d}$ is defined).

The fact that the *cut* relation is somewhat different is evidenced by the existence of IUTs that will satisfy $\leq_{cut}$, but will either fail to be a reduction or to satisfy the testing pre-order, or possibly both. Conversely, there are IUT's that satisfy the reduction or testing preorder, but not the *cut* relation. Another difference to note between $conf$ and $conf_d$ is that the latter allows as valid implementations path extensions; these fail the *cut* relation only through failing trace inclusion.

Once again, we look at conformance testing for finite Basic LOTOS specifications, and we show the existence of a single test process (a *canonical tester*) which is able to determine whether or not an implementation under test I is a cut of a specification S. We also term this particular canonical tester the *unified tester* since it is simultaneously a tester for both $\mathbf{conf_d}$ and the trace preorder relations.

Owing to the simpler nature of the tester, for a certain class of Full LOTOS behaviour expressions, we are able to refine the tester to provide useful information about an IUT's (mis-)behaviour. This is achieved by enriching the set of observers so that it gives diagnostics in the case that a test fails through failure to synchronise. Further, by appropriate definition of the data types, we are able to realize the tester in a fairly simple way as a specially designed LOTOS (text) definition.

Examples

Example 1 Any deterministic choice must be preserved in the implementation. Further, if a choice is implemented, at least one of its branches must be implemented fully:

Let `S = a;b;stop [] c;d;stop`, then we wish to allow the following to be a valid implementation: `I1 = S`, but not `I2 = a;b;stop`, `I3 = c;d;stop` or I4 = a;b;stop [] c;stop

Example 2 In a composite choice with deterministic and non-deterministic elements, any non-deterministic branch may be omitted:

Let `S = a;b;stop [] i;c;d;stop [] i;b;stop`, then we wish to allow the following to be valid implementations: `I1 = S`, `I2 = a;b;stop [] c;d;stop`, but not `I3 = c;d;stop`.

Example 3 If there is offered only non-deterministic choices, at least one branch must be implemented:

Let `S = a;b;stop [] a;c;stop`, then we wish to allow the following to be valid implementations: `I1 = S`, `I2 = a;b;stop` and `I3 = a;b;stop`, but not `S = stop`

(similarly for non-determinism arising through internal action prefixes.)

5.8.1 Comparison between two notions of conformance

Since in any parallel composition, the semantics of LOTOS give pre-emptive power to non-deterministic branches, more effort has to be made in the design of this tester. In particular, successful and unsuccessful computations need to be defined in terms of the tester passing through more than just terminal states. We illustrate our ideas through the development of an example below, where we derive a tester for $\mathbf{conf_d}$.

Consider `S=c;(a;stop [] i;b;stop)`. Under $conf_d$, we have that the following processes conform: `I1=c;a;stop`, and `I2=S`, but not `I3 = c;i;b;stop`. In our tester, we give priority to the benign choices, but in deriving a provisional $T'(S)$ from S, a simple swap between non-deterministic choices and deterministic ones is not adequate: if `T'(S) = i; c; i; a;succ;stop) [] b; succ; stop` then $S||I2$ deadlocks after both $T'(S)$ and $I2$

perform internal actions.

To allow for pre-empting, we wish to catch this behaviour in our tester: for those branches in $T(S)$ that correspond to the compulsory choices offered in S, we do indeed specify branches with internal action prefix followed by the respective matching actions. Once such a branch is taken in $I||T(S)$, I may still perform internal actions, whence there will be deadlock if the next action in I is not matched in $T(S)$. However, if there is always included such an action in $T(S)$, then choosing I to be any optional branch in S will result in a successful computation, whence $I3$ would apparently conform.

A solution to handling the pre-emptive problem is the introduction of a new type of termination 'trip' which may be reached when following a path that is only optional in S. Special 'trip' branches are allocated in $T(S)$ whenever there are both optional and compulsory choices at a node S_N of S: in $T(S)$, for each choice compulsory at S_N, there is generated an internalised branch with corresponding action; after each such action, a 'trip' branch is defined for each action that is initial to the optional branches of S_N. Thus, in our example, we define `T(S) = i;c; (b;succ;stop [] i; T1)` where `T1=a;succ;stop [] b;trip;stop`.

How should we use these results in determining verdicts? As before, it is clear that we should award the first test composition the verdict 'pass'. However, both $I2$ and $I3$ have 'succ' and 'trip' terminations after performing traces that start with an initial 'c' action. They are distinguishable by the behaviour at the node from which the 'trip' branch was executed, by considering whether or not the tester in its complete set of test runs could have performed at the node containing the 'trip' branch, some other action not leading to the 'trip' action. This is described in the next section.

A further consideration is the duplication of traces: we need to allow for cases such as `S = i;a;b;c;stop [] a;b;d;stop` where we require that `I1=a;b;d;stop` conforms, but `I1=a;b;c;stop` does not. In this instance, we note that since a given trace in the tester cannot dictate the path taken by IUT when it performs such a trace, more than one path for that trace in the tester would be superfluous, hence whenever there are traces in S that have a common prefix trace, we combine them into one, and then, where there is a difference in observable actions, the construction of $T(S)$ takes account of the structure of S.

5.9 Conclusions

In this chapter we have presented a systematic treatment of the notions of robustness and conformance in the context of process algebras, clarifying the relationships

between already existing work, specifically the Observation Framework due to Brinksma et al and the Experimental System of Hennessy and De Nicola. We have developed what we believe is a new canonical tester for the reduction preorder that is efficient in expression. Finally, we have shown how the tester may be illustrated for a subset of Full LOTOS.