University of Newcastle upon Tyne
School of Computing Science

The Theory and Practice of Refinement-After-Hiding

by

Jonathan Burton

PhD Thesis

June 2004

Contents

List of Figures

Acknowledgements

The work which appears in chapter 7 grew out of initial discussions with Steve Paynter and Neil Henderson and benefited from further conversations with Ian Clark and Fei Xia. Thanks are due to the ASL group at Newcastle who listened to talks on earlier versions of some of the work which appears in this thesis. I would also like to thank my supervisor, Maciej Koutny, for the time and effort he put into collaborating on early work, providing valuable advice and guidance and then reading drafts of this thesis. Finally, I would like to thank Cathy for her support and understanding during the long hours it took to complete this thesis.

Abstract

In software or hardware development, we take an abstract view of a process or system — i.e. a specification — and proceed to render it in a more implementable form. The relationship between an implementation and its specification is characterised in the context of formal verification using a notion called *refinement*: this notion provides a correctness condition which must be met before we can say that a particular implementation is correct with respect to a particular specification. For a notion of refinement to be useful, it should reflect the ways in which we might want to make *concrete* our *abstract* specification. In process algebras, such as those used in [28,50,63], the notion that a process Q implements or *refines* a process P is based on the idea that Q is more deterministic than P: this means that every behaviour of the implementation must be possible for the specification.

Consider the case that we build a (specification) network from a set of (specification) component processes, where communications or interactions between these processes are hidden. The abstract behaviour which constitutes these communications or interactions may be implemented using a particular protocol, replication of communication channels to mask possible faults or perhaps even parallel access to data structures to increase performance. These concrete behaviours will be hidden in the construction of the final implementation network and so the correctness of the final network may be considered using standard notions of refinement. However, we cannot directly verify the correctness of *component* processes in the general case, precisely because we may have done more than simply increase determinism in the move from specification to implementation component. Standard (process algebraic) refinement does not, therefore, fully reflect the ways in which we may wish to move from the abstract to the concrete at the level of such components. This has implications both in terms of the state explosion problem and also in terms of verifying in isolation the correctness of a component which may be used in a number of different contexts.

We therefore introduce a more powerful notion of refinement, which we shall call *refinement-after-hiding*: this gives us the power to approach verification *compositionally* even though the behaviours of an implementation component may not be contained in those of the corresponding specification,

provided that the (parts of the) behaviours which are different will be ***hidden*** in the construction of the final network. We explore both the theory and practice of this new notion and also present a means for its automatic verification. Finally, we use the notion of refinement-after-hiding, along with the means of verification, to verify the correctness of an important algorithm for asynchronous communication. The nature of the verification and the results achieved are completely new and quite significant.

Chapter 1

Introduction

1.1 Need for formal verification

In software or hardware development, we take an abstract view of a process or system — i.e. a specification — and render it in a more implementable form. An integral part of the development process is then gaining confidence that the implementation derived is valid with respect to its specification. The most common way of doing this is using testing, whereby inputs to the implementation are generated and the resulting behaviour is compared with what the specification requires on those inputs. It may often be difficult to test processes exhaustively, however, because of the large state spaces which they may have. An alternative approach is to employ *formal verification*: this is the process of showing, using mathematical methods, that an implementation meets its specification.

Formal verification is of particular interest when we come to develop *concurrent systems*, which are systems composed of components operating in parallel with each other. This is for a number of reasons. To begin with, the behaviours of such systems are extremely complex due to the many different possible interactions in which their components can engage. This means that concurrent systems can be very difficult to design, while the importance of guaranteeing their correctness is increased by the fact that they are being used more and more in safety critical applications. Yet testing is of limited value here: the non-determinism inherent in the executions of such systems, along with the state explosion from which they may suffer, means that it is impossible to test more than a small proportion of their possible behaviours.

1.1.1 Constituent elements of a verification method

Four elements are necessary in order to carry out formal verification.

These are:

- A means of representing the specification.
- A means of representing the implementation.
- A notion of what it means for the implementation to be correct with respect to the specification.
- A method for checking whether or not the implementation is, in fact, correct.

Process algebras, such as those in [28,50,63], are intended for the description and verification of concurrent systems, coming equipped with a language for process description — the same language is used to describe both specification and implementation processes — and a semantics which ascribes meaning to processes expressed in that language. In general, although different process algebras take different approaches, that an implementation process is correct with respect to a particular specification process means that the implementation is more deterministic than the specification. (We say that the implementation *refines* the specification according to a particular notion of *refinement.*) Moreover, since the primary focus is on the interactions which occur between concurrent processes, the semantics abstracts away from the internal behaviour of processes, focusing only on their externally observable behaviour.

1.2 Types of "reification"

We shall refer to the process of transforming an abstract specification into an (more concrete) implementation as *reification.*[1] In order to classify the limitations of existing methods of refinement in process algebras and to enable comparison of the work presented in this thesis with related work, it is necessary to present a taxonomy of types of reification. The purpose of this taxonomy is simply to provide a useful framework for discussion and other taxonomies may legitimately be proposed. Note that each type of reification may also be classified as either *internal* or *external*: this distinction is useful due to the distinguished role played by external behaviours in process algebraic semantics. The effects of external reification are directly visible to any observer since they alter the (externally visible) behaviours of the process; those of internal reification may be observed only indirectly.

- *Data reification.* Data abstraction is a useful tool in system specification: for example, at the specification level we may store some data

[1]This will avoid a confusing multiple use of the term *refinement*.

as a set because we are not interested in duplicates or in any ordering information relating to the individual data items. In any final implementation, we shall need to represent this set in a structured way, perhaps as a tree, and so it will be necessary to show that the concrete data representation is a correct implementation of the abstract data representation. The standard approach taken is to regard two different data representations as equivalent if they induce the same external behaviours. For example, data reification in [30] is characterised using a homomorphism and a similar approach is taken in VDM ([33]) and Z ([70]). In process algebras, data structures are represented as processes or parameters to processes and the behavioural view of data types is forced on us by the interest only in external behaviours. Data reification, by its definition, is classified as internal reification.

- *Behaviour decomposition.* Behaviour abstraction is fundamental to the process of producing a specification. In the move from a specification to an implementation we will often wish to implement abstract, high-level actions and behaviours at a lower level of detail and in a more concrete manner. For example, an abstract communication event may be implemented using a series of events which implement a particular communication protocol (see example in section 1.4.1 below). Behaviour decomposition can occur both internally and externally.

- *Relaxation of atomicity.* The complexity introduced by concurrency may be managed at the specification stage by assuming the atomic execution of certain sequences of events: for example, we may assume that database read and write transactions are executed atomically — i.e. the executions of distinct transactions do not overlap — whereas this will not be guaranteed at the implementation level. We may then reason about the (less complex) sequential specification and only later introduce concurrency. Relaxation of atomicity may also arise through the use of behaviour decomposition, since the respective implementations of discrete abstract events may interleave at the implementation level. Relaxation of atomicity may occur both internally and externally.

If these different types of reification are to be employed in system development, we need corresponding notions of refinement in order that we may verify the correctness of any particular implementation against the relevant specification. Standard notions of process algebraic refinement can be used to show correctness in all cases that reification is internal, since the semantics is concerned only with externally visible behaviours. However, it is not possible in the general case to use such notions to verify correctness when an implementation has been derived from a specification using *external* behaviour decomposition and/or *external* relaxation of atomicity, even though such types of reification are an integral part of the process of stepwise development.

This has the following consequences. Consider a (specification) network which we may build from a set of (specification) component processes, where communications or interactions between these processes are hidden. The abstract behaviour which constitutes these communications or interactions may be reified using (external) behaviour decomposition and/or (external) relaxation of atomicity.[2] These reified behaviours will be hidden in the construction of the final implementation network and so the correctness of the final network may be considered using standard notions of refinement. However, since standard (process algebraic) refinement does not fully reflect the ways in which we may wish to move from the abstract to the concrete at the level of component processes, we cannot verify directly the correctness of individual component processes in the general case. This has implications in terms of the state explosion problem and also in terms of verifying in isolation the correctness of a component which may be used in a number of different contexts.

In the remainder of this thesis, we are specifically interested in the process algebra CSP [8,9,31,63] (see chapter 2 for a presentation of the language and semantics of CSP). The semantics of CSP focuses only on external behaviours and CSP refinement generally equates to an increase in determinism. More specifically, refinement in CSP is defined in terms of containment of behaviours: that is, Q implements P if and only if the behaviours of Q are contained within those of P (it follows immediately that CSP refinement cannot generally show correctness when external behaviour decomposition and/or external relaxation of atomicity have occurred).

1.3 Further motivation

We have discussed different types of reification and shown that a number of these are not compatible with standard process algebraic refinement, including CSP refinement. Before proceeding, we comment further on the desirability of developing a notion of correctness which *can* deal with these types of reification; otherwise, it is simply an academic curiosity that CSP cannot deal with them. The most obvious justification is that they are a standard part of the software or hardware development process: the more concrete a process representation becomes, the more concrete data representation becomes, the more specifically external behaviours are defined and the

[2]These reifications are external to the particular component under consideration.

greater the likelihood that behaviours which were originally specified to be atomic can now overlap and potentially interfere with each other.

Although many examples abound to illustrate this point, we take one from the paper [24], which paper was concerned with a problem similar to that which we address here (see discussion in chapter 5). Note that the *interface* events referred to in the following example are simply *external* events in our approach.

> Consider a network of server tills that interfaces with customers. At an abstract level, it might be convenient to describe a banking transaction at some till as a single *atomic* interface event that checks the banking card and PIN code, dispenses the requested amount and charges the account. During subsequent reification steps, such events will be broken up. Inserting the card, checking the PIN code, obtaining the amount to be dispensed need to appear as separate and new interface events. More interesting, the need for a centralized database of client accounts will have to be recognized, which has an *asynchronous* interface with the tills (at least if one aims for a standard implementation). Consequently, although from the point of view of a client his transaction ends with the money being dispensed, the system will view the transaction's end only when the account has been charged. However, because these updates occur *asynchronously* it is now possible that a client, possibly the same, initiates a second transaction before the first one is completed. Clearly, there is a big difference between the computations that are described by the top-level specification and the corresponding low-level ones. If one looks at what happens at a single server till, then the high-level behaviour will be a sequence of atomic transaction events and the state in between events will always describe properly balanced accounts. The low-level behaviour looks different: first of all, the low-level events that "implement" some high-level transaction appear distributed along the sequence of events; moreover, the states in between events may now show unbalanced accounts because money may have been dispensed without the account yet showing it.

If it is agreed that an example such as this makes sense, then we need a notion of refinement which may be used to formalize the above reification steps, something which cannot be done with standard CSP refinement. Although we do not consider this example any further, the types of reification used here reappear in the example from chapter 7: high-level, atomic read and write transactions are implemented at a lower-level as a *series* of events;

moreover, the read and write implementations may overlap since communication is now asynchronous at the lower level.

1.4 Correctness in context

Standard CSP refinement has an extremely useful property, in that it is "context-insensitive". This means that, if Q refines P, there is no restricton on the context into which we might place Q to give a correct implementation network: whatever the context, the resulting implementation network will always refine the specification network which results from placing P in the same context. This follows from the fact that CSP refinement is a *pre-congruence* with respect to the operators of the language (the operators are monotonic with respect to the ordering induced by behaviour containment). More formally, if $\mathcal{S}[\![P]\!]$ returns the semantic meaning of a syntactic process term P, F denotes a CSP context taking n process arguments, $W_1, \ldots, W_n$ is a set of syntactic CSP terms and V_i is a process term for some $1 \leq i \leq n$ where $\mathcal{S}[\![V_i]\!] \subseteq \mathcal{S}[\![W_i]\!]$, then $\mathcal{S}[\![F(W_1, \ldots, V_i, \ldots, W_n)]\!] \subseteq \mathcal{S}[\![F(W_1, \ldots, W_i, \ldots, W_n)]\!]$. However, it is precisely this property of context-insensitivity which is incompatible with the types of reification which we would like to apply at a component process level.

In general, we do not use particular components independently of a particular context. It is therefore possible to consider a notion of correctness which may be called *correctness-in-context*, whereby we prove the correctness of a particular implementation component in relation to a restricted set of contexts. Correctness will be preserved if the context is "valid" but may not be if this is not the case. Such a notion has been considered previously in [45] with respect to *bisimulation equivalence* (see [50] for more details on bisimulation equivalence). The approach to verification of concurrent systems used by the *rely-guarantee* method (see [18] or [23] for example) also uses a notion of correctness-in-context to an extent: we *rely* on the fact that the context in which our component is placed meets certain properties and, in the event that it does, we can *guarantee* correct behaviour of our component. If the context does not have the properties it is meant to then we cannot guarantee the correctness of our component. Dingel's thesis ([21]) also uses a similar notion of correctness-in-context as he develops a refinement calculus to be used in the derivation of parallel programs.

We now give a smaller but more fully realised example than that discussed in the previous section (it will later be used to illustrate new concepts and techniques as they are introduced). It constitutes an instance where standard CSP refinement fails to show correctness but we do have correctness when placed in context. From this example we shall move to give a general statement of the problem which we attempt to solve in this thesis and shall distinguish from those described above the notion of correctness-in-context which is required here. (Note that the example used here is deliberately simple in order to convey the basic premise behind the work in this thesis. See chapter 7 for a more significant example, where we have to deal simultaneously with data reification, external behaviour decomposition and also external relaxation of atomicity.)

1.4.1 External behaviour decomposition from fault tolerance

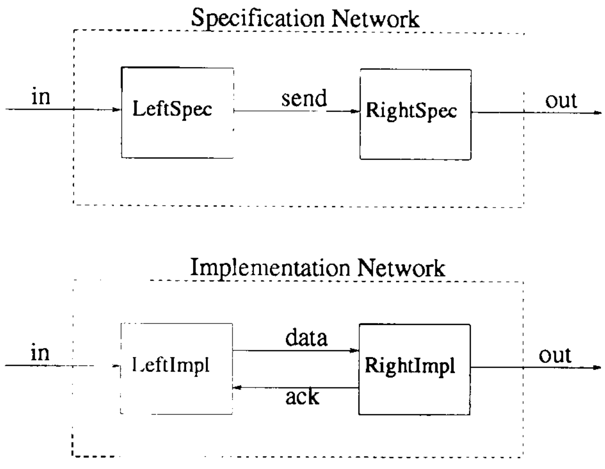

Figure 1.1: Fault-tolerant communication

Figure 1.1 shows a specification network and a corresponding implementation network, each consisting of two component processes where the communication between those two processes is hidden. The abstract communication between the two processes in the specification network has been rendered in the implementation network in a more concrete fashion using a particular communication protocol. The specification network consists of two single slot buffers, *LeftSpec* and *RightSpec*, connected by a channel *send*. The specification network, where communication on *send* is hidden, thus gives us a 2-slot buffer.[3] In the implementation network, we assume that data transmission between the buffers is actually unreliable: therefore, we have the (unreliable) channel *data* and also a channel which transmits acknowledgements. We assume that transmission of a single data item can fail at most once — failure is signified by the transmission of *no* on the channel *ack* — and

[3]We shall give a proper CSP definition of these processes in chapter 2 once we have defined and explained the syntax of the language.

so retransmission is guaranteed to succeed. When a value is received on the channel *in* by *LeftImpl*, it attempts to send that value on the channel *data*. If the transmission is successful — i.e. the value *yes* is transmitted on *ack* — then *LeftImpl* is ready to receive input again. If transmission fails, then the value is resent on *data* and *LeftImpl* is once again ready for input. Once *RightImpl* has successfully received a value on channel *data*, it outputs that value on channel *out*. The implementation *network*, where behaviour on the channels *data* and *ack* is hidden, has the same behaviour as the specification network, where behaviour on channel *send* is hidden, and this can be shown using standard CSP refinement. However, we could not use standard CSP refinement to show that *LeftImpl* implements *LeftSpec*, nor that *RightImpl* implements *RightSpec* since we have used external behaviour decomposition in the move from specification component to corresponding implementation component.

This can be illustrated using the following example. Let us assume that the data values being transmitted are bit values. Then $\langle in.0, data.0, ack.yes\rangle$ would be a possible trace[4] of *LeftImpl*. The corresponding trace of *LeftSpec* would be $\langle in.0, send.0\rangle$. Refinement would obviously fail because the two traces are different. Yet, intuitively, they are doing the same thing and would both cause the value 0 to be transmitted on channel *out* in their respective networks.

To verify correctness in such a case we need some way, for example, of interpreting $\langle in.0, data.0, ack.yes\rangle$ as $\langle in.0, send.0\rangle$. To do this requires an interpretive mapping from *traces* to *traces* and so the existing operators in CSP of hiding and renaming are not powerful enough to perform the necessary interpretation in the general case. For example, as well as interpreting $\langle in.0, data.0, ack.yes\rangle$ as $\langle in.0, send.0\rangle$, $\langle in.0, data.0, ack.no\rangle$ would have to be interpreted as $\langle in.0\rangle$.

1.5 A new notion of correctness-in-context

The notions of correctness-in-context from other authors which are described above are based on a simple premise. This is the fact that, when we compose processes in parallel, the resulting behaviours are compatible with each other according to the synchronization scheme used. In other words, the nature of the context forces the removal of certain behaviours from the implementation: the remaining behaviours will meet certain properties according to the restrictions placed upon the behaviours of the context and so "incorrect" implementation behaviours will be discarded. Crucial, therefore, is the use

[4] A *trace* is a sequence of *visible* actions which may be performed by a process. Traces are denoted using a sequence of actions contained within a pair of angled brackets $\langle\ldots\rangle$.

of parallel composition to join the implementation with any suitable context. In the example given in figure 1.1, the components are also joined using parallel composition. Of most importance, however, is the fact that, in the construction of the final implementation network, we *hide* those parts of the implementation components where external behaviour decomposition has taken place. We therefore aim to develop a notion of correctness-in-context based primarily on hiding rather than on parallel composition (although parallel composition will still play an important role), which notion we shall call *refinement-afer-hiding*.

Central to the development of such a notion is the ability to partition the events of both implementation and specification components into ***potentially finally visible*** — referred to hereafter as ***finally visible*** — and ***finally invisible*** events.[5] The set of finally visible events are those which may be left visible when we construct our final networks; in general, they are the same for both specification and implementation components. (They need not all be left visible, however, and some may still be hidden.) Finally invisible events *must* be hidden when we construct our final networks, whether implementation or specification. The events on channels *data* and *ack* would be the finally invisible events in the implementation network from figure 1.1; the events on channel *send* would be the finally invisible events from the corresponding specification network. The events on channels *in* and *out* would be the finally visible events.

We now give a more formal, though rather abstract, characterisation of what it means to constitute a notion of refinement-after-hiding. Let F_{spec} and F_{impl} be two process contexts, each taking n process arguments, where F_{spec} is a specification context and F_{impl} is the corresponding implementation context.[6] $F_{impl}(Q_1, \ldots, Q_n)$ is then an implementation network and $F_{spec}(P_1, \ldots, P_n)$ the corresponding specification network, where P_i is intended to specify Q_i in some sense for $1 \leq i \leq n$. F_{impl} must hide all finally invisible events from the processes $Q_1, \ldots, Q_n$ and F_{spec} must hide all finally invisible events from $P_1, \ldots, P_n$. For the example in figure 1.1, F_{impl} would hide the events on the channels *data* and *ack*; F_{spec} would hide the events on the channel *send*. It should then be the case that, if Q_i refines-after-hiding P_i for $1 \leq i \leq n$, $F_{impl}(Q_1, \ldots, Q_n)$ refines $F_{spec}(P_1, \ldots, P_n)$ according to standard CSP refinement.[7]

[5]It is possible that a particular component may engage only in finally visible events or only in finally invisible events.

[6]The relation between these two contexts will be made more formal in the next chapter; intuitively, however, they contain the "same" operators, although those operators may be parameterised with different sets of events according to the external reification which has taken place.

[7]This is a kind of ***soundness*** requirement. The issue of ***completeness*** — i.e. that Q_i should refine-after-hiding P_i for $1 \leq i \leq n$ whenever $F_{impl}(Q_1, \ldots, Q_n)$ refines

Using this simple definition, we are able to present some basic conditions which should be met by any notion of refinement-after-hiding. Rendering these in each of the models used to give a semantics to CSP allows us to derive in each model a theory of this notion of refinement. Using these results, it was possible to modify and extend a previously published ([12]) concrete notion of refinement-after-hiding.[8] In view of the theory, it is much easier to identify those aspects of the concrete notion which are the result of choice and those which are crucial to the fact of presenting a notion of refinement-after-hiding. One of those crucial properties is that of *compositionality*: that is, the operator allowed for process composition[9] is monotonic with respect to the ordering induced by the new refinement relation.

Intuitively, our concrete notion of refinement-after-hiding requires the *interpretation* of the behaviours of the implementation, leaving finally visible events as they are and manipulating finally invisible events. We then check for containment of these interpreted behaviours in the behaviours of the specification. For example, we would interpret $\langle in.0, data.0, ack.yes\rangle$ from *LeftImpl* in figure 1.1 by leaving events on channel *in* unchanged — events on this channel are finally visible — and interpreting events on *data* and *ack* as occurring on *send*. This would allow us to interpret $\langle in.0, data.0, ack.yes\rangle$ as $\langle in.0, send.0\rangle$, that is, as a trace of the corresponding specification. We are therefore able to verify the correctness of the implementation components *LeftImpl* and *RightImpl* and thereby infer the correctness of the implementation network without actually having to build it or the specification network (this, of course, relies on the compositionality of the scheme presented). And, more generally, we can verify correctness when external behaviour decomposition and/or external relaxation of atomicity occur in tandem with any internal reification such as data reification.

Tool support is crucial to the successful application of any means of verification, although tool development may be a very lengthy process. Encoding as a CSP context the interpretive mapping used in our notion of refinement-after-hiding allows us to use the existing tool FDR2[10] as a means of automatic verification. This confers a number of the benefits of a mature tool without requiring the effort usually needed to reach this level of maturity,

$F_{spec}(P_1, \ldots, P_n)$ according to standard CSP refinement — is not considered in this thesis, although it is identified in chapter 8 as an area for further work. Nonetheless, the work presented in chapter 3 is important with respect to this in that it attempts to establish a framework which allows the refinement-after-hiding relation to be as large as possible.

[8]By "concrete" notion of refinement-after-hiding we simply mean one that may be used in practice, in contrast to the more abstract notion given by the theory.

[9]A single operator, combining both parallel composition and hiding, is used for composition in the concrete notion. The reason for following this approach in practice is discussed in chapter 4.

[10]The tool is produced by Formal Systems; see [63] or [64] for more details.

which effort would have been beyond the scope of this thesis. Finally, we use this means of automatic verification to verify that a particular mechanism for asynchronous communication is a correct implementation of a register or variable. Since moving from the register to the asynchronous communication mechanism uses data reification, external behaviour decomposition and external relaxation of atomicity, we are able to employ the notion of refinement-after-hiding to show correctness where we could not have done so using standard CSP refinement.

1.6 Organisation of the thesis

The thesis is organised as follows:

- **Chapter 2** gives detail on the syntax and semantics of CSP, along with a more formal description of the processes given in figure 1.1, which processes are used as a running example. Useful notation and concepts are also introduced.
- **Chapter 3** presents a theory of refinement-after-hiding in each of the three semantic models of CSP.
- **Chapter 4** gives a concrete notion of refinement-after-hiding for each semantic model.
- **Chapter 5** considers related work.
- **Chapter 6** details the manner in which the existing tool FDR2 may be used to verify automatically the concrete notions of refinement-after-hiding presented in chapter 4.
- **Chapter 7** uses this means of automatic verification to verify the correctness of an asynchronous communication mechanism as described above.

Note also the following:

- For the purposes of presentation, proofs generally appear in the appendix. Where they do not appear in the appendix, they will appear in the main body of the text alongside the relevant result or an informal justification of the result will be given instead.
- A list of notation and page numbers of the various definitions used in the thesis can be found in appendix E.

1.7 Contributions of the thesis

An earlier version of chapter 4 was published as [12] and formed part of the book chapter [15]. An updated version of the work in [12] appeared as a technical report ([13]), a version of which report has been published in Fundamenta Informaticae ([16]). The algorithms for automatic verification which are discussed in chapter 6 were published as [11] and appear in an updated form in [16]. Other than these algorithms, the work in chapters 3, 6 and 7 is solely that of the author. Chapter 4 gives more detail on the aspects of the work there which are new.

Chapter 2

Modelling concurrent systems

As indicated in chapter 1, we shall use the process algebra CSP to describe and give meaning to concurrent systems. We first give a brief and informal introduction to CSP and its semantics before proceeding to consider these areas in more detail.

2.1 Brief introduction to CSP

CSP is a process algebra, which comes equipped with a language for process description and a denotational semantics which ascribes meaning to processes expressed in that language. It is intended for the description and verification of concurrent systems and is consequently equipped with operators for defining processes which are suited to that task. In particular, both specification and implementation processes are described using the same language. Since the primary focus of the formalism is on the interactions which occur between concurrent processes, the semantics abstracts away from the internal behaviour of processes, focusing only on externally observable behaviour. As a result, the behaviour of a process is characterised by the events which it may offer to any environment. A tool, FDR2([62–64]), is available for the purposes of automatically verifying the correctness of processes expressed using CSP.

The denotational semantics of CSP is designed to enable us to reason about both safety and liveness properties of processes.[1] Traces — execution sequences which abstract from the occurrence of internal actions — are used to reason about safety properties. Divergences — traces after which a process may engage in an infinite sequence of internal actions — are used to

[1]Informally, a safety property stipulates that "bad" things do not happen during a process execution and a liveness property stipulates that "good" things do eventually happen ([41]).

model the possibility of livelock. Trace/refusal pairs (failures) are used to model the possibility of deadlock: if a process P may refuse a set of events R after a particular trace t, and the environment after the execution of t only offers events from within R (where that environment must synchronize with P on every event in R), then deadlock may arise when P is composed in parallel with the environment. Divergences and failures may be used together to reason about liveness properties. Intuitively, the CSP semantics was designed in order to allow us to detect (and thereby avoid) the possibility of deadlock and also of livelock. In this sense, the liveness issue with which we are primarily concerned is that of ensuring that a process will always make progress, rather than of the exact nature of that progress.

There are three semantic models in which the behaviours of CSP processes may be denoted: these are the *traces*, *stable failures* and *failures divergences* models.[2] The traces model is sufficient for reasoning about safety properties. The stable failures model allows us to reason about both safety properties and the possibility of deadlock. The failures divergences model allows us to reason about safety properties, the possibility of deadlock and also the possibility of livelock. In each of these models, a (semantic) specification consists of a set of behaviours in the relevant semantic model. An implementation also consists of a set of behaviours. Refinement in CSP is defined in terms of containment of behaviours: that is, Q implements P if and only if the behaviours of Q are contained within those of P. If Q implements P in the failures divergences model, then Q is at least as deterministic as P.

If Q implements P in the traces model, we know that Q will never execute any traces which P cannot execute; if Q implements P in the stable failures model, after any trace t Q cannot refuse any more than P refuses after t; and Q may not livelock after t if that is not also possible for P when Q implements P in the failures divergences model. Intuitively, if we may place P in a context and the resulting network will suffer from neither deadlock nor livelock, then the same will be true of the network resulting from the placing of Q in the same context. (This latter only holds, of course, if Q implements P in the failures divergences model.)

We now consider in more detail the syntax and semantics of the language. Note that our treatment of CSP is based firmly on that flavour of it presented in [63]. However, the treatment in [63] models the fact of termination by including a distinguished termination event in the semantics; it also includes a sequential composition operator, ; , the semantics of which is defined in terms of this termination event. We do not model the fact of termination here — its consideration is orthogonal to the issues in which we are primarily

[2]Each of these models denotes processes using a different combination of traces, failures and divergences. The failures divergences model — as the name suggests — uses failures and divergences, giving the fullest and most accurate picture of processes.

interested — and so do not use the sequential composition operator.

2.2 Processes and syntax

A CSP process may be regarded as a black box which can communicate with its external environment. Atomic instances of this communication are called *events* or *actions* and must be elements of the *universal alphabet*, Σ. Σ is a *finite* set containing all events or actions which may be communicated by any process in the universe of processes under consideration. In semantic models incorporating failures, a process will *refuse* all those events from Σ which it does not offer. The most important assumptions about (communication) events in CSP are the following:

- An event occurs only when all of its participants are ready to execute it. As soon as all of the participants are ready to execute an event then it (or some other event) *must* occur.
- Event occurrences are instantaneous, as we abstract their duration into single moments. They are non-overlapping as we use an interleaving semantics.

We shall say that a process may *engage* in a particular event when it is possible for it to communicate that event at some point during its lifetime. Events in CSP occur on communication *channels*. The type of a channel c is given by a (possibly empty) sequence of data types $T_1, \ldots, T_k$ (note that the product $T_1.T_2 \ldots T_{k-1}.T_k$ may itself be regarded as a data type and so the type of c is also $T_1.T_2 \ldots T_{k-1}.T_k$). Events which may be communicated on channel c are then of the form

$$c.v_1.v_2 \ldots v_{k-1}.v_k$$

where $v_i \in T_i$ for $1 \leq i \leq k$. In the event that $k = 0$, c denotes a simple event with no explicit data content. It will also be useful to be able to refer to αc, the *alphabet* of channel c, which gives all events which may occur on that channel:

$$\alpha c \triangleq \{c.v_1.v_2 \ldots v_{k-1}.v_k \mid v_1 \in T_1, \ldots, v_k \in T_k\}.$$

It is required that $\alpha c \subseteq \Sigma$ for any channel which is defined. By the finiteness of Σ, this means that channels may be defined using only finite types. We may also use α to "complete" partially defined events on channel c, where $1 \leq j \leq k$ and $v_i \in T_i$ for $1 \leq i \leq j$:

$$\alpha c.v_1.v_2 \ldots v_j \triangleq \{c.v_1.v_2 \ldots v_j.w_{j+1} \ldots w_k \mid w_{j+1} \in T_{j+1}, \ldots, w_k \in T_k\}.$$

2.2.1 Operators

The nullary operator *STOP* is used to denote the deadlocked process, while *DIV* is used to denote the immediately diverging process. $a \rightarrow P$ (referred to as the prefix operator) gives the process which first engages in the event a (where $a \in \Sigma$) and then proceeds to behave like P. $\Box$ denotes *deterministic* choice; $P \Box Q$ is a process where the initial events of P and Q are offered simultaneously. $\sqcap$ denotes *non-deterministic* choice; in $P \sqcap Q$ we may be offered the initial events of P *or* the initial events of Q but not both. Moreover, we have no control over which is offered. The operators $\Box$ and $\sqcap$ are both commutative and associative in each of the three semantic models used here and thus may be indexed over *finite* sets. (This issue of indexing is considered further in section 2.4.6 below.)

Let P be a process and $A \subseteq \Sigma$; then $P \backslash A$ is a process that behaves like P with the actions from A made invisible ($\backslash$ is the *hiding* operator and $\backslash A$ *hides* the events in A). $\backslash A$ is *left associative*: i.e. $P \setminus A \setminus A'$ is defined as $(P \backslash A) \backslash A'$. Parallel composition $P \parallel_Y Q$ (where $Y \subseteq \Sigma$) models synchronous communication between P and Q in such a way that each of them is free to engage independently in any action which is not in Y but they have to engage simultaneously in all actions that are in Y. (We say that the parallel composition *synchronizes* on the set of events Y or that P and Q synchronize on Y.) The interleaving operator $|||$ is used to denote $\parallel_{\varnothing}$. $|||$ is commutative and associative in all three semantic models and so may be indexed by finite sets.

Let $G \subseteq \Sigma \times \Sigma$ be a relation (called a *renaming* relation) and P a process. Then $P[G]$ is a process that behaves like P except that every action a has been replaced by

$$G(a) \triangleq \{b \mid a \; G \; b\}.$$

Wherever the action a might have been enabled in P, *each* of the events in $G(a)$ will be enabled in its place in $P[G]$. Note that the relation G need not be (explicitly) total over the events of P: for a not in the domain of G, we assume that $G(a) = \{a\}$. (This mirrors the way in which renaming relations are used in FDR2.)

Recursion is introduced using a special equational style of definition. In the simplest case, we may define the process N in terms of the arbitrary CSP term P using $N = P$, where this simply means that N is taken to be the process defined by P. This gives rise to a single recursion if the name N occurs somewhere in P. More generally, mutual recursion may be introduced using a collection of equational definitions $N_i = P_i$ for $1 \leq i \leq l$, where each P_i may contain zero, one or more of the process names N_j for $1 \leq j \leq l$. We assume that all process names which appear in any syntactic definition to which a semantics is to be given are defined exactly once. In other words, we deal only with *closed* terms. (Note that such equational definitions need not only be used to introduce recursion and they are often used simply to make clearer the presentation of syntactic definitions.)

The notation $N_x = P$ or $N(x) = P$ is used to represent the *family* of processes N_v or $N(v)$ such that $N_v = P[v/x]$ (respectively $N(v) = P[v/x]$) where v is a concrete data value and $P[v/x]$ denotes the process P with all occurrences of the parameter x replaced by v. This notation generalizes in the obvious way to parameterization with multiple data values.

2.2.2 Syntactic sugar

From chapter 4 onwards, we use an operator which is not part of the standard CSP syntax and which is, in fact, shorthand for a particular combination of two operators already seen. The *network composition* operator, $\otimes_Y$, is such that

$$(P \otimes_Y Q) \triangleq (P \parallel_Y Q) \setminus Y.$$

In view of this, we need not define specifically the semantics of this operator and will derive it where necessary from the semantics of the parallel composition and hiding operators.

2.2.3 Finite non-determinism

In the failures divergences model, divergence is introduced in $P \setminus X$ when P can perform an infinite consecutive sequence of events in X. However, the failures divergences model only includes information on *finite* traces, not infinite ones. If a process possesses an infinite trace t, then u will be a (finite) trace of that process for all $u < t$. However, in the general case, the converse may fail: it may be possible that t is *not* an infinite trace of the process even though u is a trace of the process for all $u < t$. If this latter case is allowed to arise, then it is not possible to give an exact definition of the semantics of hiding in the failures divergences model.

It is possible to represent a CSP process operationally as a labelled transition system or LTS (see [63] for details). If, for any node P in an LTS C, there are only finitely many nodes we can reach from P under a single action, then C is said to be *finitely non-deterministic*. In other words, for any particular action that we engage in, there are only a finite number of possible states we can end up in. By Koenig's Lemma, if the LTS representation of a CSP process is finitely non-deterministic, then t is an infinite trace of that process if and only if u is a (finite) trace of the process for all $u < t$. (For proof see [63].)

Since Σ is finite, none of the operators which we use are capable of introducing infinite non-determinism. As a result, all processes under consideration may be represented operationally by a finitely non-deterministic LTS and so it is possible to define exactly the semantics of the hiding operator.

2.3 Notation

The following notations will prove to be useful, where $t, u, t_1, t_2, \ldots$ are traces; A is a set of actions; $\mathcal{T}, \mathcal{T}'$ are non-empty sets of traces; $G \subseteq \Sigma \times \Sigma$ is a relation; and $\mathcal{X}$ is a set of sets. Note that traces are assumed to be finite unless otherwise stated.

- $t = \langle a_1, \ldots, a_n \rangle$ is the trace whose i-th element is action a_i, and length, $|t|$, is n. Moreover, $\mathit{events}(t) \triangleq \{a_1, \ldots, a_n\}$ and, provided that $n \geq 1$, $\mathit{tail}(t) \triangleq a_n$. If $n = 0$ then t is the empty trace, denoted $\langle\,\rangle$.
- A, $|A|$ denotes the cardinality of A.
- $t \circ u$ is the trace obtained by appending u to t.
- A^* is the set of all traces — i.e. sequences — of actions from A, including the empty trace, $\langle\,\rangle$.
- A^ω is the set of all *infinite* traces of actions from A.
- $\mathcal{T}^*$ is the set of all traces $t = t_1 \circ \cdots \circ t_n$ $(n \geq 0)$ such that $t_1, \ldots, t_n \in \mathcal{T}$ (note that $t = \langle\,\rangle$ when $n = 0$).
- $\leq$ denotes the prefix relation on traces, and $t < u$ if $t \leq u$ and $t \neq u$.
- $\mathit{Pref}(\mathcal{T}) \triangleq \{u \mid (\exists t \in \mathcal{T})\ u \leq t\}$ is the *prefix-closure* of $\mathcal{T}$. (In the event that $\mathcal{T}$ is the singleton set $\{t\}$, we may use $\mathit{Pref}(t)$ in lieu of $\mathit{Pref}(\mathcal{T})$.)
- $\mathcal{T}$ is *prefix-closed* if $\mathcal{T} = \mathit{Pref}(\mathcal{T})$.
- $t\lceil A$ is a trace obtained by deleting from t all the actions that do not occur in A.
- $t \setminus A$ is a trace obtained by deleting from t all the actions that *do* occur in A.
- The definitions of $\lceil$ and $\setminus$ may be lifted to *sets* of traces in the obvious way: $\mathcal{T}\lceil A \triangleq \{t\lceil A \mid t \in \mathcal{T}\}$ and $\mathcal{T} \setminus A \triangleq \{t \setminus A \mid t \in \mathcal{T}\}$.
- $t_1, t_2, \ldots$ is an *ω-sequence* of traces if $t_1 \leq t_2 \leq \ldots$ and $\lim_{i \to \infty} |t_i| = \infty$.

- A mapping $f : \mathcal{T} \to \mathcal{T}'$ is *monotonic* if $t, u \in \mathcal{T}$ and $t \leq u$ implies $f(t) \leq f(u)$, and *strict* if $\langle\rangle \in \mathcal{T}$ and $f(\langle\rangle) = \langle\rangle$.
- The definition of G may be lifted to sets of events, traces and sets of traces:
  - $G(A) \triangleq \bigcup\{G(a) \mid a \in A\}$.
  - $\langle a_1, \ldots, a_n\rangle\ G\ \langle b_1, \ldots, b_m\rangle \Leftrightarrow n = m \wedge \forall i \leq n,\ a_i\ G\ b_i$.
  - $G(\mathcal{T}) \triangleq \{u \mid (\exists t \in \mathcal{T})\ t\ G\ u\}$.

  In the event that $\mathcal{T}$ is the singleton set $\{t\}$, we may use $G(t)$ in lieu of $G(\mathcal{T})$. Moreover, if $G(t) = \{u\}$ for some trace u then we shall denote this $G(t) = u$. Similarly, if $G(a) = \{b\}$ for some action a then we write $G(a) = b$.
- $G^{-1} \triangleq \{(b, a) \mid a\ G\ b\}$ is the *inverse* of G.
- $Sub(\mathcal{X}) \triangleq \{W \subseteq X \mid X \in \mathcal{X}\}$ is the *subset-closure* of $\mathcal{X}$.
- $\mathcal{X}$ is *subset-closed* if $\mathcal{X} = Sub(\mathcal{X})$.
- $2^S \triangleq \{X \mid X \subseteq S\}$ gives the *power set* of S. For purposes of presentation, we will sometimes use $\mathbb{P}(S)$ in lieu of 2^S.
- We introduce containment and equality between *pairs* of sets in the obvious way. Let B, B', C, C' be sets.
  - $(B, C) \subseteq (B', C')$ if and only if $B \subseteq B'$ and $C \subseteq C'$.
  - $(B, C) = (B', C')$ if and only if $B = B'$ and $C = C'$.
- For an arbitrary set of objects O and a partial ordering[3] $\preceq$ over the elements of O,

  $$max_{\preceq}(O) \triangleq \{e \in O \mid (\nexists d \in O)\ e \preceq d\ \wedge\ e \neq d\}.$$

  In the event that $max_{\preceq}(O) = \{e\}$ for some element e, we shall write $max_{\preceq}(O) = e$.

[3] A partial ordering is reflexive, transitive and antisymmetric.

2.4 Process semantics

In this section we consider each of the three semantic models in more detail, including the semantic definitions in each model of the operators introduced so far and how the semantics of (syntactic) processes may be derived using these definitions. Before doing this, however, we make the following important observations.

At various points in this thesis, we will work with (syntactic) processes, semantic denotations of such processes in any one of three semantic models and sets of behaviours which may not be derivable as the semantics of any process. It is only necessary to explicitly calculate the semantics of processes in chapter 6, as we show how to use FDR2 for automatic verification of our notion of refinement-after-hiding; moreover, we only derive semantics in the failures divergences model in that chapter with respect to the hiding operator. For this reason, semantic definitions for the full range of operators are given only for the traces and stable failures models. Chapters 3 and 4 use only the hiding and parallel composition operators and this is why only the semantics for these two operators are given for the failures divergences model.[4] Chapter 3 deals primarily with sets of behaviours and *pairs* of sets of behaviours and so requires the definition of hiding and parallel composition over such sets and over such pairs. As a result, we define three different versions of the parallel composition and hiding operators. The first version gives a *semantic* operator which may be applied to a set or sets of behaviours as appropriate; the second gives a semantic operator which may be applied to pairs of sets of behaviours; the third version gives a *syntactic* operator which is used in process definitions.[5] Note that we have already defined in the previous section the effect of applying the hiding operator to sets of traces. Note also that the semantic and syntactic versions of a particular operator will be indistinguishable textually and the particular version being used will be clear from the nature of the object or objects to which it is being applied.

In what follows, the semantics of recursion is left until all other operators have been considered, since it is treated in a rather different manner.

2.4.1 The traces model

In the traces model, a process is denoted by a (possibly infinite) set of finite execution sequences of *visible* actions (i.e. actions from Σ). For any process P, we denote the *traces* of P as τP. The following condition always holds of

[4]The interested reader may find the omitted definitions in [63].

[5]In the traces model, we actually define hiding and parallel composition over individual traces, sets of traces and processes. In the failures divergences model, we define the operators only over pairs of sets of behaviours and over processes.

$$\boxed{\begin{array}{lcl}
s \parallel_Y u & = & u \parallel_Y s. \\
\langle\rangle \parallel_Y \langle\rangle & \triangleq & \{\langle\rangle\}. \\
\langle\rangle \parallel_Y \langle y\rangle & \triangleq & \varnothing. \\
\langle\rangle \parallel_Y \langle z\rangle & \triangleq & \{\langle z\rangle\}. \\
\langle y\rangle \circ s \parallel_Y \langle z\rangle \circ u & \triangleq & \{\langle z\rangle \circ v \mid v \in (\langle y\rangle \circ s \parallel_Y u)\}. \\
\langle y\rangle \circ s \parallel_Y \langle y\rangle \circ u & \triangleq & \{\langle y\rangle \circ v \mid v \in (s \parallel_Y u)\}. \\
\langle y\rangle \circ s \parallel_Y \langle y'\rangle \circ u & \triangleq & \varnothing \text{ if } y \neq y'. \\
\langle z\rangle \circ s \parallel_Y \langle z'\rangle \circ u & \triangleq & \{\langle z\rangle \circ v \mid v \in (s \parallel_Y \langle z'\rangle \circ u)\} \cup \\
 & & \{\langle z'\rangle \circ v \mid v \in (\langle z\rangle \circ s \parallel_Y u)\}.
\end{array}}$$

Figure 2.1: Composing traces in parallel, where $s, u \in \Sigma^*$, $y, y' \in Y \subseteq \Sigma$ and $z, z' \in \Sigma - Y$

τP:[6]

T1 τP is non-empty and prefix-closed.

Figure 2.1 shows how the effect of parallel composition is defined in terms of its effect on individual traces. This operator has the following important property:

TRP For traces s and u and $Y \subseteq \Sigma$, if $t \in (s \parallel_Y u)$ then $t\lceil Y = s\lceil Y = u\lceil Y$.

Where $\mathcal{T}$ and $\mathcal{T}'$ are sets of traces[7], we may define (semantic) parallel composition over such sets in the following manner (we assume $Y \subseteq \Sigma$):

$$\mathcal{T} \parallel_Y \mathcal{T}' \triangleq \bigcup\{s \parallel_Y u \mid s \in \mathcal{T} \wedge u \in \mathcal{T}'\}.$$

The semantics of any process (minus recursion) in the traces model may then be derived according to the detail in figure 2.2.

[6]This condition and others like it which are introduced in the remainder of this chapter are theorems which may be derived in the algebra of CSP processes and their denotations.

[7]Neither need be the denotation of a particular process in the traces model and so they need not meet condition T1. In general, none of the sets of behaviours or pairs of such sets to which semantic operators may be applied need be (a component of) the denotation of a process.

$$
\boxed{\begin{array}{lcl}
\tau STOP & \triangleq & \{\langle\rangle\}. \\
\tau DIV & \triangleq & \{\langle\rangle\}. \\
\tau(a \rightarrow P) & \triangleq & \{\langle\rangle\} \cup \{\langle a\rangle \circ s \mid s \in \tau P\}. \\
\tau(P \,\square\, Q) & \triangleq & \tau P \cup \tau Q. \\
\tau(P \sqcap Q) & \triangleq & \tau P \cup \tau Q. \\
\tau(P \setminus A) & \triangleq & (\tau P) \setminus A. \\
\tau(P \parallel_Y Q) & \triangleq & \tau P \parallel_Y \tau Q. \\
\tau(P[G]) & \triangleq & G(\tau P).
\end{array}}
$$

Figure 2.2: Semantics of processes in the traces model, where $G \subseteq \Sigma \times \Sigma$ and $A, Y \subseteq \Sigma$

2.4.2 The stable failures model

In the stable failures model, a process P is denoted by a pair $(\tau P, \phi P)$, where ϕP — the *stable* failures[8] of P — is a subset of $\Sigma^* \times 2^\Sigma$. If $(t, R) \in \phi P$ then P is able to *refuse* R after t. Intuitively, this means that if the environment only offers R as a set of possible events to be executed after t (and the environment must synchronize with P on every event in R) then P can deadlock when placed in parallel with the environment. A process P is *deadlock-free* if and only if, for every $(t, R) \in \phi P$, R is a proper subset of Σ.

The following conditions always hold of τP and ϕP:

SF1 τP is non-empty and prefix-closed.

SF2 $(t, R) \in \phi P \Rightarrow t \in \tau P$.

SF3 $(t, R) \in \phi P \wedge S \subseteq R \Rightarrow (t, S) \in \phi P$.

SF4 $(t, R) \in \phi P \wedge t \circ \langle a\rangle \notin \tau P \Rightarrow (t, R \cup \{a\}) \in \phi P$.

We will consider a set of stable failures $\mathcal{F}$ to be *subset-closed* if:

$$(t, R) \in \mathcal{F} \wedge S \subseteq R \Longrightarrow (t, S) \in \mathcal{F}.$$

[8]Intuitively, a failure is stable if no invisible events are enabled at the state which generates the failure. Invisible events are classed as *urgent* in CSP and do not require synchronization with the environment before they occur. This means that we can never deadlock at a state at which an invisible event is enabled. Since the stable failures model is primarily interested in the possibility of deadlock, we need not record information on states at which it can never occur.

Where $\mathcal{F}$ and $\mathcal{F}'$ are sets of stable failures and $A, Y \subseteq \Sigma$, we may define *semantic* operations of hiding and parallel composition as follows:[9]

$$\begin{array}{lcl} \mathcal{F} \setminus A & \triangleq & \{(t \setminus A, X) \mid (t, X \cup A) \in \mathcal{F}\}. \\ \mathcal{F} \parallel_Y \mathcal{F}' & \triangleq & \{(t, X \cup Z) \mid X - Y = Z - Y \ \wedge \ ((\exists s, u)\ (s, X) \in \mathcal{F} \ \wedge \\ & & (u, Z) \in \mathcal{F}' \ \wedge \ t \in s \parallel_Y u)\}. \end{array}$$

These definitions then lift to *pairs* of sets of behaviours in a straightforward manner, where $\mathcal{T}, \mathcal{T}'$ are sets of traces, $\mathcal{F}, \mathcal{F}'$ are sets of stable failures and $A, Y \subseteq \Sigma$:

$$\begin{array}{lcl} (\mathcal{T}, \mathcal{F}) \setminus A & \triangleq & (\mathcal{T} \setminus A, \mathcal{F} \setminus A). \\ (\mathcal{T}, \mathcal{F}) \parallel_Y (\mathcal{T}', \mathcal{F}') & \triangleq & (\mathcal{T} \parallel_Y \mathcal{T}', \mathcal{F} \parallel_Y \mathcal{F}'). \end{array}$$

The stable failures of any process (minus recursion) may be derived according to the detail in figure 2.3 (the traces of the process may still be derived according to the detail in figure 2.2[10]). The following equalities then hold, where P, Q are processes and $A, Y \subseteq \Sigma$:

- $(\tau(P \setminus A), \phi(P \setminus A)) = (\tau P, \phi P) \setminus A$.
- $(\tau(P \parallel_Y Q), \phi(P \parallel_Y Q)) = (\tau P, \phi P) \parallel_Y (\tau Q, \phi Q)$.

2.4.3 The failures divergences model

In the failures divergences model, a process P is denoted as a pair $(\phi_\perp P, \delta P)$, where δP — the *divergences* of P — is a subset of Σ^* and $\phi_\perp P$ — the *failures* of P — is a subset of $\Sigma^* \times 2^\Sigma$. If $t \in \delta P$ then P is said to *diverge* after t. In the CSP model this means that the process behaves in a totally uncontrollable and unpredictable way: $\phi_\perp DIV = \Sigma^* \times 2^\Sigma$ and $\delta DIV = \Sigma^*$. Semantically, divergence obscures all other behaviours after it arises and we can make no guarantees post-divergence regarding what a process will offer or refuse at any point in time: this is reflected in conditions FD4 and FD5 below.

We introduce the notation $\tau_\perp P$ and define it as $\tau_\perp P \triangleq \{t \mid (t, \varnothing) \in \phi_\perp P\}$. As we shall see in section 2.4.8 below, $\tau_\perp P = \tau P \cup \delta P$. The following conditions then always hold of $\tau_\perp P$, $\phi_\perp P$ and δP:

[9]The particular definition of parallel composition used here reflects the following intuition. If two processes are composed in parallel, synchronizing on the set of events Y, then the composition can refuse after a particular trace all events from Y which at least one process refuses, along with all events which both refuse.

[10]In other words, for any process P, τP has the same meaning whether we are working in the traces model or in the stable failures model.

$$\begin{array}{lcl}
\phi STOP & \triangleq & \{(\langle\rangle, X) \mid X \subseteq \Sigma\}. \\
\phi DIV & \triangleq & \varnothing. \\
\phi(a \to P) & \triangleq & \{(\langle\rangle, X) \mid a \notin X \subseteq \Sigma\} \cup \\
 & & \{(\langle a\rangle \circ s, X) \mid (s, X) \in \phi P\}. \\
\phi(P \mathbin{\Box} Q) & \triangleq & \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \phi P \cap \phi Q\} \cup \\
 & & \{(s, X) \mid (s, X) \in \phi P \cup \phi Q \ \wedge \ s \neq \langle\rangle\}. \\
\phi(P \sqcap Q) & \triangleq & \phi P \cup \phi Q. \\
\phi(P \setminus A) & \triangleq & (\phi P) \setminus A. \\
\phi(P \parallel_Y Q) & \triangleq & \phi P \parallel_Y \phi Q. \\
\phi(P[G]) & \triangleq & \{(s', X) \mid (\exists s)\ s\ G\ s' \ \wedge \ (s, G^{-1}(X)) \in \phi P\}.
\end{array}$$

Figure 2.3: Semantics of processes in the stable failures model, where $G \subseteq \Sigma \times \Sigma$ and $A, Y \subseteq \Sigma$

FD1 $\tau_\perp P$ is non-empty and prefix-closed.

FD2 $(t, R) \in \phi_\perp P \ \wedge \ S \subseteq R \Rightarrow (t, S) \in \phi_\perp P$.

FD3 $(t, R) \in \phi_\perp P \ \wedge \ t \circ \langle a\rangle \notin \tau_\perp P \Rightarrow (t, R \cup \{a\}) \in \phi_\perp P$.

FD4 $s \in \delta P \ \wedge \ t \in \Sigma^* \Rightarrow s \circ t \in \delta P$.

FD5 $t \in \delta P \Rightarrow (t, R) \in \phi_\perp P$ for $R \subseteq \Sigma$.

We now define semantic operators of parallel composition and hiding in this model. By their nature, they can only be defined over *pairs* of sets of behaviours rather than over single sets of behaviours. Where $\mathcal{F}$ is a set of failures, we use $\tau_\perp \mathcal{F} \triangleq \{t \mid (t, \varnothing) \in \mathcal{F}\}$. In the following definitions, $\mathcal{F}$, $\mathcal{F}_1$, $\mathcal{F}_2$ and $\mathcal{F}'$ are sets of failures, $\mathcal{D}$, $\mathcal{D}_1$, $\mathcal{D}_2$ and $\mathcal{D}'$ are sets of divergences and $A, Y \subseteq \Sigma$.

- $(\mathcal{F}_1, \mathcal{D}_1) \parallel_Y (\mathcal{F}_2, \mathcal{D}_2) \triangleq (\mathcal{F}, \mathcal{D})$ where:
  - $\mathcal{D} = \{t \circ v \mid (\exists s \in \tau_\perp \mathcal{F}_1, u \in \tau_\perp \mathcal{F}_2)\ t \in s \parallel_Y u \ \wedge$
    $(s \in \mathcal{D}_1 \ \vee \ u \in \mathcal{D}_2)\}$.

$$\begin{array}{lcl}
\delta(P \parallel_Y Q) & \triangleq & \{t \circ v \mid (\exists s \in \tau_\perp P, u \in \tau_\perp Q)\ t \in s \parallel_Y u\ \wedge \\
 & & (s \in \delta P\ \vee\ u \in \delta Q)\}. \\
\phi_\perp(P \parallel_Y Q) & \triangleq & \{(t, X \cup Z) \mid X - Y = Z - Y\ \wedge\ ((\exists s, u) \\
 & & (s, X) \in \phi_\perp P\ \wedge\ (u, Z) \in \phi_\perp Q\ \wedge\ t \in s \parallel_Y u)\} \cup \\
 & & \{(t, X) \mid t \in \delta(P \parallel_Y Q)\ \wedge\ X \subseteq \Sigma\}. \\
\delta(P \setminus A) & \triangleq & \{(s \setminus A) \circ t \mid s \in \delta P\} \cup \\
 & & \{(u \setminus A) \circ t \mid u \in \Sigma^\omega\ \wedge\ (u \setminus A) \text{ is finite} \\
 & & \wedge\ ((\forall s < u)\ \ s \in \tau_\perp P)\}. \\
\phi_\perp(P \setminus A) & \triangleq & \{(t \setminus A, X) \mid (t, X \cup A) \in \phi_\perp P\} \cup \\
 & & \{(t, X) \mid t \in \delta(P \setminus A)\ \wedge\ X \subseteq \Sigma\}.
\end{array}$$

Figure 2.4: Semantics of operators in the failures divergences model, where $A, Y \subseteq \Sigma$

- $\mathcal{F} = \{(t, X \cup Z) \mid X - Y = Z - Y\ \wedge\ ((\exists s, u)\ (s, X) \in \mathcal{F}_1\ \wedge\ (u, Z) \in \mathcal{F}_2\ \wedge\ t \in s \parallel_Y u)\} \cup \{(t, X) \mid t \in \mathcal{D}\ \wedge\ X \subseteq \Sigma\}.$

- $(\mathcal{F}, \mathcal{D}) \setminus A \triangleq (\mathcal{F}', \mathcal{D}')$ where:
  - $\mathcal{D}' = \{(s \setminus A) \circ t \mid s \in \mathcal{D}\} \cup \{(u \setminus A) \circ t \mid u \in \Sigma^\omega\ \wedge\ (u \setminus A) \text{ is finite} \wedge ((\forall s < u)\ \ s \in \tau_\perp \mathcal{F})\}.$
  - $\mathcal{F}' = \{(t \setminus A, X) \mid (t, X \cup A) \in \mathcal{F}\} \cup \{(t, X) \mid t \in \mathcal{D}'\ \wedge\ X \subseteq \Sigma\}.$

Figure 2.4 gives the semantic definitions of the *syntactic* operators of hiding and parallel composition. They are not defined in terms of the semantic operators so that it is easier to see how the individual components of a particular denotation pair may be derived. However, since $\tau_\perp(\phi_\perp P) = \tau_\perp P$, the following equalities hold, where P, Q are processes and $A, Y \subseteq \Sigma$:

- $(\phi_\perp(P \parallel_Y Q), \delta(P \parallel_Y Q)) = (\phi_\perp P, \delta P) \parallel_Y (\phi_\perp Q, \delta Q).$
- $(\phi_\perp(P \setminus A), \delta(P \setminus A)) = (\phi_\perp P, \delta P) \setminus A.$

2.4.4 Process denotations and refinement

$[\![P]\!]_X$ denotes the semantic meaning of the process P in the model $X \in \{T, SF, FD\}$ (T gives the traces model, SF the stable failures model and FD the failures divergences model). In other words:

- $[\![P]\!]_T \equiv \tau P$.
- $[\![P]\!]_{SF} \equiv (\tau P, \phi P)$.
- $[\![P]\!]_{FD} \equiv (\phi_\perp P, \delta P)$.

We shall also use the shorthand $P =_X Q$ to indicate that $[\![P]\!]_X = [\![Q]\!]_X$. That Q is an implementation of (or *refines*) P in a particular semantic model $X \in \{T, SF, FD\}$ is denoted $Q \sqsupseteq_X P$. This means that the behaviours of Q in the relevant model are contained in those of P. In other words:

- $Q \sqsupseteq_T P$ if and only if $[\![Q]\!]_T \subseteq [\![P]\!]_T$ (i.e. $\tau Q \subseteq \tau P$).
- $Q \sqsupseteq_{SF} P$ if and only if $[\![Q]\!]_{SF} \subseteq [\![P]\!]_{SF}$ (i.e. $\tau Q \subseteq \tau P$ and $\phi Q \subseteq \phi P$).
- $Q \sqsupseteq_{FD} P$ if and only if $[\![Q]\!]_{FD} \subseteq [\![P]\!]_{FD}$ (i.e. $\phi_\perp Q \subseteq \phi_\perp P$ and $\delta Q \subseteq \delta P$).

2.4.5 Semantics of recursion

We now show how to define the semantics of recursive terms. Since we never have to calculate the semantics of recursive processes in the failures divergences model, we deal explicitly here only with the traces and stable failures models. We will consider a single recursion $N = P$ and also a mutual recursion $N_i = P_i$ for $1 \leq i \leq k$. The fundamental law of recursion is given by the following condition, which effectively states that a recursively defined process satisfies the equation defining it.

REC If $N = P$ is a recursive definition, then $N =_X P$ for $X \in \{T, SF, FD\}$.

Before proceeding, we introduce some notation which will be useful.

- $\mathbb{N}$ denotes the set of natural numbers.
- For any function $\mathcal{S}$ and $n \geq 2$, $\mathcal{S}^n(X) \triangleq \mathcal{S}(\mathcal{S}^{n-1}(X))$, where $\mathcal{S}^1 \triangleq \mathcal{S}$.
- $P[Y/N]$ denotes the process P with the process Y substituted for the name N.
- $\perp_T$ is used to give the denotation of $STOP$ in the traces model: i.e. $\perp_T \triangleq [\![STOP]\!]_T = \{\langle\rangle\}$.

- $\bot_{SF}$ is used to give the denotation of DIV in the stable failures model: i.e. $\bot_{SF} \triangleq [\![DIV]\!]_{SF} = (\{\langle\rangle\}, \varnothing)$.
- $\langle X_1, \ldots, X_k\rangle$ will be used to denote the vector with elements $X_1, \ldots, X_k$. We then assume:
  - $\overline{Y}$ is a vector of processes $\langle Y_1, \ldots, Y_k\rangle$.
  - $\overline{N}$ is the vector of names $\langle N_1, \ldots, N_k\rangle$.
  - $\overline{P}$ is the vector of processes $\langle P_1, \ldots, P_k\rangle$.
  - $P_i[\overline{Y}/\overline{N}]$ denotes the process P_i with the process Y_j substituted for the name N_j for all $1 \le j \le k$ such that N_j occurs somewhere in P_i.

Single recursion

The single recursion $N = P$ induces a (syntactic) function, F, which maps syntactic terms to syntactic terms such that, for any process Y, $F(Y) \triangleq P[Y/N]$. In the model $X \in \{T, SF\}$, $\mathcal{S}$ is the (semantic) function from process denotations to process denotations such that, for any process Q, $\mathcal{S}([\![Q]\!]_X) \triangleq [\![F(Q)]\!]_X$. For example, $\mathcal{S}(\tau Q) \triangleq \tau F(Q)$. $[\![N]\!]_X$ is then given by $\bigcup_{n\in\mathbb{N}} \mathcal{S}^n(\bot_X)$.

If N does not occur anywhere in P then $\mathcal{S}$ is a constant and its value gives directly the semantic interpretation of N: i.e. $[\![N]\!]_X = \mathcal{S}$.

Mutual recursion

The mutual recursion $N_i = P_i$ for $1 \le i \le k$ induces a *set* of syntactic functions F_i such that, for any vector of processes $\overline{Y}$, $F_i(\overline{Y}) \triangleq P_i[\overline{Y}/\overline{N}]$. In the model $X \in \{T, SF\}$, $\mathcal{S}$ may then be taken to be the (semantic) function from *vectors* of process denotations to vectors of process denotations such that, for any vector of processes $\overline{Q} = \langle Q_1, \ldots, Q_k\rangle$,

$$\mathcal{S}(\langle [\![Q_1]\!]_X, \ldots, [\![Q_k]\!]_X\rangle) \triangleq \langle [\![F_1(\overline{Q})]\!]_X, \ldots, [\![F_k(\overline{Q})]\!]_X\rangle.$$

$[\![N_i]\!]_X$ is then given by the ith element of $\bigcup_{n\in\mathbb{N}} \mathcal{S}^n(\langle \bot_X, \ldots, \bot_X\rangle)$.

Guardedness

A process name N is *guarded* in process expression P if either:

- N does not appear in P or;
- P does not contain the hiding operator and every occurrence of N is within the scope of an occurrence of the prefix operator.

If all names occurring in P are guarded then we say that P itself is guarded. If P is guarded, then all recursive equations used to define the semantics of P have a unique solution. All recursive processes for which we have to derive semantics in chapter 6 are guarded and this justifies the inductive derivation of semantics which is used there.[11]

Guardedness and divergence

Guarded processes have the following important property:

DF If $N_i = P_i$ for $1 \leq i \leq k$ and all processes P_i are guarded, then $\delta N_i = \varnothing$ for $1 \leq i \leq k$.

2.4.6 Indexing operators

Further comment is required with respect to the indexing of the operators $\Box$, $\sqcap$ and $|||$. We consider the generic process

$$\oplus_{z \in Z} P(z),$$

where $\oplus \in \{\Box, \sqcap, |||\}$ and Z is a finite set. In the event that $|Z| \geq 2$, then $\oplus_{z \in Z} P(z)$ may be represented in an obvious and straightforward manner using the binary version of $\oplus$. However, it is less clear what should be the semantics of $\oplus_{z \in Z} P(z)$ if $|Z| = 1$ or $Z = \varnothing$. We first observe that the non-deterministic choice operator, $\sqcap$, may *not* be indexed by the empty set (this is disallowed by FDR2). For our purposes, we also assume that $|||$ may not be indexed by the empty set. The remainder of the relevant cases are covered by the following definition.

Definition 2.1. *Let $\oplus \in \{\Box, \sqcap, |||\}$ and $X \in \{T, SF, FD\}$. Then the following hold:*

1. *If $Z = \{v\}$, then $\oplus_{z \in Z} P(z) =_X P(v)$.*
2. *If $Z = \varnothing$, then $\Box_{z \in Z} P(z) =_X STOP$.*

2.4.7 Parallel composition, hiding and network composition

We shall also need a semantic version of the network composition operator, $\otimes_Y$, which is defined as follows for processes P and Q, $Y \subseteq \Sigma$ and $X \in \{T, SF, FD\}$:

[11] Note that the definition of guardedness given here is stronger than the usual definition, namely that every occurrence of N in P is prefixed by an action that cannot be hidden. However, the definition given suffices for our purposes and is simpler to present.

$$[\![P]\!]_X \otimes_Y [\![Q]\!]_X \triangleq ([\![P]\!]_X \parallel_Y [\![Q]\!]_X) \setminus Y.$$

The following important results then show the consistency in all models of the syntactic and semantic versions of hiding, parallel composition and network composition. They may be proved easily using the definitions given so far in section 2.4.

Proposition 2.1. *The following hold, where P and Q are processes, $A, Y \subseteq \Sigma$ and $X \in \{T, SF, FD\}$:*

1. $[\![Q \setminus A]\!]_X = [\![Q]\!]_X \setminus A$.
2. $[\![P \parallel_Y Q]\!]_X = [\![P]\!]_X \parallel_Y [\![Q]\!]_X$.

Corollary 2.2. *Let P and Q be processes, $Y \subseteq \Sigma$ and $X \in \{T, SF, FD\}$. Then $[\![P \otimes_Y Q]\!]_X = [\![P]\!]_X \otimes_Y [\![Q]\!]_X$.*

2.4.8 Relationships between denotations

The following conditions always hold and concern relationships which exist between denotations in the various semantic models.

DR1 $\tau_\perp P = \tau P \cup \delta P$.

DR2 $\phi_\perp P = \phi P \cup \{(t, R) \mid t \in \delta P \ \wedge \ R \subseteq \Sigma\}$.

DR3 $\tau P = \{t \mid (t, \varnothing) \in \phi P\} \cup (\tau P \cap \delta P)$.

Note that $\tau P \cap \delta P$ gives those divergent traces which are actually generated operationally by P. In view of these conditions, the following result holds:

Proposition 2.3. *If $\delta P = \varnothing$ then:*

1. $\phi P = \phi_\perp P$.
2. $\tau_\perp P = \tau P = \{t \mid (t, \varnothing) \in \phi P\}$.

2.4.9 Alternative denotations in the failures divergences model

For a number of different reasons, we would prefer to work only with stable failures and traces even when working in the failures divergences model.[12] To facilitate this, we consider the notion of *minimally-divergent traces*, defined as follows for any process P:

Definition 2.2. $min\delta P \triangleq \{t \mid t \in \delta P \wedge (\nexists u \in \delta P)\ u < t\}$.

The minimally-divergent traces are a subset of those divergent traces which are generated operationally by the process under consideration rather than being present only by virtue of FD4. The following property always holds of them:

MD For any process P, $min\delta P \subseteq \tau P$.

Using $min\delta P$ allows us to deal with divergent traces in the same way as ordinary traces from τP. It is easy to see, by definition 2.2 and FD4, that the following result holds.

Proposition 2.4. $\delta P = \{t \circ v \mid t \in min\delta P \wedge v \in \Sigma^*\}$.

Using this result and DR2, $\phi_\perp P$ and δP may be reclaimed from ϕP and $min\delta P$.

2.5 Process alphabets

Process alphabets do not play any *semantic* role in the the treatment of CSP used here. Instead, they are simply used to define an upper bound on the set of events in which any process may engage and so a lower bound on the set of events which any process will *always* refuse. The alphabet of a process P, denoted αP, must always be such that $\beta(P) \subseteq \alpha P$, where $\beta(P)$ is calculated according to the rules in figure 2.5.[13] We are free to assign to αP any value we wish, provided that $\beta(P) \subseteq \alpha P$, although we will always explicitly state what we take the alphabet of a particular process to be before that alphabet is used for any purpose. Since $\beta(P) \subseteq \alpha P$, the following two conditions hold:

PA1 $\tau P \subseteq (\alpha P)^*$.

[12]The reasons for this choice are discussed at the relevant points in chapters 3 and 4.

[13]Process alphabets are used only (with respect to denotations) in the traces and stable failures models, which is why $\beta(DIV) = \varnothing$. The particular treatment given to recursive processes is necessary so that the procedure for deriving the alphabet of such a process will terminate.

$$
\begin{array}{lcl}
\beta(STOP) & \triangleq & \varnothing. \\
\beta(DIV) & \triangleq & \varnothing. \\
\beta(a \rightarrow P) & \triangleq & \{a\} \cup \beta(P). \\
\beta(P \,\Box\, Q) & \triangleq & \beta(P) \cup \beta(Q). \\
\beta(P \sqcap Q) & \triangleq & \beta(P) \cup \beta(Q). \\
\beta(P \parallel_Y Q) & \triangleq & \beta(P) \cup \beta(Q). \\
\beta(P \setminus A) & \triangleq & \beta(P) - A. \\
\beta(P[G]) & \triangleq & G(\beta(P)).
\end{array}
$$

Let $N_i = P_i$ for $1 \leq i \leq k$ be a recursive definition and let $\overline{STOP}$ denote the vector $\langle STOP, \ldots, STOP \rangle$ of length k. Then:

$$\beta(N_i) \triangleq \bigcup_{1 \leq j \leq k} \beta(P_j[\overline{STOP}/\overline{N}]).$$

Figure 2.5: Deriving alphabets

PA2 $\quad (t, R) \in \phi P \Longrightarrow (t, R \cup (\Sigma - \alpha P)) \in \phi P.$

PA2 is a consequence of PA1 and SF4 (impossible events can always be refused).

2.6 Useful algebraic laws

We include here some useful equivalences which exist between syntactic terms in all three semantic models, some of which have already been mentioned. Although they are not used formally, they are important in allowing us to provide as input to FDR2 process definitions which are less likely to suffer from state explosion when their operational semantics is calculated. ($=$ is used in the following equations to denote semantic equality in all three models.)

- $\Box$, $\sqcap$ and $|||$ are commutative and associative in all three models, meaning that they may be indexed by finite sets.
- Hiding is associative in that $(P \setminus A) \setminus A' = P \setminus (A \cup A')$.

- $P \setminus A = P$ if $\alpha P \cap A = \varnothing$.
- $(P \parallel_Y Q) \setminus A = (P \setminus A) \parallel_Y (Q \setminus A)$ if $Y \cap A = \varnothing$.

Parallel composition is symmetric but not associative in the general case. As a result, expressions involving parallel composition should always be bracketed appropriately.[14] It is, however, possible to define a weak associativity property, if we can guarantee that the two participants in a parallel composition will synchronize on at least those events that they have in common:

$$P \parallel_A (Q \parallel_B R) = (P \parallel_C Q) \parallel_D R$$

where

- $A = \alpha P \cap \alpha(Q \parallel_B R)$ and $\alpha(Q \parallel_B R) = \alpha Q \cup \alpha R$.
- $B = \alpha Q \cap \alpha R$.
- $C = \alpha P \cap \alpha Q$.
- $D = \alpha(P \parallel_C Q) \cap \alpha R$ and $\alpha(P \parallel_C Q) = \alpha P \cup \alpha Q$.

A similar result holds with the network composition operator, $\otimes$, substituted for $\parallel$.

2.7 Contexts and environments

The term "environment" will be used to denote a CSP process with which another CSP process might be composed. The term "environment" is to be distinguished from the term "context", where a context is a process term containing free process variables for which particular processes might be substituted. A component process is therefore *composed with* an environment (using an additional operator to combine the two processes) while a component is *placed in* a context (using substitution).

For our purposes, a process context F represents a syntactic term containing exactly one instance of each of the free process variables V_1 to V_n and only the parallel composition and hiding operators (or the network composition operator, $\otimes_Y$, which may be represented using only hiding and parallel composition). $F(P_1, \ldots, P_n)$ is used to denote the process where, for $1 \leq i \leq n$, process P_i has been substituted for variable V_i. This means that

[14]In chapter 3 we are usually able to dispense with such brackets and do so for the purposes of presentation; however, they are always used in chapter 7 and appendix D in the definition of processes to be used with FDR2.

$F(P_1, \ldots, P_n)$ is a *closed* term and so contains no further free (process) variables (although the P_i may contain process *names* which have been defined). Strictly speaking, $[\![F]\!]_X$ for $X \in \{T, SF, FD\}$ should be used to denote the semantic meaning of F (which is a mapping from a set of n process denotations to a process denotation). However, we shall simply denote it using F. This is because the defining expression $V_1 \parallel_Y V_2$ makes sense whether V_1 and V_2 represent syntactic terms or semantic objects; the same applies to $V_1 \setminus A$ and $V_1 \otimes_Y V_2$. In moving from the defining expression of (syntactic) F to the defining expression of $[\![F]\!]_X$, all that has changed is the type of the free variables. It will always be clear when it is used what sort of object is represented by F.

We also introduce a useful notation to be used with respect to processes built around a context F. During evaluation of the semantics of $F(P_1, \ldots, P_n)$, we evaluate the subterms of $F(P_1, \ldots, P_n)$ in a unique order and this ordering may be used to define a set of intermediate processes which are used to construct $F(P_1, \ldots, P_n)$. For example, in constructing $(P \parallel_Y Q) \parallel_Z P'$, the intermediate processes would be P, Q, $P \parallel_Y Q$ and $(P \parallel_Y Q) \parallel_Z P'$ itself. We therefore define $Imp(F(P_1, \ldots, P_n))$ in order to return this set of processes for $F(P_1, \ldots, P_n)$. (Note that $\uplus$ denotes the disjoint union operator.)

Definition 2.3. *For processes $P_1, \ldots, P_n$, and $A, Y \subseteq \Sigma$, $Imp(F(P_1, \ldots, P_n))$ is defined inductively as follows:*

- $$\begin{aligned} Imp(P_1 \setminus A) &\triangleq \{P_1, (P_1 \setminus A)\}. \\ Imp(P_1 \parallel_Y P_2) &\triangleq \{P_1, P_2, (P_1 \parallel_Y P_2)\}. \\ Imp(P_1 \otimes_Y P_2) &\triangleq \{P_1, P_2, (P_1 \otimes_Y P_2)\}. \\ Imp(F(P_1, \ldots, P_n) \setminus A) &\triangleq \{F(P_1, \ldots, P_n) \setminus A\} \cup \\ &\quad\; Imp(F(P_1, \ldots, P_n)). \end{aligned}$$

- *Where F', F'' are contexts,*
  $F(P_1, \ldots, P_n) = F'(P_{i_1}, \ldots, P_{i_m}) \parallel_Y F''(P_{j_1}, \ldots, P_{j_k})$
  and $\{1, \ldots, n\} = \{i_1, \ldots, i_m\} \uplus \{j_1, \ldots, j_k\}$:

  - $$\begin{aligned} Imp(F(P_1, \ldots, P_n)) \triangleq\ & \{(F(P_1, \ldots, P_n)\} \cup \\ & Imp(F'(P_{i_1}, \ldots, P_{i_m})) \cup \\ & Imp(F''(P_{j_1}, \ldots, P_{j_k})). \end{aligned}$$

- *Where F', F'' are contexts,*
  $F(P_1, \ldots, P_n) = F'(P_{i_1}, \ldots, P_{i_m}) \otimes_Y F''(P_{j_1}, \ldots, P_{j_k})$
  and $\{1, \ldots, n\} = \{i_1, \ldots, i_m\} \uplus \{j_1, \ldots, j_k\}$:

- $Imp(F(P_1,\ldots,P_n)) \triangleq \{(F(P_1,\ldots,P_n)\} \cup$
  $Imp(F'(P_{i_1},\ldots,P_{i_m})) \cup$
  $Imp(F''(P_{j_1},\ldots,P_{j_k}))$.

This notation will be used to reclaim the operators and processes which are used to define any process $F(P_1,\ldots,P_n)$.

2.8 Maximality and monotonicity

2.8.1 Maximality of failures

Let $\mathcal{F}$ be a set of failures. We define two notions of maximality with regard to the elements of that set.

- $(t, R) \in \mathcal{F}$ is *refusal-maximal* if and only if there does not exist $(t, X) \in \mathcal{F}$ such that $R \subset X$.
- $(t, R) \in \mathcal{F}$ is *maximal* if and only if there does not exist $(s, X) \in \mathcal{F}$ such that $t < s$ or such that $t = s$ and $R \subset X$.

We shall denote $max(\mathcal{F}) = \{(t, R) \in \mathcal{F} \mid (t, R) \text{ is maximal }\}$. In the event that $max(\mathcal{F}) = \{(t, R)\}$ for some failure (t, R), we write $max(\mathcal{F}) = (t, R)$. Refusal-maximality is mainly used in the statement and proofs of results from chapter 3. Its important property is captured by the following result, which follows directly from SF2 and SF4:

Proposition 2.5. *If $(t, R) \in \phi Q$ is refusal-maximal, then $t \circ \langle a \rangle \in \tau Q$ for every $a \in (\Sigma - R)$.*

The notion of maximality is also used in the proofs of results from chapter 3.

2.8.2 Monotonicity

The following are standard results which may be proved straightforwardly.

Proposition 2.6. *For traces s and u and $A \subseteq \Sigma$, if $s \leq u$ then $s \setminus A \leq u \setminus A$.*

We define the ordering $\preceq$ over sets of traces $\mathcal{T}$, $\mathcal{T}'$ such that $\mathcal{T} \preceq \mathcal{T}'$ if and only if, for every $t \in \mathcal{T}$, there exists $u \in \mathcal{T}'$ such that $t \leq u$.

Proposition 2.7. *Let $Y \subseteq \Sigma$ and s, u, v, w be traces such that $v \leq s$, $w \leq u$ and $s \parallel_Y u \neq \varnothing$. Then $(v \parallel_Y w) \preceq (s \parallel_Y u)$.*

Proposition 2.8. *Let $A, Y \in \Sigma$ and $\mathcal{P}, \mathcal{P}', \mathcal{Q}, \mathcal{Q}'$ be either sets of traces; pairs of sets of traces and sets of stable failures; or pairs of sets of failures and sets of divergences.*

1. *If $\mathcal{P} \subseteq \mathcal{P}'$, then $\mathcal{P} \setminus A \subseteq \mathcal{P}' \setminus A$.*
2. *If $\mathcal{P} \subseteq \mathcal{P}'$ and $\mathcal{Q} \subseteq \mathcal{Q}'$, then $\mathcal{P} \parallel_Y \mathcal{Q} \subseteq \mathcal{P}' \parallel_Y \mathcal{Q}'$.*
3. *If $\mathcal{P} \subseteq \mathcal{P}'$ and $\mathcal{Q} \subseteq \mathcal{Q}'$, then $\mathcal{P} \otimes_Y \mathcal{Q} \subseteq \mathcal{P}' \otimes_Y \mathcal{Q}'$.*

Corollary 2.9. *Let P, P, Q, Q' be processes, $X \in \{T, SF, FD\}$ and $A, Y \in \Sigma$.*

1. *If $[\![P]\!]_X \subseteq [\![P']\!]_X$, then $[\![P \setminus A]\!]_X \subseteq [\![P' \setminus A]\!]_X$.*
2. *If $[\![P]\!]_X \subseteq [\![P']\!]_X$ and $[\![Q]\!]_X \subseteq [\![Q']\!]_X$, then $[\![P \parallel_Y Q]\!]_X \subseteq [\![P' \parallel_Y Q']\!]_X$.*
3. *If $[\![P]\!]_X \subseteq [\![P']\!]_X$ and $[\![Q]\!]_X \subseteq [\![Q']\!]_X$, then $[\![P \otimes_Y Q]\!]_X \subseteq [\![P' \otimes_Y Q']\!]_X$.*

2.9 Determinism

Definition 2.4 (Determinism). *A process P is* deterministic *if:*

1. *$\delta P = \varnothing$.*
2. *$\phi P = \{(t, R) \mid t \in \tau P \;\wedge\; R \subseteq (\Sigma - \{a \mid t \circ \langle a \rangle \in \tau P\})\}$.*

If P is deterministic, it is completely characterised by τP; in particular, it always responds in the same manner to the same external stimulus. We are interested in the property of determinism with respect to processes which might be used during verification using FDR2 and so which can be represented operationally using a finite-state LTS. (Note that FDR2 may be used to check for the determinism of any such process.) Any such "finite-state" process which is deterministic may be represented syntactically using only indexed deterministic choice, the prefix operator, recursion and $STOP$. Before showing how to construct such a representation, we introduce some additional notation, where $\mathcal{T}$ is a prefix-closed set of traces:

- $init(\mathcal{T}) \triangleq \{a \mid \langle a \rangle \circ s \in (\mathcal{T} - \{\langle\rangle\})\}$ gives the initial events of all traces from $\mathcal{T}$.
- $aft(\mathcal{T}, a) \triangleq \{s \mid \langle a \rangle \circ s \in \mathcal{T}\}$ gives the set of traces from $\mathcal{T}$ which are possible after the "execution" of a.

For a prefix-closed set of traces $\mathcal{T}$, $N_\mathcal{T}$ is defined as:

$$N_\mathcal{T} = \Box_{a \in init(\mathcal{T})}(a \rightarrow N_{aft(\mathcal{T}, a)}) \text{ where } N_{\{\langle\rangle\}} = STOP.$$

The following condition always holds for "finite-state" P:

DE If P is deterministic, then $P =_X N_{\tau P}$ for $X \in \{T, SF, FD\}$.

2.10 Constructing processes

In the proofs of results from chapter 3, it is sometimes necessary to construct syntactic processes with exactly specified semantics (in the traces or stable failures models). In this section, we show how to do this. Before proceeding to define the specific processes which we shall need, we define some useful sub-processes. In the following definitions, $a \in \Sigma$ is an event, s is a trace and $R \subseteq \Sigma$ is a set of events.

- $TP_{\langle\rangle} = DIV$.
- $TP_{\langle a\rangle \frown s} = (a \rightarrow TP_s) \, \Box \, DIV$.
- $FP_{(\langle\rangle, R)} = \Box_{a \in (\Sigma - R)}(a \rightarrow DIV)$.
- $FP_{(\langle a\rangle \frown s, R)} = (a \rightarrow FP_{(s,R)}) \, \Box \, DIV$.

TP and *FP* are standard constructions taken from [63] (this is why the statements of their semantics in propositions 2.10 and 2.11 below are given without proof). Before proceeding to give the semantics of these processes we observe an important property of *DIV* in the stable failures model, which is important to us here and also in the construction of processes which are used in chapter 6 for the purposes of automatic verification. Recall that $\phi DIV = \varnothing$ and, according to figure 2.3, the stable failures semantics of $\Box$ is defined as follows:

$$\begin{aligned} \phi(P \, \Box \, Q) \triangleq \quad & \{(\langle\rangle, X) \mid (\langle\rangle, X) \in \phi P \cap \phi Q\} \cup \\ & \{(s, X) \mid (s, X) \in \phi P \cup \phi Q \ \wedge \ s \neq \langle\rangle\}. \end{aligned}$$

Hence, for any process P,

$$\phi(P \Box DIV) = \{(t, R) \mid (t, R) \in \phi P \ \wedge \ t \neq \langle\rangle\}.$$

We can therefore use *DIV*, along with the deterministic choice operator, to effectively remove (stable) failures when necessary. The traces, stable failures and alphabets[15] of TP_t and $FP_{(t,R)}$ are then as follows.

Proposition 2.10. *Let t be a trace. Then the following hold:*

1. $\tau TP_t = Pref(t)$.
2. $\phi TP_t = \varnothing$.
3. $\beta(TP_t) = events(t)$.

[15]Alphabets are calculated here using β and so using the detail in figure 2.5

Proposition 2.11. *Let (t, R) be a failure. Then the following hold:*

1. $\tau FP_{(t,R)} = \mathit{Pref}(t) \cup \{t \circ \langle a \rangle \mid a \in (\Sigma - R)\}$.
2. $\phi FP_{(t,R)} = \{(t, X) \mid X \subseteq R\}$.
3. $\beta(FP_{(t,R)}) = \mathit{events}(t) \cup (\Sigma - R)$.

Note that, in deriving results on the remainder of the constructions given here, we shall always appeal implicitly to the detail from section 2.4 and that from figure 2.5.

2.10.1 Finite set of traces

Where $\mathcal{T}$ is a finite, non-empty set of traces (not necessarily prefix-closed),

$$FST\langle \mathcal{T} \rangle \triangleq \Box_{t \in \mathcal{T}} TP_t.$$

The identifier FST is used to indicate *finite set of traces.* In the event that $\mathcal{T}$ is the singleton set $\{t\}$, we may use $FST\langle t \rangle$ in lieu of $FST\langle \mathcal{T} \rangle$. The necessary properties of $FST\langle \mathcal{T} \rangle$ are then given by the following result, which follows easily from proposition 2.10.

Proposition 2.12. *Let $\mathcal{T}$ be a finite, non-empty set of traces. Then the following hold:*

1. $\tau FST\langle \mathcal{T} \rangle = \mathit{Pref}(\mathcal{T})$.
2. $\beta(FST\langle \mathcal{T} \rangle) = \bigcup_{t \in \mathcal{T}} \mathit{events}(t)$.

2.10.2 Single failure and finite set of traces

We now show how to construct a process around a single failure and a finite, non-empty set of traces. We use the identifier SFT here to indicate *single failure and traces* and, where $\mathcal{T}$ is a finite, non-empty set of traces and (t, R) is a failure, define:

$$SFT\langle (t, R), \mathcal{T} \rangle \triangleq FP_{(t,R)} \sqcap (\Box_{u \in \mathcal{T}} TP_u).$$

The necessary properties of $SFT\langle (t, R), \mathcal{T} \rangle$ are given by the following result, which follows from propositions 2.10 and 2.11.

Proposition 2.13. *Let $\mathcal{T}$ be a finite, non-empty set of traces and (t, R) be a failure. Then the following hold:*

1. $\tau SFT\langle (t, R), \mathcal{T} \rangle = \mathit{Pref}(\mathcal{T}) \cup \mathit{Pref}(t) \cup \{t \circ \langle a \rangle \mid a \in (\Sigma - R)\}$.
2. $\phi SFT\langle (t, R), \mathcal{T} \rangle = \{(t, X) \mid X \subseteq R\}$.
3. $\beta(SFT\langle (t, R), \mathcal{T} \rangle) = \mathit{events}(t) \cup (\Sigma - R) \cup \bigcup \{\mathit{events}(u) \mid u \in \mathcal{T}\}$.

2.10.3 Refusal-maximal failure and traces of specified process

Before proceeding to give the definition of the process which is used here, we observe the following standard result (recalling that $|||$ denotes $\|_{\varnothing}$).

Proposition 2.14. *The following hold:*

1. $\tau(P|||DIV) = \tau P$.
2. $\phi(P|||DIV) = \varnothing$.
3. $\beta(P|||DIV) = \beta(P)$.

We now show how to construct a process around a single failure (which will be refusal-maximal in practice) and the traces of a specified process. We use the identifier *MFP* here to indicate *maximal failure and process* and, where P is a process and (t, R) is a failure, define:

$$MFP\langle(t,R),P\rangle \triangleq FP_{(t,R)} \sqcap (P|||DIV).$$

The necessary properties of $MFP\langle(t,R),P\rangle$ are given by the following result.

Proposition 2.15. *Let P be a process and $(t,R) \in \phi P$ be refusal-maximal. Then the following hold:*

1. $\tau MFP\langle(t,R),P\rangle = \tau P$.
2. $\phi MFP\langle(t,R),P\rangle = \{(t,X) \mid X \subseteq R\}$.
3. $\beta(MFP\langle(t,R),P\rangle) = \beta(P)$.

Proof. 1. By proposition 2.11(1) and proposition 2.14(1), we observe that

$$\tau MFP\langle(t,R),P\rangle = \tau P \cup Pref(t) \cup \{t \circ \langle a\rangle \mid a \in (\Sigma - R)\}.$$

Since $(t,R) \in \phi P$, we have that $t \in \tau P$ by condition SF2 and so $Pref(t) \subseteq \tau P$ by SF1. Moreover, we have that $\{t \circ \langle a\rangle \mid a \in (\Sigma - R)\} \subseteq \tau P$ by proposition 2.5. The proof of this part follows.

2. The proof of this part follows by proposition 2.11(2) and proposition 2.14(2).

3. By proposition 2.11(3) and proposition 2.14(3), we observe that

$$\beta(MFP\langle(t,R),P\rangle) = \beta(P) \cup events(t) \cup (\Sigma - R).$$

By the proof of part 1 of the proposition,

$$\{t\} \cup \{t \circ \langle a\rangle \mid a \in (\Sigma - R)\} \subseteq \tau P.$$

If we take $\alpha P = \beta(P)$, it follows by PA1 that $\tau P \subseteq \beta(P)^*$ and hence $events(t) \cup (\Sigma - R) \subseteq \beta(P)$. □

2.10.4 Refusal-maximal failure, finite set of traces and process

Finally, we show how to construct a process around a single failure (which will be refusal-maximal in practice), a finite, non-empty set of traces, and the traces and failures of a specified process. We use the identifier *MFTP* here to indicate *maximal failure, traces and process* and, where P is a process, $\mathcal{T}$ is a finite, non-empty set of traces and (t, R) is a failure, define:

$$MFTP\langle (t,R), \mathcal{T}, P\rangle \triangleq SFT\langle (t,R), \mathcal{T}\rangle \sqcap P$$

The necessary properties of $MFTP\langle (t,R), \mathcal{T}, P\rangle$ are given by the following result.

Proposition 2.16. *Let P be a process and $\mathcal{T}$ a finite, non-empty set of traces. Let the failure (t, R) be such that $(t, X) \in \phi P$ is refusal-maximal and $X \subseteq R$. Then the following hold:*

1. $\tau MFTP\langle (t,R), \mathcal{T}, P\rangle = Pref(\mathcal{T}) \cup \tau P$
2. $\phi MFTP\langle (t,R), \mathcal{T}, P\rangle = \{(t,Z) \mid Z \subseteq R\} \cup \phi P.$
3. $\beta(MFTP\langle (t,R), \mathcal{T}, P\rangle) = \beta(P) \cup \bigcup\{events(u) \mid u \in \mathcal{T}\}.$

Proof. 1. By proposition 2.13(1),

$$\tau MFTP\langle (t,R), \mathcal{T}, P\rangle = Pref(\mathcal{T}) \cup Pref(t) \cup \{t \circ \langle a\rangle \mid a \in (\Sigma - R)\} \cup \tau P.$$

By a proof similar to that of part 1 of proposition 2.15 and since $X \subseteq R$, we have that $Pref(t) \cup \{t \circ \langle a\rangle \mid a \in (\Sigma - R)\} \subseteq \tau P.$

2. The proof of this part follows by proposition 2.13(2).

3. By proposition 2.13(3), $\beta(MFTP\langle (t,R), \mathcal{T}, P\rangle)$ is given by:

$$\beta(P) \cup events(t) \cup (\Sigma - R) \cup \bigcup\{events(u) \mid u \in \mathcal{T}\}.$$

By a proof similar to that of part 3 of proposition 2.15 and since $X \subseteq R$, we have that $events(t) \cup (\Sigma - R) \subseteq \beta(P)$. □

2.11 Further consideration of parallel composition

In the traces and stable failures models, we almost always use a restricted form of parallel composition, where processes (and sets of behaviours) have to synchronize on at least those events in which they can both engage. The following results concern the semantics of this restricted form of parallel composition. In general, they will be appealed to implicitly when needed.

2.11.1 Traces

Theorem 2.17. *Let P, Q be processes and $Y = \alpha P \cap \alpha Q$. Then:*

$$\begin{aligned} \tau(P \parallel_Y Q) \;=\; & \{t \in (\alpha P \cup \alpha Q)^* \mid (\exists s \in \tau P, u \in \tau Q)\; t\lceil \alpha P = s \\ & \wedge\; t\lceil \alpha Q = u\}. \end{aligned}$$

Proof. The proof in both directions proceeds by a straightforward induction on the length of traces using PA1, the definition of $\parallel_Y$ given in figure 2.2 and the fact that:

- If $a \in Y$ then $a \in \alpha P$ and $a \in \alpha Q$.
- If $a \notin Y$ then a cannot be in both αP and αQ.

□

2.11.2 Stable failures

A similar result is given here with respect to the stable failures of (syntactic) parallel compositions. However, we first give a more generic result in terms of parallel composition of sets of failures. This allows us to prove what we need here and is also reused in chapter 3. Since alphabets are calculated syntactically and so cannot be generated as such for an arbitrary set of failures, $\mathcal{R}$ is used to capture the property of process alphabets which is crucial here.

Definition 2.5. *Let $\mathcal{F}$ be a set of stable failures. Then $\mathcal{R}(\mathcal{F})$ is the set of all $A \subseteq \Sigma$ such that*

$$(t, R) \in \mathcal{F} \Longrightarrow (t, R \cup (\Sigma - A)) \in \mathcal{F}.$$

The following result shows that any parallel composition of subset-closed sets of failures also enjoys the property of subset-closure.

Proposition 2.18. *If $\mathcal{F}_1$, $\mathcal{F}_2$ are subset-closed sets of stable failures and $Y \subseteq \Sigma$, then $\mathcal{F}_1 \parallel_Y \mathcal{F}_2$ is also subset-closed.*

Proof. Let $(t, R) \in \mathcal{F}_1 \parallel_Y \mathcal{F}_2$ and $Z \subseteq R$. We show that $(t, Z) \in \mathcal{F}_1 \parallel_Y \mathcal{F}_2$. By definition of $\parallel_Y$ in section 2.4.2, there are $(s, S) \in \mathcal{F}_1$, $(u, U) \in \mathcal{F}_2$ such that:

$$t \in (s \parallel_Y u),\ R = S \cup U \text{ and } S - Y = U - Y.$$

Let $S' = S \cap Z$ and $U' = U \cap Z$. Then $S' - Y = U' - Y$. Moreover, since $Z \subseteq S \cup U$, $S' \cup U' = (S \cap Z) \cup (U \cap Z) = (S \cup U) \cap Z = Z$. Hence, the only thing left to prove is that $(s, S') \in \mathcal{F}_1$ and $(u, U') \in \mathcal{F}_2$, which follows by the subset-closure of $\mathcal{F}_1$ and $\mathcal{F}_2$. □

We now give the generic result, before using it to prove the final result we want.

Proposition 2.19. *Let $\mathcal{F}_1$, $\mathcal{F}_2$ be subset-closed sets of stable failures, $A_1 \in \mathcal{R}(\mathcal{F}_1)$, $A_2 \in \mathcal{R}(\mathcal{F}_2)$ and $A_1 \cap A_2 = Y$. Then:*

$$\begin{array}{rcl} \mathcal{F}_1 \parallel_Y \mathcal{F}_2 & = & \{(t, S \cup U \cup Z) \mid Z \subseteq (\Sigma - (A_1 \cup A_2)) \wedge \\ & & ((\exists (s,S) \in \mathcal{F}_1, (u,U) \in \mathcal{F}_2)\ t \in (s \parallel_Y u) \wedge \\ & & S \subseteq A_1 \ \wedge\ U \subseteq A_2)\} \end{array}$$

Proof. ($\subseteq$) Let $(t,R) \in (\mathcal{F}_1 \parallel_Y \mathcal{F}_2)$. Moreover, let $C = R \cap (A_1 \cup A_2)$, $D = R - (A_1 \cup A_2)$ and so $C \cup D = R$. By proposition 2.18, $(t,C) \in (\mathcal{F}_1 \parallel_Y \mathcal{F}_2)$. By definition of $\parallel_Y$ in section 2.4.2, there are $(s,S') \in \mathcal{F}_1$ and $(u,U') \in \mathcal{F}_2$ such that:

$$t \in (s \parallel_Y u),\ C = S' \cup U' \text{ and } S' - Y = U' - Y.$$

The latter and $A_1 \cap A_2 = Y$ means that $S' - A_1 = U' - A_1$ and $S' - A_2 = U' - A_2$.

Let $S = S' - (A_2 - A_1)$ and $U = U' - (A_1 - A_2)$. We have $S \subseteq A_1$ and $U \subseteq A_2$ and, by the subset-closure of $\mathcal{F}_1$ and $\mathcal{F}_2$, $(s,S) \in \mathcal{F}_1$ and $(u,U) \in \mathcal{F}_2$. Hence, since $D \subseteq (\Sigma - (A_1 \cup A_2))$, the only thing to prove is that $S' \cup U' = S \cup U$. We have:

$$\begin{array}{rcl} S \cup U & = & (S' - (A_2 - A_1)) \cup (U' - (A_1 - A_2)) \\ & = & (S' - A_2) \cup (S' \cap A_1 \cap A_2) \cup (U' - A_1) \\ & & \cup\, (U' \cap A_1 \cap A_2) \\ & = & (S' - A_2) \cup (U' - A_2) \cup (S' \cap A_1 \cap A_2) \cup \\ & & (U' - A_1) \cup (S' - A_1) \cup (U' \cap A_1 \cap A_2) \\ & = & S' \cup U'. \end{array}$$

($\supseteq$) Let t, (s,S), (u,U), Z be as in the definition of $\mathcal{X}$. We have to show that $(t, S \cup U \cup Z) \in (\mathcal{F}_1 \parallel_Y \mathcal{F}_2)$, where $\parallel_Y$ is as defined in section 2.4.2.

Let $S' = Z \cup S \cup (U - Y)$ and $U' = Z \cup U \cup (S - Y)$. Since $U \subseteq A_2$ and $Y = A_1 \cap A_2$, $(U - Y) \cap A_1 = \varnothing$. Hence, $(Z \cup (U - Y)) \subseteq (\Sigma - A_1)$ and so $(s,S') \in \mathcal{F}_1$ by definition 2.5 and by the subset-closure of $\mathcal{F}_1$. Similarly, $(u,U') \in \mathcal{F}_2$. Moreover,

$$S' \cup U' = Z \cup S \cup (U - Y) \cup Z \cup U \cup (S - Y) = Z \cup S \cup U.$$

Hence, the only thing we need to show is that $S' - Y = U' - Y$. In other words, that

$$(Z \cup S \cup (U - Y)) - Y = (Z \cup U \cup (S - Y)) - Y$$

which is equivalent to

$$Z \cup (S - Y) \cup (U - Y) = Z \cup (U - Y) \cup (S - Y)$$

which clearly holds. $\square$

Theorem 2.20. *Let P and Q be processes and $Y = \alpha P \cap \alpha Q$. Then:*

$$\begin{aligned}\phi(P \parallel_Y Q) \;=\; & \{(t, S \cup U \cup Z) \mid Z \subseteq (\Sigma - (\alpha P \cup \alpha Q)) \wedge \\ & ((\exists (s,S) \in \phi P, (u,U) \in \phi Q)\ t \in (s \parallel_Y u) \wedge \\ & S \subseteq \alpha P \ \wedge\ U \subseteq \alpha Q)\}\end{aligned}$$

Proof. By the definition in figure 2.3, $\phi(P \parallel_Y Q) = \phi P \parallel_Y \phi Q$. By SF3, ϕP and ϕQ are both subset-closed. By PA2, $\alpha P \in \mathcal{R}(\phi P)$ and $\alpha Q \in \mathcal{R}(\phi Q)$. The proof follows by proposition 2.19. $\square$

2.12 Model-checking CSP

The tool FDR2 may be used to perform model-checking of CSP processes. Specifically, we may check whether or not one process refines another in each of the three semantic models, as well as performing checks for determinism, deadlock-freedom and divergence-freedom. The tool takes as input a text file containing process descriptions written in the machine-readable dialect of CSP. (See [63] or the FDR2 manual ([64]) for details.) Any operators used in this thesis to define processes to be supplied as input to FDR2 have a direct counterpart in the machine-readable syntax.

2.12.1 Additional operators

There are two operators and one construct which are used in defining processes in chapter 7 and appendix D which have not yet been introduced. (Note that they are not used in any processes for which we have to derive a formal semantics.)

if $\mathcal{B}$ then P else Q is the process which behaves like P if the boolean expression $\mathcal{B}$ evaluates to true and otherwise behaves like Q. (In defining boolean expressions, we use tests for equality, $==$, tests for inequality, $!=$, and the connectives "and" and "or".)

Where c is a channel and x is a variable, $c?x \rightarrow P(x)$ denotes the process which waits for input on channel c; when it receives a value v on c, it then proceeds to behave as P with v substituted for x in P. Due to the finiteness of channel data types, $c?x \rightarrow P(x)$ may be represented as $\Box_{c.x \in \alpha c} c.x \rightarrow P(x)$. A more general version of this construct is also used, where data transfer can be in several directions at once. For example, where the type of channel c is given by the types $T_1, \ldots, T_5$, and x_i is a variable and $v_j \in T_j$ a concrete value for $1 \leq i, j \leq 5$,

$$c?x_1!v_2!v_3?x_4!v_5 \rightarrow P(x_1, x_4)$$

may be used to input values into the variables x_1 and x_4 and to communicate the concrete data values v_2, v_3 and v_5: in such a data transfer, ? is used to denote the input of a value into the variable immediately to its right; ! is used to denote the communication or output of the concrete value immediately to its right. $c?x_1!v_2!v_3?x_4!v_5 \rightarrow P(x_1, x_4)$ may be represented as:

$$\Box_{x_1 \in T_1, x_4 \in T_4}(c.x_1.v_2.v_3.x_4.v_5 \rightarrow P(x_1, x_4)).$$

Note also that ! should only be used to the right of the first occurrence of ?. Thus,

$$c.v_1.v_2.v_3?x_4!v_5 \rightarrow P(x_4)$$

is the correct way to write the process which, on its first event, communicates the concrete values v_1, v_2, v_3 and v_5 and reads data into the variable x_4, while

$$c!v_1!v_2!v_3?x_4!v_5 \rightarrow P(x_4)$$

is incorrect.

The other new construct used is the *let.....within* notation, used to make definitions local to an expression. It is generally used in the following manner:

$$\begin{aligned} P(x, y, V) \quad = \quad & \\ & \textit{let } P(v) = \ldots \\ & \textit{within } P(V). \end{aligned}$$

In such a case, we are effectively defining the process $P(v)$, which will have v initialised to V and within which x and y are constants, for example channel names. It allows us to easily define a family of processes which differ only in the values stored by the constants x and y. (See chapter 7 for examples of this.)

Further details on all of these language features can be found in [63].

2.12.2 Channels and data types

The following are the constructs which we shall need to declare channels and data types in FDR2. A channel c with data type d is declared as follows:

$$\textit{channel } c : d$$

A set may be assigned an identifier as in the construct:

$$data = \{0, 1\}$$

data is then regarded as a data type. The data type d which consists of the product of the data types $d_1, d_2 \ldots, d_n$ may be declared:

$$datatype\ d = d_1.d_2 \ldots d_{n-1}.d_n$$

Note also that $d_1.d_2 \ldots d_{n-1}.d_n$ may be regarded itself as a data type. Finally, the data type d which consists of the constants $C_1, C_2, \ldots, C_n$ may be declared:

$$datatype\ d = C_1 \mid C_2 \mid \ldots \mid C_n$$

Note that none of the constants C_i need exist prior to such a declaration.

2.12.3 Immediately diverging process

The immediately diverging process does not have a distinguished syntactic representation in FDR2 and so we define *DIV* thus:

$$DIV \triangleq X \setminus \{a\} \qquad \text{where} \qquad X = a \rightarrow X.$$

Processes which include *DIV* in their definition may still be treated as guarded (even though the definition given here for it introduces hiding), since we have a direct statement of its semantics and so may still regard it as a syntactic constant.

2.13 Running example

We are now in a position to give a CSP rendering of the example processes given in figure 1.1. They are used in the remainder of the thesis as an aid to explanation. We assume that channels *in*, *out*, *send* and *data* all communicate values from $\{0, 1\}$; we also assume that all events which may be communicated on channels *in* and *out* are finally visible. The specification network from the figure, which we denote *SpecNet*, is defined as follows

$$SpecNet \triangleq LeftSpec \otimes_{a_{send}} RightSpec$$

where

- $LeftSpec = \Box_{v \in \{0,1\}}(in.v \rightarrow send.v \rightarrow LeftSpec)$.
- $RightSpec = \Box_{v \in \{0,1\}}(send.v \rightarrow out.v \rightarrow RightSpec)$.

The specification network effectively functions, therefore, as a two-slot buffer. The implementation network, *ImplNet*, is defined as:

$$\mathit{ImplNet} \triangleq \mathit{LeftImpl} \otimes_{(\alpha data \cup \alpha ack)} \mathit{RightImpl}$$

where

$$\mathit{LeftImpl} = \Box_{v \in \{0,1\}}(in.v \rightarrow data.v \rightarrow LI(v))$$
$$LI(x) = (ack.yes \rightarrow \mathit{LeftImpl}) \Box (ack.no \rightarrow data.x \rightarrow \mathit{LeftImpl})$$

and

$$\mathit{RightImpl} = \Box_{v \in \{0,1\}}(data.v \rightarrow (RI(v) \sqcap RI'))$$
$$RI(x) = ack.yes \rightarrow out.x \rightarrow \mathit{RightImpl}$$
$$RI' = ack.no \rightarrow \Box_{w \in \{0,1\}}(data.w \rightarrow out.w \rightarrow \mathit{RightImpl}).$$

Here, *LeftImpl* sends on the channel *data* the value it has just received; in the event that a negative acknowledgement is subsequently received (i.e. *ack.no* occurs) then the value is resent on *data*. The non-deterministic choice operator in *RightImpl* is used to model the possibility that the message transmission on *data* may be lost or corrupted: if it is lost or corrupted, then *RightImpl* communicates the event *ack.no* and waits for the value to be resent; otherwise it communicates *ack.yes* and outputs on the channel *out* the value it has just received.

Chapter 3

Towards a theory of refinement-after-hiding

In the papers [39], [40] and [12] one can observe the evolution of a notion of refinement-after-hiding which has its roots in [49] and an attempt to present a formal notion of what it means for one process to be a valid implementation of another when *replication*[1] is used as a reification technique. Two main issues were raised on completion of the work in [12]. Firstly, the notion of refinement-after-hiding presented there could not be used to verify compositionally that an implementation network refined a specification network in terms of standard CSP refinement in anything other than the traces model. Once failures were introduced, the refinement relation whose existence could be proved was non-standard. Secondly, the conceptualisation of the existing notion was based quite closely on the fault-tolerance mechanisms, such as replication, which had inspired it originally. This had the effect that there was no characterisation in the most general sense of what it meant to be a notion of refinement-after-hiding. One of the major consequences of this was that it was not clear which parts of the existing framework were absolutely necessary and which could be dispensed with or altered.

The work in this chapter aims, therefore, to address each of these issues. We take from [12] the most fundamental features of the treatment given there. These are essentially the use of an interpretive mapping, along with certain restrictions on the sets which may be hidden and on which parallel composition may occur as we build our implementation networks (these restrictions are stated in section 3.2.6). We then present a statement of what it means to constitute a notion of refinement-after-hiding in a general sense, along with a basic set of conditions on our interpretive mapping which are sufficient to guarantee that it may be used as a basis for such a notion. From these rather abstract conditions we derive in each of the three

[1]See [49] for a description of replication.

semantic models a set of more detailed conditions which themselves define a notion of refinement-after-hiding. At a stroke this generates a solution to the problems encountered when working in models incorporating failures. It also gave, among other things, a framework within which extensions and improvements to the work in [12] could be considered; this issue is discussed at greater length in chapter 4, where such extensions and improvements are presented.

3.1 The basic framework

We assume the existence of a *semantic* mapping,[2] λ, from process denotations to sets of behaviours or pairs of sets of behaviours as appropriate.[3] Intuitively, λ transforms an implementation process so that the resulting behaviours may be compared directly with those of the specification.[4] It is best regarded as a *meta-mapping*, representing all possible concrete mappings which we might use in practice. We use $Q \sqsupseteq_X^\lambda P$ to indicate that the process Q refines-after-hiding the process P, under the mapping λ, in the semantic model $X \in \{T, SF, FD\}$ and define it as follows.

Definition 3.1. *$Q \sqsupseteq_X^\lambda P$ if and only if $\lambda([\![Q]\!]_X)$ is defined and $\lambda([\![Q]\!]_X) \subseteq [\![P]\!]_X$.*

Component processes will be placed in context to form a network and we shall restrict the form of the contexts which may be used so that networks may be built only from component processes and the hiding and parallel composition operators. A (syntactic) context, *Con*, may be defined using the following grammar, where V, V_1, V_2 represent process variables and A, Y represent sets of events.[5]

[2]We choose to work with a semantic rather than a syntactic mapping for a number of reasons. These are considered in chapter 5, in the context of a discussion of related work.

[3]The sets of behaviours returned will be traces, failures or divergences as appropriate.

[4]This implies that λ is an interpretive mapping which makes behaviours more abstract, since "specification behaviours" are usually abstract and "implementation behaviours" more concrete. However, there is nothing in the theory developed here which requires this and so λ may also be used to interpret abstract behaviours at a more concrete level: this might be necessary if we wanted to show that a particular specification network refined the corresponding implementation network in order to show that the two were equivalent. As a result, it is perhaps best to view "implementations" as simply those processes which are interpreted using λ and "specifications" as those processes in whose behaviours we check for containment of those interpreted "implementation" behaviours.

[5]Strictly speaking, brackets should also be placed around any parallel composition. However, their absence will not cause us any problems in this chapter and so we omit them for purposes of presentation. (Recall that the hiding operator is left-associative and so it need not be bracketed.)

$$Con = V\backslash A \mid V_1 \|_Y V_2 \mid Con\backslash A \mid Con\|_Y Con \mid Con \|_Y V \mid V \|_Y Con.$$

In order to relate an implementation context to the corresponding specification context, we overload the mapping, λ, and apply it to that implementation context. This means that λ must be defined over contexts, which, in turn, necessitates its definition over the parallel composition and hiding operators. The effect of applying λ to a context is defined recursively in figure 3.1; for $A, Y \subseteq \Sigma$, $\lambda(\backslash A)$ returns $\backslash B$ for some set of events B and $\lambda(\|_Y)$ returns $\|_Z$ for some set of events Z (we shall see in section 3.3 how to characterize B and Z exactly).

Definition 3.2. *Let $A, Y \subseteq \Sigma$. Then $\lambda(\backslash A) = \backslash B$ and $\lambda(\|_Y) = \|_Z$, for some $B, Z \subseteq \Sigma$.*

Effectively, λ transforms a context by transforming in turn each operator contained therein: more specifically, it transforms the set of events with which the relevant operator is parameterized to reflect the fact that implementation processes may be expressed at a different level of abstraction to specification processes. Note that definition 3.2 defines λ over both the *syntactic* and *semantic* operators[6] of hiding and parallel composition, since the textual representation of the syntactic version of either hiding or parallel composition is the same as that of its semantic counterpart. Of course, we assume that λ applied to a syntactic operator returns a syntactic operator and λ applied to a semantic operator returns a semantic operator.

We introduce the notation *Fvis* to denote the set of finally visible events, in which both implementation and specification processes may engage. In order for $\sqsupseteq^{\lambda}_{X}$ for $X \in \{T, SF, FD\}$ to constitute an acceptable notion of refinement-after-hiding, the following condition must then be met, where F_{impl} and F_{spec} — each containing n process variables — are implementation and specification contexts respectively and $F_{spec} \triangleq \lambda(F_{impl})$; also, $Q_1, \ldots, Q_n$ and $P_1, \ldots, P_n$ are processes.

Condition 1. *If $Q_i \sqsupseteq^{\lambda}_{X} P_i$ for $1 \leq i \leq n$ and $\alpha F_{impl}(Q_1, Q_2, \ldots, Q_n) \subseteq Fvis$, then $F_{impl}(Q_1, Q_2, \ldots, Q_n) \sqsupseteq_X F_{spec}(P_1, P_2, \ldots, P_n)$.*

For $1 \leq i \leq n$, Q_i is a *component implementation process* and P_i is the corresponding *component specification process*. $F_{impl}(Q_1, Q_2, \ldots, Q_n)$ then gives an *implementation network* and $F_{spec}(P_1, P_2, \ldots, P_n)$ the corresponding *specification network*.[7] Intuitively, if the implementation network may

[6]Recall that syntactic operators take processes as arguments and semantic operators take sets of behaviours or pairs of sets of behaviours as arguments.

[7]Note that we will use the generic term ***implementation process*** to refer to any $Q \in Imp(F_{impl}(Q_1, Q_2, \ldots, Q_n))$. (*Imp* is defined in definition 2.3 in section 2.7.) What it means to constitute an implementation process will be made more formal in section 3.2.

$$\begin{array}{lcl}
\lambda(V \setminus A) & \triangleq & V\ \lambda(\setminus A). \\
\lambda(V_1 \parallel_Y V_2) & \triangleq & V_1\ \lambda(\parallel_Y)\ V_2. \\
\lambda(Con \setminus A) & \triangleq & \lambda(Con)\ \lambda(\setminus A). \\
\lambda(Con \parallel_Y Con) & \triangleq & \lambda(Con)\ \lambda(\parallel_Y)\ \lambda(Con). \\
\lambda(Con \parallel_Y V) & \triangleq & \lambda(Con)\ \lambda(\parallel_Y)\ V. \\
\lambda(V \parallel_Y Con) & \triangleq & V\ \lambda(\parallel_Y)\ \lambda(Con).
\end{array}$$

Figure 3.1: Defining λ over contexts, where V, V_1, V_2 are process variables and $A, Y \subseteq \Sigma$

engage only in finally visible events then any external behaviour decomposition and/or relaxation of atomicity which was used in deriving Q_i from P_i, where $1 \le i \le n$, has been hidden. This means that the implementation and specification networks may be related using standard CSP refinement.

In view of condition 1, the main problem in defining a notion of refinement-after-hiding in practice lies not in giving a general definition of λ which has the required properties.[8] Rather, it is to define equations on λ such that the property of condition 1 is met *and* our refinement-after-hiding relation is as *large as possible.* This is another significant factor leading us to *derive* a notion of refinement-after-hiding rather than building from the bottom up a set of conditions which happen to imply condition 1.

We therefore need an appropriate high-level approximation of condition 1 which is as weak as possible and from which such a set of equations may be derived. The conditions RAH1-3 given in figure 3.2 fulfil this role: by theorem 3.1 below they are sufficient to imply condition 1, while imposing few restrictions on λ and so on any notion of refinement-after-hiding based thereon.

Theorem 3.1. *If conditions* RAH1-3 *hold of* λ*, then condition 1 also holds.*

Proof. Let $Q_1, \ldots, Q_n$ and $P_1, \ldots, P_n$ be processes; let F_{impl} and F_{spec} be contexts each containing n process variables such that $\lambda(F_{impl}) = F_{spec}$. We assume $Q_i \sqsupseteq^{\lambda}_{X} P_i$ for $1 \le i \le n$ — i.e. $\lambda([\![Q_i]\!]_X)$ is defined and $\lambda([\![Q_i]\!]_X) \subseteq$

[8]For example, the identity mapping would suffice here but would not give us any extra power: in fact, in this case, our notion of refinement-after-hiding would be equivalent to standard CSP refinement.

RAH1 If $\alpha Q \subseteq Fvis$ then $\lambda([\![Q]\!]_X)$ is defined and $\lambda([Q]_X) = [Q]_X$.

RAH2 If $\lambda([\![Q]\!]_X)$ is defined and $\lambda(\backslash A) = \backslash B$, then $\lambda([Q \setminus A]_X)$ is defined and $\lambda([\![Q \setminus A]\!]_X) = \lambda([\![Q]\!]_X) \setminus B$.

RAH3 If $\lambda([\![P]\!]_X)$, $\lambda([\![Q]\!]_X)$ are defined and $\lambda(\|_Y) = \|_Z$ then:
- $\lambda([\![P \|_Y Q]\!]_X)$ is defined.
- $\lambda([\![P \|_Y Q]\!]_X) = \lambda([\![P]\!]_X) \|_Z \lambda([\![Q]\!]_X)$

Figure 3.2: Conditions from which the theory will be derived, where P, Q are implementation processes, $X \in \{T, SF, FD\}$ and $A, Y \subseteq \Sigma$

$[\![P_i]\!]_X$ — and $\alpha F_{impl}(Q_1, Q_2, \ldots, Q_n) \subseteq Fvis$. By induction on the number of operators in F_{impl} using conditions RAH2 and RAH3 and also the information in figure 3.1, we have

$$\lambda([\![F_{impl}(Q_1, Q_2, \ldots, Q_n)]\!]_X) = F_{spec}(\lambda([\![Q_1]\!]_X), \lambda([\![Q_2]\!]_X), \ldots, \lambda([\![Q_n]\!]_X)).$$

By inductive application of proposition 2.8,

$$F_{spec}(\lambda([\![Q_1]\!]_X), \lambda([\![Q_2]\!]_X), \ldots, \lambda([\![Q_n]\!]_X)) \subseteq F_{spec}([\![P_1]\!]_X, [\![P_2]\!]_X, \ldots, [\![P_n]\!]_X).$$

By inductive application of proposition 2.1,

$$F_{spec}([\![P_1]\!]_X, [\![P_2]\!]_X, \ldots, [\![P_n]\!]_X) = [\![F_{spec}(P_1, P_2, \ldots, P_n)]\!]_X.$$

Hence, by RAH1,

$$[\![F_{impl}(Q_1, Q_2, \ldots, Q_n)]\!]_X \subseteq [\![F_{spec}(P_1, P_2, \ldots, P_n)]\!]_X.$$

□

Comments on RAH1-3

The roles of the conditions RAH1-3 are made clear by their use in the proof of theorem 3.1 and this proof also makes clear the role of the finally visible events from *Fvis*. That RAH1-3 are stated in terms of equalities — for example, we have $\lambda([\![Q \backslash A]\!]_X) = \lambda([\![Q]\!]_X) \backslash B$ rather than $\lambda([\![Q \backslash A]\!]_X) \subseteq \lambda([\![Q]\!]_X) \backslash B$ — is crucial as we derive the theory which is presented in this chapter. However, the detailed conditions which form this theory are only sufficient in general to imply versions of RAH2 and RAH3 with $\subseteq$ substituted for $=$. Were they to imply the original versions of these conditions, they would place restrictions on λ which would make it difficult to use in practice. This is why condition 1 uses containment and refinement rather than equality and equivalence. These issues are discussed at greater length at the appropriate points below.

3.1.1 Applying λ in the traces model

In general, we define the effect of applying λ to a particular process denotation in terms of applying it to the individual behaviours which constitute that denotation. Since it is needed in the next section, we give here the necessary definition for the traces model.

Definition 3.3. *Let $\mathcal{T}$ be a set of traces and u a trace. If $\lambda(u)$ is defined, then $\lambda(u)$ returns a trace. Moreover:*

1. *$\lambda(\mathcal{T})$ is defined if and only if $\lambda(t)$ is defined for every $t \in \mathcal{T}$.*
2. *If $\lambda(\mathcal{T})$ is defined, $\lambda(\mathcal{T}) \triangleq \{\lambda(t) \mid t \in \mathcal{T}\}$.*

We shall also require the monotonicity of λ defined over traces and the fact that the domain of λ over traces is prefix-closed:

TR-MONO	If $\lambda(t)$ and $\lambda(u)$ are defined and $t \leq u$, then $\lambda(t) \leq \lambda(u)$.
PREF-CLOS	If $\lambda(u)$ is defined and $t \leq u$, then $\lambda(t)$ is defined.

The fact of monotonicity means that receiving more information regarding a particular implementation trace cannot reduce our knowledge about the corresponding specification trace. Moreover, condition PREF-CLOS is a natural requirement once we have assumed monotonicity. In the proofs of results from this chapter, we sometimes have traces t, u such that $\lambda(u)$ is defined, $t \leq u$ and we wish to show that $\lambda(t) \leq \lambda(u)$. In order to do this, it is necessary to appeal to *both* TR-MONO and PREF-CLOS. However, in practice, we usually appeal explicitly only to TR-MONO and assume it is understood that PREF-CLOS is also appealed to.

3.2 Sets used in the theory

This section introduces a number of different sets which will be used in the derivation of the theory in this chapter, along with certain restrictions to be placed on implementation networks if they are to be verified using refinement-after-hiding. One of the most important of these sets is *AllSet*, a set of sets of events: we shall require that all alphabets of implementation processes used in this chapter are taken from *AllSet* and that any hiding operator used to define an implementation context, F_{impl}, must be parameterized by a set from *AllSet*; we shall also impose a restriction on the parallel composition to be used in building an implementation network from a set of component processes, so that the composition always synchronizes on a set from *AllSet*.

Further details on *AllSet* and on these restrictions are given in the remainder of this section.

Before proceeding to the derivation of the theory proper, therefore, we fix some notions regarding the nature of the sets we shall use. There are six main sets which we shall need to deal with, two of which have already been introduced. The sixth of these sets is introduced after the other five. (In general, these should be viewed as *meta sets*, representing all possible such sets with which we might work in practice.)

- Σ_{impl}, denoting the set of events in which implementation processes may engage.
- Σ_{spec}, denoting the set of (specification) events which may be "engaged" in by sets of behaviours produced by applying λ to (the denotation of) an implementation process.
- *Fvis*, denoting the set of finally visible events.
- *AllSet*, containing all possible sets with which we may parameterize the hiding and parallel composition operators used to construct implementation networks. That we will restrict the sets with which these operators may be parameterized reflects the approach of [12].
- *BTrace*, a non-empty *finite*[9] set of implementation traces such that $\lambda(t)$ is defined for every $t \in BTrace$.

Intuitively, *BTrace* contains only implementation traces which may be regarded as "atoms" or as indivisible in some sense: that is, implementation traces which it does not make sense to decompose further into sub-traces.[10] More specifically, we assume that each specification action in Σ_{spec} may be implemented by a (finite) number of implementation traces and that *BTrace* consists of exactly those traces. For example, *send*.0 from the running example may be implemented by both $\langle data.0, ack.yes\rangle$ and $\langle data.0, ack.no, data.0\rangle$ and both traces would be members of *BTrace*. It does not make sense to decompose further into sub-traces these particular implementation traces since it is not possible to decompose the high-level action which they are being used to implement.

[9]Although the proofs of certain results in this chapter as they are presented at the moment require the property of finiteness, it has been realised that they may be presented in such a way that this property is *not* needed. In any case, the property of finiteness is not needed at all as we prove the sufficiency as a notion of refinement-after-hiding of the conditions which are derived in this chapter.

[10]Note, however, that we do not assume these traces to be "atomic" in the sense that they will *execute* as indivisible entities: i.e. it is not the case that, when one "atom" is executing, no others may be executing. In practice, they may be interleaved with each other and with other traces not contained in *BTrace*.

3.2.1 Σ_{impl}, Σ_{spec} and *BTrace*

For any implementation process Q, the possibility should exist that $\lambda(\tau Q)$ is defined. In view of this and definition 3.3, Σ_{impl} and Σ_{spec} may be characterised more formally as follows. (That $Fvis \subseteq \Sigma_{impl}$ in the following definition reflects the intuition that implementation processes should be free to engage in any of the finally visible events.)

Definition 3.4. *We assume that* $Fvis \subseteq \Sigma_{impl} \subseteq \Sigma$, $Fvis \neq \varnothing$ *and* $\Sigma_{spec} \subseteq \Sigma$. *Moreover:*

1. $\Sigma_{impl} \triangleq \bigcup\{events(t) \mid \lambda(t) \text{ is defined}\} \triangleq \bigcup\{events(t) \mid t \in BTrace\}$.
2. $\Sigma_{spec} \triangleq \bigcup\{events(\lambda(t)) \mid \lambda(t) \text{ is defined}\} \triangleq \bigcup\{events(\lambda(t)) \mid t \in BTrace\}$.

Conditions are also imposed on *BTrace* as part of this definition: namely that the traces it contains cover exactly the events from Σ_{impl} and, after the application of λ, exactly the events from Σ_{spec}. We note that this definition is consistent with the intuition given above with respect to the members of *BTrace*. With respect to part 2 of the definition, we assume that, for every $a \in \Sigma_{spec}$, there exists at least one trace, t, such that $\lambda(t) = \langle a \rangle$: i.e. such that t implements a. By the intuition given above, *BTrace* would consist of all such t.[11] With respect to part 1 of the definition, we assume that all traces which an implementation process may execute can be built in some way from the "atoms" in *BTrace* and so the events of these "atoms" will give exactly the events in Σ_{impl}. However, it must also be noted that definition 3.4 gives the only formal statement we have of the properties of *BTrace* — other than the statement in the previous subsection that it is finite and that $\lambda(t)$ is defined for every $t \in BTrace$ — and so the intuition that it contains only "atoms" is not recorded formally. In other words, this intuition plays no role in this chapter in the derivation of conditions sufficient to define a notion of refinement-after-hiding, nor does it appear in those conditions themselves. In fact, it is used solely to justify (informally) the restrictions on hiding and parallel composition which have been mentioned above and which are imposed in section 3.2.6. Further consideration of the nature of *BTrace* is given in section 3.7, once the theory has been presented in its entirety.

3.2.2 Considering *AllSet*

We now consider *AllSet* and its definition. In doing so, *MinSet* is introduced, the sixth of the sets with which we will have to deal: Σ_{impl} may be partitioned

[11] Since we assumed above that each $a \in \Sigma_{spec}$ is implemented by a finite number of traces and since Σ_{spec} is finite, then *BTrace* constructed in this way would also be finite.

using the traces of *BTrace* as a basis and *MinSet* is defined as the set of sets comprising this partition.

Definition 3.5. *MinSet is a partition of* Σ_{impl} *such that, if* $t, u \in BTrace$ *and* $A, B \in MinSet$ *where* $A \neq B$*:*

1. *If* $events(t) \cap A \neq \varnothing$*, then* $events(t) \subseteq A$*.*
2. *If* $events(t) \subseteq A$ *and* $events(u) \subseteq B$*,* $events(\lambda(t)) \cap events(\lambda(u)) = \varnothing$*.*

The sets from *MinSet* essentially group together (the events from) the traces from *BTrace* into disjoint sets, where the events from any particular trace in *BTrace* are contained in only one of the sets. Moreover, the sets in *MinSet* also enjoy a property of disjointness *after* applying λ to the traces from *BTrace* whose events they contain. Each set in *AllSet* is then defined as the union of a number of sets from *MinSet* (although *AllSet* also contains the empty set).

Definition 3.6. $AllSet \triangleq \{\bigcup \mathcal{X} \mid \mathcal{X} \in \mathbb{P}(MinSet)\}$.

By virtue of definitions 3.4 and 3.5, the events of each "atom" in *BTrace* are totally contained in exactly one set from *MinSet*.[12] Definition 3.6 then means that, for each $t \in BTrace$ and $A \in AllSet$, either $events(t) \subseteq A$ or $events(t) \cap A = \varnothing$. Due to the restrictions which are imposed on implementation processes and networks in section 3.2.6, the hiding and parallel composition operators used to build implementation networks from component implementation processes may be parameterized only with sets from *AllSet*, which means that those operators will regard the traces from *BTrace* as indivisible entities. That is, either all events from such a trace will be hidden or none will; either we require synchronization in parallel on all the events of such a trace or on none of them. In general, *MinSet* defines the smallest sets which may be hidden or on which we may synchronize in parallel while still regarding the traces from *BTrace* as indivisible.[13]

In view of definition 3.6, $A = \bigcup_{i \in I} A_i$ for any set $A \in AllSet$, where I is an indexing set into *MinSet* and so $A_i \in MinSet$ for every $i \in I$. In the event that $A = \varnothing$, then $I = \varnothing$. Moreover, $MinSet \subseteq AllSet$ and so $A \in AllSet$ for any $A \in MinSet$. The following notation, used to define the smallest set

[12]In terms of the running example, $\{data.0, data.1, ack.yes, ack.no\}$ could constitute a set from *MinSet*: this set contains all events from the atoms $\langle data.0, ack.yes\rangle$, $\langle data.1, ack.yes\rangle$, $\langle data.0, ack.no, data.0\rangle$ and $\langle data.1, ack.no, data.1\rangle$.

[13]We have considered here only the consequences of the restriction imposed by definition 3.5(1). That we ignore definition 3.5(2) is simply because it may actually be derived as part of the theory (see section 3.7). It appears as a definition rather than as a derived theorem since the fact that it could be derived was realised only on a final revision of this thesis.

from *AllSet* in which another specified set is contained, will also prove useful. (In respect of this, note that $\Sigma_{impl} \in \mathit{AllSet}$ by definitions 3.5 and 3.6.)

Definition 3.7. *For $X \subseteq \Sigma_{impl}$, $[[X]]$ denotes the smallest set $A \in \mathit{AllSet}$ such that $X \subseteq A$.*

When the notation $[[X]]$ is used in what follows, we will only show explicitly that $X \subseteq \Sigma_{impl}$ if it is not clear from the context that this is the case. In respect of this, we observe that the following hold by definition 3.4.

- $\mathit{events}(s) \subseteq \Sigma_{impl}$ for $s \in \mathit{BTrace}$.
- $\mathit{events}(t) \subseteq \Sigma_{impl}$ for any trace t such that $\mathit{events}(t) \subseteq \mathit{Fvis}$.
- $\mathit{events}(u) \subseteq \Sigma_{impl}$ for any trace u such that $\lambda(u)$ is defined.

We also define the application of λ to sets from *AllSet*. This will prove to be useful when considering the effect of applying λ to operators.

Definition 3.8. $\lambda(A) \triangleq \bigcup\{\mathit{events}(\lambda(t)) \mid t \in \mathit{BTrace} \;\wedge\; \mathit{events}(t) \subseteq A)\}$ *for $A \in \mathit{AllSet}$.*

3.2.3 Implementation processes and process alphabets

The following definition is used to characterise implementation processes in terms of Σ_{impl}.

Definition 3.9. *Let Q be a process. Q is an implementation process if and only if $\beta(Q) \subseteq \Sigma_{impl}$.*

In view of this, the following result is immediate.

Proposition 3.2. *Let P, Q be implementation processes and $A, Y \subseteq \Sigma$. Then:*

1. *$P \setminus A$ is an implementation process.*
2. *$P \parallel_Y Q$ is an implementation process.*

Proposition 3.2 shows that, for an implementation context F_{impl} and component implementation processes $Q_1, \ldots, Q_n$, $Q \in \mathit{Imp}(F_{impl}(Q_1, Q_2, \ldots, Q_n))$ is also an implementation process. (Recall that the notation *Imp* is defined in definition 2.3 in section 2.7.) The alphabet of any implementation process is then defined as follows.

Definition 3.10. *Let Q be an implementation process. Then $\alpha Q \triangleq [[\beta(Q)]]$ and so $\alpha Q \in AllSet$.*[14]

Since $\beta(Q) \subseteq \Sigma_{impl}$ for any implementation process Q, then αQ is defined for any such process. In the proofs of some of the results in this chapter, it is necessary to construct, using the definitions in section 2.10, processes Q' which have a certain pre-defined semantics. All of these processes are such that $\beta(Q') \subseteq \Sigma_{impl}$ and it will be clear from the relevant definitions that this is the case. We will therefore regard them as implementation processes without further comment and consider that $\alpha Q'$ is defined for any such Q'. Indeed, as a general rule, definition 3.9 and proposition 3.2 regarding implementation processes will only be appealed to implicitly in proofs of results from the remainder of this chapter.

The following result shows that the alphabets of implementation processes relate to those of their components in the way that we would expect in view of the detail in figure 2.5, provided that the hiding and parallel composition operators used are parameterized with sets from *AllSet*. This latter restriction will be imposed in general in section 3.2.6.

Proposition 3.3. *Let P, Q be implementation processes and $A, Y \in AllSet$.*

1. $\alpha(P \setminus A) = (\alpha P) - A$.
2. $\alpha(P \parallel_Y Q) = \alpha P \cup \alpha Q$.

It is possible that we may have an implementation process Q where $\beta(Q) \not\subseteq \Sigma_{impl}$, even though $\lambda([\![Q]\!]_X)$ is defined. This can arise due to the use of parallel composition: for example, $\beta(P \parallel_Y P') = \beta(P) \cup \beta(P')$, while it is possible that $(P \parallel_Y P')$ never engages in any events from $\beta(P')$ due to the nature of P and the choice of Y. In such a case, in lieu of Q, we would assume the implementation process to be $Q \setminus Z$, where $Z = \Sigma - (\Sigma_{impl} \cap \beta(Q))$. All events in which Q may engage are contained in Σ_{impl} since $\lambda([\![Q]\!]_X)$ is defined (this will turn out to hold in general); moreover, all events in which it may engage are also contained in $\beta(Q)$. As a result, $[\![Q]\!]_X = [\![Q \setminus Z]\!]_X$, while $\beta(Q \setminus Z) \subseteq \Sigma_{impl}$.

3.2.4 Basic results regarding *MinSet*, *AllSet* and λ

We are now able to give some basic results characterising the sets from *MinSet* and *AllSet* and the application to them of λ. The first reflects the

[14]That αQ for any implementation process Q is taken from *AllSet* reflects the approach of [12]. Taken with restriction R2 from section 3.2.6 below, it means that any parallel composition operator used to build an implementation network from component processes must be parameterized by a set from *AllSet* (see proposition 3.11, also in section 3.2.6).

fact that each set in *AllSet* is constructed from a number of sets in *MinSet* and distinct sets in *MinSet* are disjoint.

Proposition 3.4. *Let $A \in MinSet$ and $B \in AllSet$ be such that $A \cap B \neq \varnothing$. Then $A \subseteq B$.*

The following result shows a useful way of characterising any set from *AllSet*.

Proposition 3.5. $A = \bigcup\{events(t) \mid t \in BTrace \wedge events(t) \subseteq A\}$ *for* $A \in AllSet$.

The next result shows that λ distributes across the set union operator when the latter is used to compose sets from *MinSet*.

Proposition 3.6. $\lambda(\bigcup_{i\in I} A_i) = \bigcup_{i\in I} \lambda(A_i)$, *where I is an indexing set into MinSet.*

The following result shows that if we apply λ to disjoint sets then the results of that application are also disjoint.

Proposition 3.7. *If $A \cap A' = \varnothing$ for $A, A' \in AllSet$ then $\lambda(A) \cap \lambda(A') = \varnothing$.*

The next result shows that *AllSet* is closed under the application of the set operators of subtraction, union and intersection; moreover, λ distributes across the same operators when they are applied to sets from *AllSet*.

Proposition 3.8. *Let $A, B \in AllSet$ and $\oplus \in \{-, \cup, \cap\}$. Then:*

1. $A \oplus B \in AllSet$.
2. $\lambda(A \oplus B) = \lambda(A) \oplus \lambda(B)$.

This final result further characterises the relationship between Σ_{impl}, Σ_{spec} and *AllSet*.

Proposition 3.9. *The following hold:*

1. $\Sigma_{impl} \in AllSet$.
2. $\Sigma_{spec} = \bigcup_{A\in MinSet} \lambda(A)$.
3. *For every $A \in AllSet$, $A \subseteq \Sigma_{impl}$ and $\lambda(A) \subseteq \Sigma_{spec}$.*

3.2.5 Using $[[X]]$ for $X \subseteq \Sigma_{impl}$

The following result concerns the $[[X]]$ notation, used to return the smallest set $A \in \mathit{AllSet}$ such that $X \subseteq A$. It will usually be appealed to implicitly whenever it is needed.

Proposition 3.10. *Let $A \in \mathit{AllSet}$ and $R, S, X \subseteq \Sigma_{impl}$.*

1. *If $X \subseteq A$ then $[[X]] \subseteq A$.*
2. $[[R \cup S]] = [[R]] \cup [[S]]$.

3.2.6 Restrictions

In the remainder of this chapter, we impose the following restrictions on all implementation networks $F_{impl}(Q_1, \ldots, Q_n)$, where $Q_1, \ldots, Q_n$ are component implementation processes. (Recall that the notation *Imp* is defined in definition 2.3 in section 2.7.)

R1 Let $\backslash A$ be used in the definition of F_{impl}. Then $A \in \mathit{AllSet}$.

R2 Let $(P \parallel_Y Q) \in \mathit{Imp}(F_{impl}(Q_1, \ldots, Q_n))$. Then $Y = \alpha P \cap \alpha Q$.

These are essentially the restrictions imposed in [12], although they are not stated explicitly as such in that paper; similar restrictions are also imposed in [59], which presents an implementation relation that is effectively a notion of refinement-after-hiding.[15] Condition R2 enforces the requirement that the parallel composition operator used to build any implementation network is that given by Tony Hoare in [31] and used in [12]. The following important result follows from condition R2.

Proposition 3.11. *Let $Q_1, \ldots, Q_n$ be component implementation processes and $F_{impl}(Q_1, \ldots, Q_n)$ an implementation network. Let $\parallel_Y$ be such that it is used in the definition of F_{impl}. Then $Y \in \mathit{AllSet}$.*

Condition R1 and the result from proposition 3.11 are essential for the derivation of the theory presented in this chapter: their most immediate practical effect is that we are able to characterise exactly the result of applying λ to the hiding and parallel composition operators respectively (see theorems 3.16 and 3.17 in section 3.3). This characterisation then has far-reaching implications in the remainder of the chapter (this issue is discussed further

[15]Note that R1 and R2 are restrictions only on the operators which are used to build implementation networks from component implementation processes. They do *not* restrict the nature of the operators which may be used to construct the component implementation processes themselves.

in section 3.7). Moreover, the restrictions imposed by R1 and proposition 3.11 make sense in practice, as has been discussed above: since the traces in *BTrace* are to be regarded as atoms, then they should be treated as such by hiding and parallel composition. (Recall that by definitions 3.5 and 3.6, for any $t \in BTrace$ and $A \in AllSet$, either $events(t) \subseteq A$ or $events(t) \cap A = \varnothing$.)

For example, consider the case that $t \in BTrace$ is used to implement the specification event $a \in \Sigma_{spec}$. When hiding behaviours in any specification process in which a appears, it is only possible to hide all of a or to leave it visible. Thus, it does not make sense to be able to hide a non-empty subset of the events of t, as this could leave a partial implementation for a still-visible a or a "dangling" partial implementation which no longer has a corresponding specification event. Similarly, when composing specification processes in parallel, it is only possible to either synchronize on all of a or not synchronize on it at all. If we are able to synchronize on a non-empty subset of the events of t, then we may end up with an implementation trace which is neither a single execution of t nor two interleaved executions of t: it is not clear how this relates to what is possible for a in the specification (i.e. a single occurrence of a if we have to synchronize on it or two occurrences of a if we do not have to synchronize on it). For example, the trace $\langle data.0, ack.yes\rangle$ is used to implement $send.0$ in the running example. If we were to able to compose two instances of $\langle data.0, ack.yes\rangle$ in parallel, synchronizing only on $\{data.0\}$, then the resulting trace would be $\langle data.0, ack.yes, ack.yes\rangle$. It would not make much sense to view this as an implementation of $\langle send.0, send.0\rangle$ — i.e. the composition in parallel of two instances of $\langle send.0\rangle$ when we do not synchronize on $\{send.0\}$ — but nor does it make sense to regard it as an implementation of $\langle send.0\rangle$, which would arise if we did have to synchronize on $\{send.0\}$.

We choose to impose condition R2 and then derive proposition 3.11 rather than simply imposing directly the statement from the proposition because R2 is useful in its own right. In particular, the semantic definition of parallel composition in the stable failures model without the assumption of R2 is difficult to work with; by virtue of R2 the definition becomes much more tractable (see theorem 2.20 in chapter 2). In any case, R2 simply means we work with the parallel composition operator which is defined in Hoare's book on CSP ([31]).

In the statement of the conditions RAH1-3 in figure 3.2, the only restriction placed on the sets A and Y used there is that they should be subsets of Σ. In view of R1, we impose the further restriction that $A \in AllSet$ for the set A used in the statement of RAH2; in view of R2, we impose the restriction that $Y = \alpha P \cap \alpha Q$ in the statement of RAH3. We do this because we wish RAH1-3 to be as weak as possible. These additional restrictions will be reflected in the rendering of RAH1-3 as necessary in each of the three semantic models.

3.2.7 Finally visible events

We impose one extra condition on *AllSet*, in relation to finally visible events. This reflects the intuition that, given an implementation process, it should be possible to hide *exactly* the finally *invisible* events, leaving visible exactly those events from *Fvis* in which the process may engage.

HIDE-INVIS Let Q be an implementation process. If $\lambda(\llbracket Q \rrbracket_T)$ is defined, there exists $A \in \mathit{AllSet}$ such that $\alpha Q - A = \alpha Q \cap \mathit{Fvis}$.

In view of this condition, we are able to derive the following result, namely that *Fvis* is a set in *AllSet*.

Proposition 3.12. *Fvis* $\in$ *AllSet*.

3.2.8 Summary

In the following sections, we present the derivation of the theory proper in all three semantic models. In particular, the conditions RAH1-3 are rendered in each of the semantic models and conditions are then derived which refer to the effect of λ defined over individual behaviours rather than over process denotations as a whole. Before proceeding, we recall the sets which have been introduced in this section.

- Σ_{impl} denotes the set of events in which implementation processes may engage.
- Σ_{spec} denotes the set of (specification) events which may be "engaged" in by sets of behaviours produced by applying λ to (the denotation of) an implementation process.
- *Fvis* denotes the set of finally visible events.
- *BTrace* is a set of traces, each of which may be regarded as an "atom" and so as an indivisible entity which may not be decomposed further into sub-traces.
- *MinSet* is a set of sets of events. It partitions the events of Σ_{impl} in such a way that, for each trace in *BTrace*, the events of that trace are fully contained in one of the sets of the partition.

- Each set in *AllSet* is the union of a number of sets from *MinSet* and, in building implementation networks, we may only hide or compose in parallel on members of this set. (Recall, though, that *AllSet* also contains the empty set.) This essentially means that any set to be hidden or synchronized on during parallel composition as we build implementation networks regards traces from *BTrace* as indivisible.

3.3 RAH1-3 in the traces model and applying λ to operators

As indicated above, we shall define our notion of refinement-after-hiding in terms of the effect of applying λ to *individual behaviours.* In order to move towards that goal in the traces model, in this section we derive counterparts to RAH1 and RAH2 which refer to individual traces in place of process denotations. (It turns out that a counterpart to RAH3 is not needed.) These are used in the next section to derive the final conditions defining refinement-after-hiding in the traces model. In addition, we prove results which show how to define exactly the result of applying λ to the hiding and parallel composition operators respectively.

As a first step, we render the conditions RAH1-3 in the traces model (recall that the original statement of the conditions in figure 3.2 was parameterized by the variable X to denote one of the three semantic models; here we substitute T for X). We also impose here the restrictions on hiding and parallel composition that were introduced in section 3.2.6. This means that any set to be hidden (from RAH2) will be taken from *AllSet* and any two processes composed in parallel (in RAH3) must synchronize on the events in the intersection of their respective alphabets. The new conditions, denoted TI1, TI2 and TI3, are given in figure 3.3. Using TI1 and TI2, we are able to derive conditions RAH1-T and RAH2-T below which refer to the application of λ to individual traces rather than to process denotations as a whole. There is no equivalent condition given in relation to parallel composition and derived from TI3, since we do not need such a condition in any of the proofs which follow. This is not to say, however, that TI3 is redundant, since it is used in the proof of theorem 3.17 below and in the proof of a result from the next section.

Theorem 3.13 (RAH1-T). *Let t be a trace such that $events(t) \subseteq Fvis$. Then $\lambda(t)$ is defined and $\lambda(t) = t$.*

Theorem 3.14. *$\lambda(\langle\rangle)$ is defined and $\lambda(\langle\rangle) = \langle\rangle$.*

Proof. Since $events(\langle\rangle) = \varnothing$, $events(\langle\rangle) \subseteq Fvis$. The proof follows by RAH1-T. □

TI1	If $\alpha Q \subseteq Fvis$, then $\lambda([\![Q]\!]_T)$ is defined and $\lambda([\![Q]\!]_T) = [\![Q]\!]_T$.
TI2	If $\lambda([\![Q]\!]_T)$ is defined, $A \in AllSet$ and $\lambda(\backslash A) = \backslash B$, then $\lambda([\![Q \setminus A]\!]_T)$ is defined and $\lambda([\![Q \setminus A]\!]_T) = \lambda([\![Q]\!]_T) \setminus B$.
TI3	If $\lambda([\![P]\!]_T)$, $\lambda([\![Q]\!]_T)$ are defined, $Y = \alpha P \cap \alpha Q$ and $\lambda(\|_Y) = \|_Z$, then: – $\lambda([\![P \|_Y Q]\!]_T)$ is defined. – $\lambda([\![P \|_Y Q]\!]_T) = \lambda([\![P]\!]_T) \|_Z \lambda([\![Q]\!]_T)$.

Figure 3.3: Rendering RAH1-3 in the traces model, where P and Q are implementation processes

Theorem 3.15 (RAH2-T). *Let t be a trace and $A \in AllSet$, where $\lambda(\backslash A) = \backslash B$. If $\lambda(t)$ is defined, then:*

- *$\lambda(t \setminus A)$ is defined.*
- *$\lambda(t \setminus A) = \lambda(t) \setminus B$.*

3.3.1 Applying λ to operators

We now show how the application of λ to operators is defined. Before proceeding, we first require a condition to reflect the intuition that mapped-to sets of behaviours may only engage in events from Σ_{spec} and so mapped-to operators should only be parameterised by subsets of Σ_{spec}.

Definition 3.11. *By definition:*

1. *Let $\lambda(\backslash A) = \backslash B$. Then $B \subseteq \Sigma_{spec}$.*
2. *Let $\lambda(\|_Y) = \|_Z$. Then $Z \subseteq \Sigma_{spec}$.*

The following results play a crucial role in the remainder of this chapter and also in general. For they tell us that, provided we use in practice only sets which have the properties of those from *AllSet*, we have no discretion in defining the effect of λ applied to the operators we use.

Theorem 3.16. *Let $A \in AllSet$. Then $\lambda(\backslash A) = \backslash\lambda(A)$.*

Theorem 3.17. *Let $Y \in AllSet$. Then $\lambda(\|_Y) = \|_{\lambda(Y)}$.*

When we appeal to RAH2-T in what follows, we shall implicitly use theorem 3.16 as well: that is, we shall use $\lambda(A)$ in place of B from RAH2-T.

3.4 Sufficient conditions in the traces model

In this section, we present the sufficient conditions which must be met by our mapping if it is to function as a basis for a notion of refinement-after-hiding in the traces model. We will need the following additional results, which concern the application of λ to traces.

Theorem 3.18. *Let $A \in MinSet$ be such that $A \subseteq Fvis$. Let t be a trace such that $events(t) \subseteq A$. Then $\lambda(t)$ is defined and $\lambda(t) = t$.*

Proof. The proof is immediate by RAH1-T. □

Theorem 3.19. *Let $t \circ \langle a \rangle$ be a trace such that $a \in A \in MinSet$.*

1. *$\lambda(t)$ and $\lambda((t \circ \langle a \rangle)\lceil A)$ are defined if and only if $\lambda(t \circ \langle a \rangle)$ is defined.*
2. *If $\lambda(t \circ \langle a \rangle)$ is defined then $\lambda(t \circ \langle a \rangle) = \lambda(t) \circ r$, where the trace r is such that:*
   (a) *$\lambda((t \circ \langle a \rangle)\lceil A) = \lambda(t\lceil A) \circ r$.*
   (b) *$events(r) \subseteq \lambda(A)$.*

Figure 3.4 then presents those conditions which are sufficient to define a notion of refinement-after-hiding in the traces model. Note the compositional nature of the definition of λ — encapsulated in conditions S5 and Ts4 — which means that it need only be defined directly over sets from *MinSet* and over traces t such that $events(t) \subseteq A \in MinSet$.[16] One of the main roles played by the sets in *MinSet* is illustrated by condition Ts4. By Ts4, we define $\lambda(t \circ \langle a \rangle)$, where $a \in A \in MinSet$, in terms of $\lambda(t)$ and $\lambda((t \circ \langle a \rangle)\lceil A)$. In a sense, $\lambda((t \circ \langle a \rangle)\lceil A)$ is a function from a trace, $t\lceil A$, and an event, a, to the trace extension r which is used in the statement of Ts4. This means that A tells us what we need to know of t in order to determine the additional information which is given to us by the occurrence of a after t.

A comment is also required with regard to condition S1(d) and what it means to constitute an implementation process. In order to simplify the proofs of results in the remainder of this chapter, S1(d) is only appealed to implicitly whenever it is needed. In particular, for any implementation process Q we will assume without further comment that $\beta(Q) \subseteq \Sigma_{impl}$, meaning that $[[\beta(Q)]]$ and so αQ are defined. Moreover, on the basis of proposition 3.2, it is implicit that any new process constructed from an implementation process or processes using hiding and parallel composition will also be an implementation process.

[16]Examples given in chapter 4 show how we may then define in practice the result of applying the mapping to such a trace t.

S1 (a) *BTrace* is a non-empty set of traces such that $\lambda(t)$ is defined for every $t \in \mathit{BTrace}$.

(b) Σ_{impl} and Σ_{spec} are as defined in definition 3.4.

(c) *MinSet* and *AllSet* are as defined in definitions 3.5 and 3.6.

(d) Q is an implementation process if and only if $\beta(Q) \subseteq \Sigma_{impl}$.

S2 If Q is an implementation process, then $\alpha Q \triangleq [[\beta(Q)]]$ and so $\alpha Q \in \mathit{AllSet}$.

S3 $\mathit{Fvis} \in \mathit{AllSet}$ and $\mathit{Fvis} \neq \varnothing$.

S4 For $A \in \mathit{MinSet}$,
$\lambda(A) \triangleq \bigcup\{\mathit{events}(\lambda(t)) \mid t \in \mathit{BTrace} \wedge \mathit{events}(t) \subseteq A)\}$.

S5 Let $A \in \mathit{AllSet}$ be such that $A = \bigcup_{i \in I} A_i$, where I is an indexing set into *MinSet*. Then $\lambda(A) = \bigcup_{i \in I} \lambda(A_i)$.

S6 If $A \in \mathit{AllSet}$, then $\lambda(\backslash A) = \backslash \lambda(A)$.

S7 If $Y \in \mathit{AllSet}$, then $\lambda(\|_Y) = \|_{\lambda(Y)}$.

(a) Conditions on sets.

TS1 Let $\mathcal{T}$ be a set of traces and u a trace. If $\lambda(u)$ is defined, then $\lambda(u)$ returns a trace. Moreover:

- $\lambda(\mathcal{T})$ is defined if and only if $\lambda(t)$ is defined for every $t \in \mathcal{T}$.
- If $\lambda(\mathcal{T})$ is defined, $\lambda(\mathcal{T}) \triangleq \{\lambda(t) \mid t \in \mathcal{T}\}$.

TS2 Let t be a trace, $A \in \mathit{MinSet}$ such that $\mathit{events}(t) \subseteq A \subseteq \mathit{Fvis}$. Then $\lambda(t)$ is defined and $\lambda(t) = t$.

TS3 Let $t \circ \langle a \rangle$ be a trace such that $a \in A \in \mathit{MinSet}$. Then $\lambda(t)$ and $\lambda((t \circ \langle a \rangle)\lceil A)$ are defined if and only if $\lambda(t \circ \langle a \rangle)$ is defined.

TS4 Let $t \circ \langle a \rangle$ be a trace such that $a \in A \in \mathit{MinSet}$. If $\lambda(t \circ \langle a \rangle)$ is defined then $\lambda(t \circ \langle a \rangle) = \lambda(t) \circ r$, where:

- $\lambda((t \circ \langle a \rangle)\lceil A) = \lambda(t\lceil A) \circ r$.
- $\mathit{events}(r) \subseteq \lambda(A)$.

(b) Conditions on traces

Figure 3.4: Sufficient conditions

That the conditions in figure 3.4 are sufficient to define a notion of refinement-after-hiding follows from theorem 3.20. In order to prove this result, we use only the conditions S1-7 and Ts1-4 and no other results or conditions which have already appeared in this chapter.[17] To emphasise this fact, some necessary supporting results which have already appeared are restated and reproved (in the appendix) using only S1-7 and Ts1-4. Recall also before we proceed that conditions R1 and R2 are imposed on any implementation network $F_{impl}(Q_1, Q_2, \ldots, Q_n)$.

Theorem 3.20. *Let F_{impl} and F_{spec} be implementation and specification contexts respectively, containing n process variables, such that $\lambda(F_{impl}) = F_{spec}$. Let $Q_1, \ldots Q_n$ be component implementation processes and $P_1, \ldots, P_n$ be processes. Assume that conditions S1-7 and TS1-4 from figure 3.4 all hold. If $Q_i \sqsupseteq_T^\lambda P_i$ for $1 \leq i \leq n$ and $\alpha F_{impl}(Q_1, Q_2, \ldots, Q_n) \subseteq Fvis$, then $F_{impl}(Q_1, Q_2, \ldots, Q_n) \sqsupseteq_T F_{spec}(P_1, P_2, \ldots, P_n)$.*

3.4.1 Additional comments

Proposition A.15 in the appendix is weaker than its counterpart Ti3, since the former uses set containment, $\subseteq$, in its statement while the latter uses equality, $=$. The reason for this is the form taken by proposition A.12, which effectively states that, where $\lambda(s)$ and $\lambda(u)$ are defined:

$$t \in s \parallel_Y u \Longrightarrow \lambda(t) \in \lambda(s) \parallel_Z \lambda(u).$$

In other words, we are only able to prove that $\lambda(s \parallel_Y u) \subseteq \lambda(s) \parallel_Z \lambda(u)$ rather than $\lambda(s \parallel_Y u) = \lambda(s) \parallel_Z \lambda(u)$. However, this limitation is a positive benefit since the latter result would place requirements on λ which are too restrictive to be of use in practice. Consider the case that λ is a mapping which makes behaviours more abstract. For traces s, u, it may be that $s \parallel_Y u = \varnothing$, while $\lambda(s) \parallel_Z \lambda(u) \neq \varnothing$ — i.e. $\lambda(s \parallel_Y u) \neq \lambda(s) \parallel_Z \lambda(u)$ — for an otherwise sensible mapping λ. For example, consider the traces $s = \langle data.0, ack.no, data.0 \rangle$ and $u = \langle data.0, ack.yes \rangle$ from the running example. If we attempt to compose in parallel on $Y = \alpha data \cup \alpha ack$, then $s \parallel_Y u = \varnothing$ and so $\lambda(s \parallel_Y u) = \varnothing$. However, we would expect that $\lambda(s) = \lambda(u) = \langle send.0 \rangle$ (data retransmission always succeeds) and so $\lambda(s) \parallel_Z \lambda(u)$ will be non-empty whatever the value of Z. It is also difficult to avoid the fact in general that $\lambda(v \parallel_Y w) \subset \lambda(v) \parallel_Z \lambda(w)$ for traces v, w when λ is being used to make behaviours more concrete.

[17]We do, however, assume that definitions 3.1, 3.2 and 3.7 still hold.

3.5 The stable failures model

In our consideration of refinement-after-hiding in the stable failures model, we assume that we work within the same framework of definitions and restrictions employed in the traces model. The only changes are the fact that we now have to define λ over process denotations in this new model and also have to render RAH1-3 in this model. This means that all definitions, conditions and restrictions which were stated in sections 3.1 and 3.2 are still in force here. As a result, we may appeal here to any and all results proved in those sections.

3.5.1 Applying λ to process denotations in the stable failures model

Consider an implementation process Q. In order to apply λ to $[\![Q]\!]_{SF}$ we effectively apply it separately to τQ and ϕQ. The necessary detail is given in the following definition.

Definition 3.12. *Let Q be an implementation process.*

1. *$\lambda([\![Q]\!]_{SF})$ is defined if and only if $\lambda(\tau Q)$ is defined and $\lambda(\phi Q)$ is defined.*
2. *If $\lambda([\![Q]\!]_{SF})$ is defined then $\lambda([\![Q]\!]_{SF}) \triangleq (\lambda(\tau Q), \lambda(\phi Q))$.*
3. *Let (t, R) be a failure. $\lambda(R, t)$ is defined if and only if $R \subseteq \Sigma_{impl}$ and $\lambda(t)$ is defined. If $\lambda(R, t)$ is defined, then $\lambda(R, t) \subseteq \Sigma$. Moreover:*
   - *(a) $\lambda(\phi Q)$ is defined if and only if $\lambda(R \cap \alpha Q, t)$ is defined for every $(t, R) \in \phi Q$.*
   - *(b) If $\lambda(\phi Q)$ is defined then:*
   $$\lambda(\phi Q) \triangleq \{(\lambda(t), X) \mid \begin{array}{l}(\exists (t, R) \in \phi Q)\ R \subseteq \alpha Q \wedge \\ X \subseteq \lambda(R, t) \cup (\Sigma - \lambda(\alpha Q))\}.\end{array}$$

We present the definition of λ here in terms of process denotations, rather than in terms of arbitrary sets of behaviours as is done in definition 3.3, simply because we need to use the syntactic notion of an alphabet. In order to compute $\lambda(\phi Q)$, we apply λ separately to the trace component and the refusal component respectively of each of the failures $(t, R) \in \phi Q$ such that $R \subseteq \alpha Q$. In actual fact, we apply λ to each *refusal/trace pair* (R, t), since what it means for a process to refuse a particular set of events may differ according to the trace after which they are refused. In terms of the running example, it would make no sense for *LeftImpl* to offer either *ack.yes* or *ack.no* after the trace $\langle in.0 \rangle$ — i.e. before it had attempted any communication along channel *data* — and so it should be perfectly acceptable for it to exhibit the failure $(\langle in.0\rangle, \{ack.yes, ack.no\})$. However, were *LeftImpl* to refuse $\{ack.yes, ack.no\}$ after $\langle in.0, data.0\rangle$ then that would be more significant: it would signify that the implementation process was refusing to progress its implementation of *send*.0 and so we might need to refuse $\{send.0\}$ at the corresponding point in the specification. As a result, it may be necessary to allow t from any failure (t, R) to influence what R is mapped to.

Only refusals contained in the alphabet of the process under consideration have the mapping applied to them and we then close up the refusal sets returned using $\Sigma - \lambda(\alpha Q)$. This reflects the fact that our mapped-to set of behaviours will only engage in events from $\lambda(\alpha Q)$ — if $events(t) \subseteq A$ for $A \in AllSet$ then $events(\lambda(t)) \subseteq \lambda(A)$ — and so may refuse all other events. Note also that the failures of $\lambda(\phi Q)$ are subset-closed. This plays an important role in allowing RAH1 to be met in this model: if $\alpha Q \subseteq Fvis$ then $\lambda(\phi Q) = \phi Q$ and so $\lambda(\phi Q)$ must meet the consistency condition SF3, requiring the subset-closure of failures.

We also impose some additional conditions on the mapping applied to refusal/trace pairs. REF-MONO makes the mapping over refusal/trace pairs monotonic in the refusal argument. REF-BOUND simply guarantees that, for a failure (t, R), $\lambda(R, t)$ is bounded according to the nature of t and R; this will prove to be useful in what follows. Note that $\lambda(R, t)$ is not required to be defined in the statement of REF-MONO. This is simply because, if $\lambda(S, t)$ is defined and $R \subseteq S$, then $\lambda(R, t)$ will be defined by definition 3.12(3).

REF-MONO	Let t be a trace, $R, S \subseteq \Sigma$ and $\lambda(S, t)$ be defined. If $R \subseteq S$ then $\lambda(R, t) \subseteq \lambda(S, t)$.
REF-BOUND	Let t be a trace, $R \subseteq \Sigma$ and $A \in AllSet$ be such that $\lambda(A) = B$ and $\lambda(R, t)$ is defined. If $events(t) \cup R \subseteq A$, then $\lambda(R, t) \subseteq B$.

3.5.2 Working in the stable failures model

We now render in the stable failures model the conditions RAH1-3, under the restrictions on hiding and parallel composition that were introduced in section 3.2.6. The new conditions, denoted SFI1-3, are given in figure 3.5. They may be used to show that the conditions TI1-3 hold in this model.

Theorem 3.21. *If conditions* SFI1-3 *hold then conditions* TI1-3 *also hold.*

Theorem 3.21 allows us to derive in the stable failures model all results given in sections 3.3 and 3.4 and so means that we can appeal to them here. Among other things, this means that we are able to reuse the conditions

SFI1 If $\alpha Q \subseteq Fvis$, then:
- $\lambda([\![Q]\!]_{SF})$ is defined.
- $\lambda([\![Q]\!]_{SF}) = [\![Q]\!]_{SF}$.

SFI2 If $\lambda([\![Q]\!]_{SF})$ is defined, $A \in AllSet$ and $\lambda(\backslash A) = \backslash B$, then:
- $\lambda([\![Q \setminus A]\!]_{SF})$ is defined
- $\lambda([\![Q \setminus A]\!]_{SF}) = \lambda([\![Q]\!]_{SF}) \setminus B$.

SFI3 If $\lambda([\![P]\!]_{SF})$, $\lambda([\![Q]\!]_{SF})$ are defined, $Y = \alpha P \cap \alpha Q$ and $\lambda(\|_Y) = \|_Z$, then:
- $\lambda([\![P \|_Y Q]\!]_{SF})$ is defined.
- $\lambda([\![P \|_Y Q]\!]_{SF}) = \lambda([\![P]\!]_{SF}) \|_Z \lambda([\![Q]\!]_{SF})$.

Figure 3.5: Rendering RAH1-3 in the stable failures model, where P and Q are implementation processes

from figure 3.4 when stating conditions sufficient for refinement-after-hiding in this model.

3.5.3 Parallel composition

In certain circumstances, for implementation processes P and Q we shall need to evaluate the result of composing $\lambda(\phi P)$ in parallel with $\lambda(\phi Q)$. The following result lets us do that in terms of the alternative semantics of parallel composition given in section 2.11. It will generally be appealed to implicitly whenever it is needed.

Theorem 3.22. *Let P and Q be implementation processes such that $\lambda(\phi P)$ and $\lambda(\phi Q)$ are defined. Let $Y = \alpha P \cap \alpha Q$ and $Z = \lambda(Y)$. Then:*

$$\begin{aligned}\lambda(\phi P) \|_Z \lambda(\phi Q) \;=\; & \{(t, S \cup U \cup R) \mid R \subseteq (\Sigma - (\lambda(\alpha P) \cup \lambda(\alpha Q))) \wedge \\ & ((\exists (s,S) \in \lambda(\phi P), (u,U) \in \lambda(\phi Q))\ t \in (s \|_Z u) \wedge \\ & S \subseteq \lambda(\alpha P) \wedge U \subseteq \lambda(\alpha Q))\}.\end{aligned}$$

3.5.4 From processes to individual behaviours

We now move on to to derive conditions governing the application of λ to individual refusal/trace pairs. In order to do this, we require equations governing the application of λ to sets of failures rather than to complete process denotations. We therefore derive the following results, which effectively recast conditions SFI1-3 in terms of failures alone (these restatements also take advantage of theorems 3.16 and 3.17).

Proposition 3.23. *Let Q be an implementation process. If $\alpha Q \subseteq Fvis$, then $\lambda(\phi Q)$ is defined and $\lambda(\phi Q) = \phi Q$.*

Proposition 3.24. *Let Q be an implementation process. If $\lambda([\![Q]\!]_{SF})$ is defined, $A \in AllSet$ and $\lambda(A) = B$, then:*

1. *$\lambda(\phi(Q \setminus A))$ is defined.*
2. *$\lambda(\phi(Q \setminus A)) = \lambda(\phi Q) \setminus B$.*

Proposition 3.25. *Let P, Q be implementation processes. If $\lambda([\![P]\!]_{SF})$, $\lambda([\![Q]\!]_{SF})$ are defined, $Y = \alpha P \cap \alpha Q$ and $\lambda(Y) = Z$, then:*

1. *$\lambda(\phi(P \parallel_Y Q))$ is defined.*
2. *$\lambda(\phi(P \parallel_Y Q)) = \lambda(\phi P) \parallel_Z \lambda(\phi Q)$.*

These results are used to derive conditions RAH1-SF, RAH2-SF and RAH3-SF below, which are the counterparts at the level of refusal/trace pairs of RAH1-3. RAH1-SF gives with regard to refusal/trace pairs the standard result that λ is the identity (in the refusal argument) where behaviours contained in *Fvis* are concerned. RAH3-SF shows that λ applied to refusal/trace pairs enjoys a distributivity property with respect to set union (by theorem 2.20, the effect of parallel composition on sets of refusals is given by set union). RAH2-SF(1) ensures that an implementation failure will be destroyed by hiding A only if the corresponding specification failure is destroyed by hiding $\lambda(A)$. RAH2-SF(2) will allow us to define $\lambda(R, t)$ compositionally (for certain R, t) in terms of $\lambda(R \cap A', t\lceil A')$ for $A' \in MinSet$.

Theorem 3.26 (RAH1-SF). *Let t be a trace and $R \subseteq \Sigma$ be such that $events(t) \cup R \subseteq Fvis$. Then $\lambda(R, t)$ is defined and $\lambda(R, t) = R$.*

Before giving the other two results, we make the following observation. Let P be a process and $(t, X) \in \phi P$ be refusal-maximal. Then, by PA2, $\Sigma - \alpha P \subseteq X$. As a result, we may partition any such X into $X \cap \alpha P$ and $\Sigma - \alpha P$. We take advantage of this fact in the statement and proofs of the following two results.

Theorem 3.27 (RAH3-SF). *Let P and Q be implementation processes such that $\lambda([\![P]\!]_{SF})$ and $\lambda([\![Q]\!]_{SF})$ are defined. Let $(s, S \cup (\Sigma - \alpha P)) \in \phi P$ be a refusal-maximal failure such that $S \subseteq \alpha P$. Let $(u, U \cup (\Sigma - \alpha Q)) \in \phi Q$ be a refusal-maximal failure such that $U \subseteq \alpha Q$. Moreover, let s, u be such that $s \parallel_Y u = \{t\}$, where $Y = \alpha P \cap \alpha Q$. Then $\lambda(S \cup U, t) = \lambda(S, s) \cup \lambda(U, u)$.*

Theorem 3.28 (RAH2-SF). *Let P be an implementation process such that $\lambda([\![P]\!]_{SF})$ is defined. Let $(t, R \cup (\Sigma - \alpha P)) \in \phi P$ be a refusal-maximal failure such that $R \subseteq \alpha P$. Let $A \in AllSet$ and $\lambda(A) = B$. Then:*

1. *If $A \subseteq R$ then $B \subseteq \lambda(R, t)$.*
2. *$\lambda(R - A, t \setminus A)$ is defined and $\lambda(R - A, t \setminus A) = \lambda(R, t) - B$.*

In the proofs of RAH2-SF and RAH3-SF, we construct processes such that the (refusal-maximal) failures given in the respective theorem statements are *maximal* in those constructed processes; moreover, it is also necessary that λ is defined over the denotation in the stable failures model of any such constructed process. It is for this to be possible that the failures given in the respective statements of RAH2-SF and RAH3-SF are required to be refusal-maximal. Note also that RAH2-SF(2) contains a result on definedness; this is simply to ease the proof of theorem 3.32(1) below. This is why RAH3-SF lacks a similar result. In general, it is not necessary to deal explicitly in such results with the definedness of λ over refusal/trace pairs, since this can usually be established using definition 3.12(3) and results derived in previous sections with respect to the traces model.

3.5.5 Sufficient conditions for refinement-after-hiding

Using conditions RAH1-SF, RAH2-SF and RAH3-SF, we are able to derive conditions sufficient to define a notion of refinement-after-hiding. Theorems 3.29 and 3.30 give conditions which must hold of λ applied to refusal/trace pairs at the level of *MinSet*. They are the counterparts at this level of RAH1-SF and RAH2-SF(1) respectively.

Theorem 3.29. *Let t be a trace, $R \subseteq \Sigma$ and $A \in MinSet$ be such that $events(t) \cup R \subseteq A \subseteq Fvis$. Then $\lambda(R, t)$ is defined and $\lambda(R, t) = R$.*

Proof. The proof is immediate by RAH1-SF. □

Theorem 3.30. *Let t be a trace such that $events(t) \subseteq A \in MinSet$. If $\lambda(A, t)$ is defined then $\lambda(A, t) = \lambda(A)$.*

Theorems 3.31 and 3.32 then give two different compositional rules with respect to defining λ over refusal/trace pairs. The requirement in each of them that $\lambda(t \circ \langle a \rangle)$ is defined for every a in a certain set of events is essentially the counterpart here to the requirement in RAH2-SF and RAH3-SF that the failures under consideration are refusal-maximal. If $(t, R \cup (\Sigma - \alpha P)) \in \phi P$ is refusal-maximal, where $R \subseteq \alpha P$, then $t \circ \langle a \rangle \in \tau P$ for every $a \in \alpha P - R$. Moreover, if $\lambda([\![P]\!]_{SF})$ is defined then $\lambda(\tau P)$ is defined and so $\lambda(t \circ \langle a \rangle)$ is defined for every $a \in \alpha P - R$. In theorem 3.31, A plays the role of αP while, in theorem 3.32, it is played by $[[events(t) \cup R]]$.

SFS1	The conditions in definition 3.12 are assumed to hold.
SFS2	If $\mathit{events}(t) \cup R \subseteq A \subseteq \mathit{Fvis}$, then $\lambda(R,t)$ is defined and $\lambda(R,t) = R$.
SFS3	If $\mathit{events}(t) \cup R \subseteq A$ and $\lambda(R,t)$ is defined then $\lambda(R,t) \subseteq \lambda(A)$.
SFS4	If $\mathit{events}(t) \subseteq A$ and $\lambda(A,t)$ is defined then $\lambda(A,t) = \lambda(A)$.
SFS5	If $\lambda(S,t)$ is defined and $R \subseteq S$, then $\lambda(R,t) \subseteq \lambda(S,t)$.
SFS6	If $\lambda(R,t)$, $\lambda(S,t)$ are defined, $\mathit{events}(t) \cup R \cup S \subseteq A$ and $\lambda(t \circ \langle a \rangle)$ is defined for every $a \in (A - R) \cup (A - S)$ then $\lambda(R \cup S, t) = \lambda(R,t) \cup \lambda(S,t)$.
SFS7	If $\lambda(R,t)$ is defined and $\lambda(t \circ \langle a \rangle)$ is defined for every $a \in [[\mathit{events}(t) \cup R]] - R$ then: - $\lambda(R \cap A', t\lceil A')$ is defined for every $A' \in \mathit{MinSet}$. - $\lambda(R,t) = \bigcup_{A' \in \mathit{MinSet}} \lambda(R \cap A', t\lceil A')$.

Figure 3.6: Sufficient conditions in the stable failures model, where t is a trace, $R, S \subseteq \Sigma$ and $A \in \mathit{MinSet}$

Theorem 3.31. *Let t be a trace and $R, S \subseteq \Sigma$ be such that $\lambda(R,t)$, $\lambda(S,t)$ are defined and $\mathit{events}(t) \cup R \cup S \subseteq A \in \mathit{MinSet}$. Moreover, assume that $\lambda(t \circ \langle a \rangle)$ is defined for every $a \in (A-R) \cup (A-S)$. Then $\lambda(R \cup S, t) = \lambda(R,t) \cup \lambda(S,t)$.*

Theorem 3.32. *Let t be a trace and $R \subseteq \Sigma$ such that $\lambda(R,t)$ is defined and $\lambda(t \circ \langle a \rangle)$ is defined for every $a \in [[\mathit{events}(t) \cup R]] - R$. Then:*

1. *$\lambda(R \cap A, t\lceil A)$ is defined for every $A \in \mathit{MinSet}$.*
2. *$\lambda(R,t) = \bigcup_{A \in \mathit{MinSet}} \lambda(R \cap A, t\lceil A)$.*

Figure 3.6 gives those conditions which, along with the conditions from figure 3.4, are sufficient to define a notion of refinement-after-hiding in the stable failures model. That this is the case is shown by theorem 3.33. Note also that SFS3 from figure 3.6 is a restatement of REF-BOUND at the level of sets from *MinSet*; moreover, SFS5 is simply REF-MONO. The part of condition SFS7 — taken from theorem 3.32(1) — which refers to definedness is not strictly necessary since it can be derived from other conditions in figures 3.4 and 3.6. However, it is included because it makes certain of the proofs more straightforward.

Theorem 3.33. *Let F_{impl} and F_{spec} be implementation and specification contexts respectively, containing n process variables, such that $\lambda(F_{impl}) = F_{spec}$. Let $Q_1, \dots Q_n$ be component implementation processes and $P_1, \dots, P_n$ be processes. Assume that the conditions in figures 3.4 and 3.6 all hold. If $Q_i \sqsupseteq^{\lambda}_{SF} P_i$ for $1 \leq i \leq n$ and $\alpha F_{impl}(Q_1, Q_2, \dots, Q_n) \subseteq Fvis$, then $F_{impl}(Q_1, Q_2, \dots, Q_n) \sqsupseteq_{SF} F_{spec}(P_1, P_2, \dots, P_n)$.*

3.5.6 Further comment regarding SFS4

Proposition A.28 in the appendix is weaker than its counterpart SFI2, since the former uses set containment, $\subseteq$, in its statement while the latter uses equality. This is for the following reason. It is possible to derive a version of RAH2-SF(1) as follows:

$$A \subseteq R \text{ if and only if } \lambda(A) \subseteq \lambda(R, t).$$

Had we proved this stronger version, then we could have derived a stronger version of SFS4 as follows, where we assume that $events(t) \cup R \subseteq A \in MinSet$ and $\lambda(R, t)$ is defined:

$$\lambda(R, t) = \lambda(A) \text{ if and only if } R = A.$$

And had such a condition been given in figure 3.6, then it would have been possible to prove a version of proposition A.28 which uses $=$ in place of $\subseteq$ and so which is equivalent to SFI2. However, had we used the stronger version of SFS4, it would have placed restrictions on λ which would have been too restrictive to be of use in practice. In particular, the means given in chapter 4 to define λ over refusal/trace pairs does not meet this condition. This is because, in the approach from chapter 4, $\lambda(R, t)$ for $events(t) \cup R \subseteq A \in MinSet$ such that $A \cap Fvis = \varnothing$ can only return $\varnothing$ or $\lambda(A)$. Thus, it is often the case that, in order to guarantee (an equivalent condition to) SFS6 is always met, $\lambda(R, t) = \lambda(A)$ even though $R \subset A$.

3.5.7 A comment on process alphabets

By SFS1 (definition 3.12(3b)), it can be seen that the value of $\lambda(\phi Q)$ is dependent on the value of αQ. It may be the case that processes with the same stable failures have different alphabets. However, the result of applying λ to the respective sets of stable failures will yield the same result whatever the alphabet. This is illustrated by the following result.

Theorem 3.34. *Let P and Q be implementation processes such that $\lambda(\tau P)$, $\lambda(\tau Q)$ are defined and $\phi P = \phi Q$. Then $\lambda(\phi P) = \lambda(\phi Q)$.*

We require here, for example, that $\lambda(\tau P)$ is defined rather than that $\lambda(\llbracket P \rrbracket_{SF})$ or $\lambda(\phi P)$ are defined, because the latter two can be reclaimed from the former. Moreover, as is shown by proposition A.34, if $\lambda(\llbracket P \rrbracket_{FD})$ is defined then $\lambda(\tau P)$ is also defined. This therefore makes clear the fact that theorem 3.34 is still valid in the failures divergences model: i.e. if $\lambda(\llbracket P \rrbracket_{FD})$ and $\lambda(\llbracket Q \rrbracket_{FD})$ are defined and $\phi P = \phi Q$, then theorem 3.34 may be used to show that $\lambda(\phi P) = \lambda(\phi Q)$.

3.6 The failures divergences model

We now move to consider the failures divergences model. The approach we take here is different to that followed with regard to the other two semantic models. In particular, our goal here is not to derive a theory as such. Rather, it is simply to derive a condition or conditions which may be used to augment those in figures 3.4 and 3.6 in order to define a notion of refinement-after-hiding in the failures divergences model. As a result, we assume that all of the conditions and restrictions imposed in section 3.5 still hold here. This means we can assume that all of the results derived earlier with respect to the stable failures and traces models still hold. The reason for this change of approach is that, were we to render RAH1-3 in this model, it is not clear how we would proceed. In sections 3.4 and 3.5, we relied on the fact that processes could be constructed in which there was a *unique* maximal behaviour, as a result of which an equality expressed in terms of process denotations — such as that given by TI2 — could be translated into an equality expressed in terms of individual behaviours, such as RAH2-T. This is no longer possible in the failures divergences model, partly because of the additional behaviours which are automatically generated by the closure conditions FD4 and FD5 and partly because the immediately divergent process can no longer be used to obscure failures as in the stable failures model. In relation to this latter point, recall that the process constructions used in the derivations of RAH2-SF and RAH3-SF in the previous section use extensively the fact that *DIV* can be used to obscure failures in the stable failures model.

The definition of applying λ to processes in the failures divergences model is therefore given in terms of its application to stable failures and minimally divergent traces (since $min\delta P \subseteq \tau P$ for any process P by MD, this means that minimally divergent traces may effectively be treated in the same way as non-divergent traces). In any case, this allows for a cleaner treatment since, by FD4 and FD5, it is unlikely that λ would be defined over *all* divergent traces and with respect to *all* (non-stable) failures of any particular implementation process.

Definition 3.13. *Let Q be an implementation process.*

FDI	Let Q be an implementation process such that $\lambda([\![Q]\!]_{FD})$ is defined. Moreover, let $A \in \mathit{AllSet}$ where $\lambda(\backslash A) = \backslash B$. Then $\lambda([\![Q \setminus A]\!]_{FD})$ is defined and $\lambda([\![Q \setminus A]\!]_{FD}) = \lambda([\![Q]\!]_{FD}) \setminus B$.

Figure 3.7: Rendering RAH2 in the failures divergences model

1. *$\lambda([\![Q]\!]_{FD})$ is defined if and only if $\lambda(\phi_\perp Q)$ and $\lambda(\delta Q)$ are defined.*
2. *If $\lambda([\![Q]\!]_{FD})$ is defined then $\lambda([\![Q]\!]_{FD}) \triangleq (\lambda(\phi_\perp Q), \lambda(\delta Q))$.*
3. *$\lambda(\delta Q)$ is defined if and only if $\lambda(\tau Q \cap \delta Q)$ is defined. If $\lambda(\delta Q)$ is defined, then $\lambda(\delta Q) \triangleq \{\lambda(t) \circ u \mid t \in min\delta Q \ \wedge \ u \in \Sigma^*\}$.*[18]
4. *$\lambda(\phi_\perp Q)$ is defined if and only if $\lambda(\phi Q)$ and $\lambda(\delta Q)$ are defined. If $\lambda(\phi_\perp Q)$ is defined, $\lambda(\phi_\perp Q) \triangleq \lambda(\phi Q) \cup \{(t, R) \mid t \in \lambda(\delta Q) \ \wedge \ R \subseteq \Sigma\}$.*

It turns out that we need only render RAH2 in this model and the relevant condition, FDI, is given in figure 3.7. However, it is necessary to impose the following additional condition.[19]

SEQ Let $\ldots, t_i, \ldots$ be an ω-sequence such that each $\lambda(t_i)$ is defined. Then there exists a deterministic implementation process Q such that $\tau Q = \mathit{Pref}(\{\ldots, t_i, \ldots\})$.

This reflects the intuition that any ω-sequence with which we might have to deal in practice may be generated by a syntactic term. That this condition is rather strong and is simply imposed is not so significant now that we are no longer aiming to derive a theory as such.

[18]Note that $min\delta Q \subseteq (\tau Q \cap \delta Q)$ by MD and since $min\delta Q \subseteq \delta Q$; thus, $\lambda(min\delta Q)$ is defined if $\lambda(\tau Q \cap \delta Q)$ is defined. We require that $\lambda(\tau Q \cap \delta Q)$ is defined rather than simply that $\lambda(min\delta Q)$ is defined because it allows us to infer the definedness of $\lambda(\tau Q)$ from the definedness of $\lambda([\![Q]\!]_{FD})$ (see proposition A.34 in appendix A.5). This is necessary if we are to define refinement-after-hiding in the failures divergences model using the conditions and results presented with respect to the stable failures model. In any case, minimally divergent traces are used here because the minimality property eases the proofs significantly; in practice — i.e. if defining algorithms for verification over (variants of) transition systems — we would work with $\tau Q \cap \delta Q$ because of the difficulty of establishing the minimality of any particular divergent trace and so would require λ to be defined over $\tau Q \cap \delta Q$. The notion of definedness used in the failures divergences model in chapter 4 is similar in that it, too, requires definedness over divergent traces to which the mapping is not actually applied. Note that, by TR-MONO and the definition of $\lambda(\delta Q)$, working with $\tau Q \cap \delta Q$ in place of $min\delta Q$ would not alter the result of any verification.

[19]$\{\ldots, t_i, \ldots\}$ is used to denote the set of traces which constitute the ω-sequence $\ldots, t_i, \ldots$.

FDS1	The conditions in definition 3.13 hold.
FDS2	Let $A \in \mathit{MinSet}$. Let $\ldots, t_i, \ldots$ be an ω-sequence such that $\lambda(t_i)$ is defined and $\mathit{events}(t_i) \subseteq A$ for each t_i. Then $\ldots, \lambda(t_i), \ldots$ is also an ω-sequence.

Figure 3.8: Sufficient conditions in the failures divergences model

Using FDI and SEQ, we are able to derive the following result, which is the only extra condition we need — other than those relating to the definition of λ applied to process denotations in this model — in order to define refinement-after-hiding in the failures divergences model.

Theorem 3.35. *Let $A \in \mathit{MinSet}$. Let $\ldots, t_i, \ldots$ be an ω-sequence such that $\lambda(t_i)$ is defined and $\mathit{events}(t_i) \subseteq A$ for each t_i. Then $\ldots, \lambda(t_i), \ldots$ is also an ω-sequence.*

This result is essentially used to guarantee that if a sequence of traces in an implementation process may lead to the presence of divergence after hiding then the corresponding sequence of traces in the specification process will also give rise to divergence after hiding. Note again the fact that we need only enforce the condition at the level of sets from *MinSet*.

The extra conditions from the failures divergences model are given in figure 3.8. We are then able to prove that these conditions, along with those used in the traces and stable failures models, are sufficient to define a notion of refinement-after-hiding in this model. Note that, in the results used in the proof of theorem 3.36, we sometimes introduce explicit specification processes rather than dealing only with mapped-to sets of behaviours. This is intended to simplify the presentation and to avoid the need to introduce additional notation in order to extract the failures and divergences respectively of any arbitrary failures/divergences pair. (Previously we avoided this problem by proving separately results relating to traces and to stable failures; however, it is not possible to separate the treatment of failures and divergences, due to the way in which they are calculated.)

Theorem 3.36. *Let F_{impl} and F_{spec} be implementation and specification contexts respectively, containing n process variables, such that $\lambda(F_{impl}) = F_{spec}$. Let $Q_1, \ldots Q_n$ be component implementation processes and $P_1, \ldots, P_n$ be processes. Assume that the conditions in figures 3.4, 3.6 and 3.8 all hold. If $Q_i \sqsupseteq^{\lambda}_{FD} P_i$ for $1 \leq i \leq n$ and $\alpha F_{impl}(Q_1, Q_2, \ldots, Q_n) \subseteq \mathit{Fvis}$, then $F_{impl}(Q_1, Q_2, \ldots, Q_n) \sqsupseteq_{FD} F_{spec}(P_1, P_2, \ldots, P_n)$.*

3.7 Further consideration of *BTrace* and related issues

In this section, we revisit some of the issues which have been highlighted through the course of this chapter, mainly in relation to *BTrace* and the restrictions that are imposed on sets by R1 and R2.

3.7.1 The role of restriction R1 and proposition 3.11

In section 3.2.6, we impose restrictions R1 and R2 on the hiding and parallel composition operators which may be used to build implementation networks from component implementation processes. R2 is then used to derive proposition 3.11 and both R1 and proposition 3.11 are of crucial importance in the derivation of the theory which has been presented in this chapter.[20] We here give further consideration to the roles which they play.

In the first instance, they allow us to characterise exactly the effect of applying λ to the hiding and parallel composition operators which may be used to build implementation networks (see theorems 3.16 and 3.17). More specifically, we are able to take advantage of the property that, for $t \in \mathit{BTrace}$ and $A \in \mathit{AllSet}$, either $\mathit{events}(t) \subseteq A$ or $\mathit{events}(t) \cap A = \varnothing$ by definitions 3.5 and 3.6. This allows the resolution of unknowns in certain equations, which then allows the use of those equations in the respective proofs of theorems 3.16 and 3.17. For example, RAH2-T states that $\lambda(t \setminus A) = \lambda(t) \setminus B$ for any trace t such that $\lambda(t)$ is defined and $A \in \mathit{AllSet}$, where $\lambda(\setminus A) = \setminus B$. In this equation, there are two unknowns, namely $\lambda(t \setminus A)$ and B. If we take t to be a member of *BTrace*, we can resolve $\lambda(t \setminus A)$ into either $\lambda(\langle\rangle) = \langle\rangle$ or $\lambda(t)$ and so can derive useful results on the nature of B. This approach is used in the proof of theorem 3.16, where we show that $\lambda(\setminus A) = \setminus\lambda(A)$ for $A \in \mathit{AllSet}$. If it were possible for A to be an arbitrary set, then this resolution of unknowns would not be possible. Similar comments apply with respect to the use of TI3 in the proof of theorem 3.17, which result shows that $\lambda(\|_Y) = \|_{\lambda(Y)}$ for $Y \in \mathit{AllSet}$.

The derivation of theorems 3.16 and 3.17 then has two main effects. Firstly, theorem 3.16 allows us to translate condition RAH2-T which refers to hiding into an equivalent condition which refers to projection: in other words, we are able to derive that $\lambda(t\lceil A) = \lambda(t)\lceil\lambda(A)$ for trace t and $A \in \mathit{AllSet}$ (see proposition A.3 in appendix A.3). This then lets us derive condition TS4 from figure 3.4, which gives a straightforward, compositional way to define the effect of λ applied to traces. Secondly, theorems 3.16 and 3.17 allow us

[20]Recall that R1 and proposition 3.11 require that the hiding and parallel composition operators which may be used to build implementation networks can be parameterized only by sets from *AllSet*.

to apply λ to the operators of hiding and parallel composition in what is effectively the same manner: we simply apply λ to the set with which the relevant operator is parameterized. This has the consequence that, in proving the sufficiency of the conditions from figure 3.4, condition Ts4 can be used to define the effect of λ when applied to traces generated using *either* hiding *or* parallel composition. This is significant because Ts4 is derived only from conditions — such as Ti2, Rah2-t and theorem 3.16 — which refer to the interaction between λ and the hiding operator: one would expect that a similar condition would have to be derived from conditions like Ti3 which refer specifically to the parallel composition operator.

3.7.2 The role of *BTrace*

In section 3.2, a very specific intuition behind *BTrace* was presented, namely that each specification action in Σ_{spec} may be implemented by a (finite) number of implementation traces and that *BTrace* consists of exactly those traces. Thereafter, *BTrace* is used in the definition of *MinSet* and so *AllSet*, and so plays a role in restricting those sets which are candidates for *AllSet*. Since implementation networks may be built using only hiding and parallel composition operators which are parameterized with sets from *AllSet*, the nature of *BTrace* plays a very significant role in determining the range of systems to which the theory presented in this chapter might be applied. There are, however, a number of comments to be made with regard to this.

Firstly, the intuition recalled above regarding the nature of *BTrace* is not recorded formally anywhere and plays no role in this chapter in the derivation of conditions sufficient to define a notion of refinement-after-hiding, nor does it appear in those conditions themselves. Although we would expect the intuition given to make sense in most cases where refinement-after-hiding might be used, it does not limit the systems to which the theory might be applied. Of course, we would still expect the traces contained in *BTrace* to be "atoms" or indivisible to the extent that it would not make sense to decompose them further into sub-traces, so that the restrictions imposed by R1 and proposition 3.11 may still be justified. Furthermore, there is no *explicit* counterpart to *BTrace* in the concrete notion of refinement-after-hiding which is presented in chapter 4, although such a notion may sometimes be used implicitly. The reasons for and consequences of this fact are discussed more fully in that chapter, once the concrete notion has been presented. It should be noted, however, that the absence of *BTrace* as an explicit notion in practice does not indicate a mismatch with the theory, nor does it mean that the sets in *MinSet* and *AllSet* could be made smaller in practice than is possible in the theory: i.e. it does not mean that the restrictions on hiding and parallel composition may be made lighter in practice.

3.7.3 Deriving the statement in definition 3.5(2)

Definition 3.5 states that *MinSet* is a partition of Σ_{impl} such that, if $t, u \in$ *BTrace* and $A, B \in$ *MinSet* where $A \neq B$ then:

1. If $\mathit{events}(t) \cap A \neq \varnothing$, then $\mathit{events}(t) \subseteq A$.
2. If $\mathit{events}(t) \subseteq A$ and $\mathit{events}(u) \subseteq B$, $\mathit{events}(\lambda(t)) \cap \mathit{events}(\lambda(u)) = \varnothing$.

In section 3.2, we stated that definition 3.5(2) may actually be derived as part of the theory, although this was realised only on a final revision of the thesis. Here, we show how to do that using only results which do not use definition 3.5(2) in their respective proofs.

Proposition 3.37. *Let $t, u \in$ BTrace and $A, B \in$ MinSet, where $A \neq B$. If $\mathit{events}(t) \subseteq A$ and $\mathit{events}(u) \subseteq B$ then $\mathit{events}(\lambda(t)) \cap \mathit{events}(\lambda(u)) = \varnothing$.*

Proof. By RAH2-T and theorem 3.16, $\lambda(t \setminus A) = \lambda(t) \setminus \lambda(A)$ and $\lambda(u \setminus B) = \lambda(u) \setminus \lambda(B)$. Hence, $\mathit{events}(\lambda(t)) \subseteq \lambda(A)$ and $\mathit{events}(\lambda(u)) \subseteq \lambda(B)$ by theorem 3.14. We prove that $\mathit{events}(\lambda(t)) \cap \mathit{events}(\lambda(u)) = \varnothing$ by assuming there exists $a \in \mathit{events}(\lambda(t)) \cap \mathit{events}(\lambda(u))$. Hence, $a \in \lambda(A)$. Again by RAH2-T and theorem 3.16, $\lambda(u \setminus A) = \lambda(u) \setminus \lambda(A)$. Thus, $\lambda(u) \setminus \lambda(A) \neq \lambda(u)$ since $a \in \lambda(u) \cap \lambda(A)$ and so $\lambda(u \setminus A) \neq \lambda(u)$. Hence, $\mathit{events}(u) \cap A \neq \varnothing$ and so $A \cap B \neq \varnothing$, which contradicts the fact that *MinSet* is a partition by definition 3.5(1). □

We observe that three results are appealed to in the proof of proposition 3.37: RAH2-T and theorems 3.14 and 3.16, none of the proofs of which refer to definition 3.5(2). The only derived results referred to in the proof of RAH2-T are from chapter 2. The only derived results referred to in the proof of theorem 3.16 are RAH2-T and theorem 3.14. Theorem 3.14 is a special case of RAH1-T and that part of the proof of RAH1-T which deals with this special case uses only a single derived result, from chapter 2. Hence, there is no circularity involved in the use of these results to derive the condition from definition 3.5(2).

3.8 Conclusion

Starting from a high-level statement of what it means to constitute a notion of refinement-after-hiding, we have presented in each of the three semantic models a set of conditions sufficient to define such a notion. Of particular significance is the fact that we need define directly the result of applying the mapping only at the level of sets from *MinSet* and any conditions imposed on it are enforced at this level. This allows any mapping used in practice to be defined compositionally and so allows for reuse of predefined mapping components. In the next chapter, we present a concrete notion of refinement-after-hiding which may be used in practice and in the development of which the conditions given here played a role. Since this was the intended purpose of the work in this chapter, we postpone until then a detailed discussion of its significance.

Chapter 4

A concrete notion of refinement-after-hiding

We now move on to present a concrete notion of refinement-after-hiding which may be used in practice. If we are to follow the template laid out in the previous chapter, we need three main things in order to proceed:

- counterparts to the sets from *MinSet* and counterparts, at the level of those sets, to the mapping defined over sets, traces and refusal/trace pairs.
- a set of compositional rules to allow general definitions to be built up from these component definitions.
- a structure within which these components can actually be defined.

The compositional rules used here are direct counterparts of S5 and Ts4 from figure 3.4 and SFS7 from figure 3.6. For the rest, we introduce the notion of *extraction pattern*.[1] An extraction pattern consists of a tuple and a set of conditions imposed on the constituent elements of that tuple; the exact nature of the tuple and these conditions depends on the semantic model in which we are working.

We first introduce extraction patterns in the traces model. All of the detail presented with respect to that model will continue to be relevant when we consider the other two semantic models. In particular, this is true of the detail on constructing a universe of extraction patterns, defining process alphabets and considering implementation networks and contexts.

[1]Extraction patterns appear in [16,39,40] among other papers; the name refers to the fact that they are used to "extract" specification behaviours from an implementation. Modifications made here to the notation are explained and highlighted in section 4.8.

EP1	A is a non-empty set of events, called the *implementation alphabet* and B is a non-empty set of events called the *specification alphabet.* Moreover, if $A \cap Fvis \neq \varnothing$ then $A \subseteq Fvis$.
EP2	Θ is a possibly empty set of events such that $\Theta \subseteq A$.
EP3-T	*Dom* is a non-empty, prefix-closed set of traces over the implementation alphabet.
EP4	*extr* is a strict, monotonic mapping defined for traces in *Dom*; for every $t \in Dom$, $extr(t)$ is a trace over the specification alphabet.

(a) General conditions.

	If $A \subseteq Fvis$ then:
EP1-FVI	$A = B$.
EP3-FVI	$Dom = A^*$.
EP4-FVI	If $events(t) \subseteq A$ then $extr(t) = t$.

(b) Over finally visible events.

Figure 4.1: Conditions on extraction patterns.

4.1 Extraction patterns in the traces model

An *extraction pattern* in the traces model is a tuple

$$ep \triangleq (A, B, \Theta, Dom, extr)$$

satisfying the conditions given in figure 4.1. (We assume that *Fvis* still denotes the set of finally visible events, as in the previous chapter.) The conditions from figure 4.1(a) relate to extraction patterns in the general case. EP3-T is used to denote the third condition there, rather than EP3, because it will be superseded by a different condition when working in the stable failures and failures divergences models. The conditions from figure 4.1(b) relate to the specific case that the extraction pattern is being used to interpret behaviours over finally visible events. They are labelled so as to relate each one to the corresponding condition from figure 4.1(a). We now relate these components and conditions to the sufficient conditions from the previous chapter where that is possible.[2]

The general case

The extraction pattern component A gives a set from *MinSet*, while B effectively gives $\lambda(A)$. Moreover, *extr* gives the mapping over traces from A^*; it is defined for all $t \in Dom$. The domain, *Dom*, of the mapping *extr* is given explicitly because it is used as part of the condition of refinement-after-hiding in all three semantic models. Θ is used as part of a condition relating to traces which makes our refinement-after-hiding relation larger than it would otherwise be: without it, it would be significantly more difficult to use the relation successfully in practice. Both of these features are discussed further where they are used.

Condition S4 from figure 3.4 implies that B should be fully characterised by *extr* and *Dom* in the following way:

$$B = \bigcup\{events(extr(t)) \mid t \in Dom\}.$$

By EP4, however, we may only infer that

$$\bigcup\{events(extr(t)) \mid t \in Dom\} \subseteq B.$$

Similarly, we only have by EP3-T that $\bigcup\{events(t) \mid t \in Dom\} \subseteq A$ rather than $\bigcup\{events(t) \mid t \in Dom\} = A$. Proceeding in this way simply gives greater flexibility and makes it easier to define these sets in practice: for example, if every event from any trace t over which *extr* is defined occurs on a channel b, it is easier to set A as αb even if not all of the events from αb are used. Finally, the part of EP4 which states that $extr(t)$ is a trace over the specification alphabet B is a counterpart to the second part of TS4 in figure 3.4.

The reader may observe that no explicit counterpart to *BTrace* is used in the definition of the extraction pattern components A and B; moreover, no such counterpart plays any role in the formal definition of the notion of refinement-after-hiding which is presented in this chapter. The reasons for this and other related issues are considered in section 4.7 at the end of the chapter.

Finally visible events

Turning to figure 4.1(b), EP1-FVI is a direct result of conditions S1(c) (definition 3.5), S4 and TS2 from figure 3.4. It is necessary because we state B

[2]Where this is not possible, it is generally the case that a feature which does not appear in the theory has been introduced in order to make things work better in practice.

EP-UNI1	Let $ep, ep' \in EP$ be such that $ep \neq ep'$. Then $ep.A \cap ep'.A = \varnothing$ and $ep.B \cap ep'.B = \varnothing$.
EP-UNI2	Let $a \in Fvis$. Then there exists $ep \in EP$ such that $a \in ep.A$.

Figure 4.2: Considering the universe of extraction patterns

directly and do not give a means of deriving it from A. EP3-FVI recognises the fact that, by TS2, if $events(t) \subseteq A$ and $A \subseteq Fvis$ then $extr(t)$ is defined. EP4-FVI also comes from condition TS2. Note that, for any extraction pattern ep such that $ep.A \subseteq Fvis$, we may dispense with the component Θ. The reason for this will become clear when its role is discussed in section 4.2.

4.1.1 Universe of extraction patterns

During any verification procedure, we assume the existence of a universe of extraction patterns, EP, containing all extraction patterns which may be used in the current verification. We impose on this universe the conditions EP-UNI1 and EP-UNI2 given in figure 4.2.[3] That $ep.A \cap ep'.A = \varnothing$ in EP-UNI1 comes from S1(c) (definition 3.5) in figure 3.4; that $ep.B \cap ep'.B = \varnothing$ comes from S1(c) (definition 3.5(2)) and S4. In the absence of an (explicitly stated) equivalent notion to Σ_{impl}, EP-UNI2 essentially gives the fact that $Fvis \subseteq \Sigma_{impl}$, as stated in S1(b)(definition 3.4).

EP may be used to define a counterpart to $AllSet$, as in the manner of S1(c) (definition 3.6), which we shall call here $ImplSet$.

Definition 4.1. $ImplSet \triangleq \{\bigcup C \mid C \in \mathbb{P}(\{ep.A \mid ep \in EP\})\}$.

We shall also need the equivalent of $\lambda(A)$ for any $A \in ImplSet$. We denote this $extr^{set}(A)$ and define it as follows.

Definition 4.2. *Let $A \in ImplSet$ be such that $A = \bigcup_{i \in I} ep_i.A$, where I is an indexing set into EP. Then $extr^{set}(A) \triangleq \bigcup_{i \in I} ep_i.B$.*

This effectively gives us the mapping applied to sets which is given in condition S5. Finally, EP-UNI2, definition 4.1 and EP1 effectively give a counterpart to S3: i.e. $Fvis \in ImplSet$.

[3]Note that we use $ep.A$ to denote the implementation alphabet, A, of the extraction pattern ep. Similarly, we may refer to $ep.B$, $ep.\Theta$ and so on.

4.1.2 Implementation and specification contexts

We again use F_{impl} and F_{spec} to denote corresponding implementation and specification contexts, each containing n free process variables, although they are slightly different to those used in chapter 3. In particular, F_{impl} may be defined using only the network composition operator, $\otimes_Y$, where $Y \in$ *ImplSet*.[4] Where $V_1, \dots, V_n$ and $W_1, \dots, W_n$ are free process variables:[5]

- $F_{impl} \triangleq (V_1 \otimes_{Y_1} V_2 \otimes_{Y_2} \dots \otimes_{Y_{n-1}} V_n)$
- $F_{spec} \triangleq (W_1 \otimes_{Z_1} W_2 \otimes_{Z_2} \dots \otimes_{Z_{n-1}} W_n)$, where $extr^{set}(Y_i) = Z_i$ for $1 \leq i \leq (n-1)$.

Implicit in the definition of F_{spec} are the conditions S6 and S7 from figure 3.4: we have mapped the operator we use by simply applying the necessary mapping to the sets with which it is parameterised. Since F_{impl} may only be defined using network composition, as soon as two implementation processes are composed in parallel during the construction of an implementation network, the set of events on which synchronization has occurred must be hidden. This has two effects, both of which are necessary. The first is that events may only be hidden *after* they have been synchronized on during parallel composition. This is relevant due to an issue raised when working with traces. The second is that only *two* processes in a particular network may synchronize on any particular set of events. The reason for this is bound up with the way in which the mapping is applied to refusals in practice. These issues are discussed further in sections 4.2 and 4.3 respectively.

4.1.3 Implementation processes and their interpretation

We define implementation processes as follows.

Definition 4.3. *Q is an implementation process if and only if, for every $a \in \beta(Q)$, there exists $ep \in EP$ such that $a \in ep.A$.*

This definition is a counterpart to S1(d) from figure 3.4 in the absence of an equivalent notion to Σ_{impl}; as in the case of S1(d), it will generally be appealed to implicitly. The following result states that the composition of any two implementation processes is also an implementation process and may be proved easily using definition 4.3 and the detail in figure 2.5.

[4]That $Y \in$ *ImplSet* does not pose a restriction in practice due to corollary 4.4 below and the restriction REP1 which is imposed on implementation networks in section 4.1.6.

[5]For the purposes of presentation, we have not bracketed here the expressions denoting the two contexts. In general, however, this would be necessary because of the fact that network composition, like parallel composition, is not associative.

TR-GLOBAL1	$Dom_{EP(Q)}$ is the set of $t \in (A_1 \cup \ldots \cup A_m)^*$ such that $t \lceil A_i \in Dom_i$ for $1 \leq i \leq m$.
TR-GLOBAL2	– $extr_{EP(Q)}(\langle\rangle) \triangleq \langle\rangle$. – Let $t \circ \langle a \rangle \in Dom_{EP(Q)}$ be such that $a \in A_i$ for $ep_i \in EP(Q)$. Then $extr_{EP(Q)}(t \circ \langle a \rangle) \triangleq extr_{EP(Q)}(t) \circ u$, where u is such that $extr_i(t \lceil A_i \circ \langle a \rangle) = extr_i(t \lceil A_i) \circ u$.

Figure 4.3: Global definitions in the traces model, where Q is an implementation process

Proposition 4.1. *Let P, Q be implementation processes and $Y \subseteq \Sigma$. Then $P \otimes_Y Q$ is also an implementation process.*

For any implementation process, Q, we shall need a *set* of extraction patterns with which to interpret its behaviour. For such a process Q we shall denote this set $EP(Q)$, defined as follows.[6]

Definition 4.4. $EP(Q) \triangleq \{ep \in EP \mid ep.A \cap \beta(Q) \neq \varnothing\}$.

For every event a in which Q may engage, there will therefore be $ep \in EP(Q)$ such that $a \in ep.A$. We assume throughout this chapter and chapter 6 that m gives the cardinality of $EP(Q)$ and $EP(Q) = \{ep_i \mid 1 \leq i \leq m\}$ (m gives this cardinality only for implementation processes with the label Q). Moreover, the various components of the extraction patterns in $EP(Q)$ can be subscripted to avoid ambiguity, giving $ep_i = (A_i, B_i, \Theta_i, Dom_i, extr_i)$ for $ep_i \in EP(Q)$. We then lift some of the notions introduced with respect to individual extraction patterns to the set $EP(Q)$. These are given in figure 4.3. TR-GLOBAL1 is similar in character to TS3 from figure 3.4 and describes a domain of traces $Dom_{EP(Q)}$. TR-GLOBAL2 is a counterpart to TS4 and describes a mapping $extr_{EP(Q)}$ which is defined for all traces in $Dom_{EP(Q)}$.

4.1.4 Process alphabets

For any implementation process, Q, the alphabet of Q is defined as follows.

Definition 4.5. $\alpha Q \triangleq \bigcup\{A_i \mid ep_i \in EP(Q)\}$.

[6]It can be seen that $EP(Q)$ depends on the syntactic form of Q; as in the previous chapter, however, the outcome of any verification will be the same for two processes with the same denotation.

In view of definitions 4.4 and 4.1, this definition effectively gives S2 from figure 3.4. The following important result is easy to prove using definitions 4.3, 4.4 and 4.5.

Proposition 4.2. *If Q is an implementation process then $\beta(Q) \subseteq \alpha Q$.*

Since the network composition operator is used only in a restricted way in the building of implementation networks — which restriction is imposed in section 4.1.6 as REP1 — we are able to show, among other things, that $EP(Q)$ is effectively defined compositionally (proposition 4.5) and αQ behaves in the way one would expect according to the detail in figure 2.5 (proposition 4.6).

Proposition 4.3. *Let P, Q be implementation processes and $Y = \alpha P \cap \alpha Q$. Then $Y = \bigcup\{A_i \mid ep_i \in EP(P) \cap EP(Q)\}$.*

Corollary 4.4. *Let P, Q be implementation processes. Then $\alpha P \cap \alpha Q \in ImplSet$.*

Proposition 4.5. *Let P, Q be implementation processes and $Y = \alpha P \cap \alpha Q$. Then $EP(P \otimes_Y Q) = (EP(P) \cup EP(Q)) - (EP(P) \cap EP(Q))$.*

Proposition 4.6. *Let P, Q be implementation processes and $Y = \alpha P \cap \alpha Q$. Then $\alpha(P \otimes_Y Q) = (\alpha P \cup \alpha Q) - (\alpha P \cap \alpha Q)$.*

4.1.5 Communication capabilities

We shall also need an additional notion relating to the sets of events in which any implementation process Q may engage, which will be used to restrict the nature of compositions which can occur. For any $ep_i \in EP(Q)$, we identify the *communication capability* of Q with respect to A_i, given by $Comm(A_i, Q)$, as either *Left* or *Right*. We will then require that two processes may synchronize on A_i only if it is labelled as *Left* in one and *Right* in the other. Further discussion of what this labelling actually means and is used for appears in sections 4.2 and 4.3.[7]

It is also necessary to define *Comm* compositionally with respect to the network composition operator (that we refer in this definition only to extraction patterns in $EP(P) - EP(Q)$ and $EP(Q) - EP(P)$ follows from proposition 4.5).

Definition 4.6. *Let P, Q be implementation processes and $Y = \alpha P \cap \alpha Q$.*

[7]The designation of a set of events as either *Left* or *Right* with respect to a particular process is arbitrary to an extent. However, extraction pattern components whose use in defining refinement-after-hiding is dependent on this designation must be defined with it in mind. This applies to Θ and *ref*, the latter being introduced in section 4.3.

1. *Let $ep_i \in EP(P) - EP(Q)$. Then* $Comm(A_i, P \otimes_Y Q) \triangleq Comm(A_i, P)$.

2. *Let $ep_j \in EP(Q) - EP(P)$. Then* $Comm(A_j, P \otimes_Y Q) \triangleq Comm(A_j, Q)$.

4.1.6 Restrictions on implementation networks

Where $Q_1, \ldots, Q_n$ are component implementation processes, the following restrictions are imposed on any implementation network $F_{impl}(Q_1, \ldots, Q_n)$. (Recall that the notation *Imp* is defined in definition 2.3 in section 2.7.)

REP1 Let $(P \otimes_Y Q) \in Imp(F_{impl}(Q_1, \ldots, Q_n))$. Then $Y = \alpha P \cap \alpha Q$.

REP2 Let $(P \otimes_Y Q) \in Imp(F_{impl}(Q_1, \ldots, Q_n))$ and $ep_i \in EP(P) \cap EP(Q)$. Then $Comm(A_i, P) \neq Comm(A_i, Q)$.

Condition REP1 is similar to R2 from chapter 3. REP2 is imposed for reasons bound up with the way in which the mapping is applied in practice to sets of refusals (see section 4.3 for further details), although it is also used in part of the condition for defining refinement-after-hiding in the traces model.

4.1.7 Extraction pattern for running example

We show here how we may construct an extraction pattern, ep_{ack}, to interpret in the traces model the behaviours of the processes *LeftImpl* and *RightImpl* from figure 1.1.[8] We first assume $Comm(\alpha data \cup \alpha ack, LeftImpl) = Left$ and $Comm(\alpha data \cup \alpha ack, RightImpl) = Right$. We also define:

$$\begin{aligned} Complete \triangleq\ & \{\langle data.0, ack.yes\rangle, \langle data.0, ack.no, data.0\rangle, \\ & \langle data.1, ack.yes\rangle, \langle data.1, ack.no, data.1\rangle\}^*. \end{aligned}$$

The components of $ep_{ack} = (A_{ack}, B_{ack}, \Theta_{ack}, Dom_{ack}, extr_{ack})$ are then defined as follows:

- $A_{ack} = \alpha data \cup \alpha ack$.
- $B_{ack} = \alpha send$.

[8] We do not explicitly give an extraction pattern to interpret the behaviours over channels *in* and *out*, since $\alpha in \cup \alpha out \subseteq Fvis$ and so the relevant structures may be constructed from the conditions in figure 4.1(b) and the fact that the Θ component is null in such cases. In any case, when we come to consider automatic verification in chapter 6, we need never explicitly construct such extraction patterns.

- $\Theta_{ack} = \varnothing$ (the reason for this choice is explained in section 4.2).
- $Dom_{ack} = Pref(Complete)$.

$extr_{ack}$ is defined as follows, where $t \in Complete$ and $t \circ u \in Dom$:

$$extr_{ack}(t \circ u) \triangleq \begin{cases} \langle\rangle & \text{if } t \circ u = \langle\rangle \\ extr_{ack}(t) \circ \langle send.v \rangle & \text{if } u = \langle data.v, ack.yes \rangle \\ & \text{or } u = \langle data.v, ack.no, data.v \rangle \\ extr_{ack}(t) & \text{otherwise} \end{cases}$$

Here, intuitively, $\langle send.0 \rangle$ may be implemented by two sequences of communications: $\langle data.0, ack.yes \rangle$ and $\langle data.0, ack.no, data.0 \rangle$ (and similarly for $\langle send.1 \rangle$).

4.2 Refinement-after-hiding in the traces model

We now move on to consider refinement-after-hiding proper in the traces model. The way in which processes are interpreted in this model is virtually the same as in the previous chapter. However, there is one significant difference. In chapter 3, $\lambda([\![Q]\!]_T)$ was defined if and only $\lambda(t)$ was defined for every $t \in \tau Q$ and verification could only proceed in the event that $\lambda([\![Q]\!]_T)$ was actually defined. Here, a less restrictive approach is taken.

4.2.1 Introducing a rely-guarantee condition

In order to explain this change of approach, consider *RightImpl* from the running example. In particular, we observe that *RightImpl* may engage in the trace $\langle data.0, ack.no, data.1 \rangle$: this trace may arise when a communication on channel *data* has been lost — hence the receipt of the negative acknowledgement — and *RightImpl* is ready to receive a retransmission of the data.[9] Since *RightImpl* should not know anything of the content of the failed transmission, then it must be ready to receive any possible retransmission: i.e. either *data*.0 or *data*.1. However, $\langle data.0, ack.no, data.1 \rangle$ is *not* a member of Dom_{ack} (see definition in section 4.1.7), since it is not clear what such a trace should be intended to implement. This means that $extr_{ack}$ is not defined for $\langle data.0, ack.no, data.1 \rangle$ and so, by TR-GLOBAL2, $extr_{EP(RightImpl)}$ is not defined for that trace either. Yet *LeftImpl* can never

[9]See process definitions in section 2.13.

perform a trace of which $\langle data.0, ack.no, data.1\rangle$ is a sub-trace and so all such "problem" traces from *RightImpl* will disappear after composition with *LeftImpl* (recall that *LeftImpl* and *RightImpl* synchronize on $A_{ack} = \alpha data \cup \alpha ack$). Moreover, $LeftImpl \otimes_{A_{ack}} RightImpl$ is a correct implementation of $LeftSpec \otimes_{B_{ack}} RightSpec$, even though $extr_{EP(RightImpl)}$ is not defined over every trace of *RightImpl*. Thus, we shall allow an implementation process Q to engage in behaviours which are outside of the domain $Dom_{EP(Q)}$, provided that composition with suitable implementation processes would remove these problem behaviours.[10] The extraction pattern components Θ_i and Dom_i and the notation $Comm(A_i, Q)$ for $ep_i \in EP(Q)$ are used for this purpose.

If $Comm(A_i, Q) = Left$, then Θ_i denotes those actions from A_i on which Q may go outside of the domain Dom_i. If $Comm(A_i, Q) = Right$, then $(A_i - \Theta_i)$ denotes the actions from A_i on which Q may go outside of the domain Dom_i. If implementation processes P and Q are composed during the building of an implementation network then $Comm(A_i, P) \neq Comm(A_i, Q)$ for $ep_i \in EP(P) \cap EP(Q)$, because of condition REP2. This means that the traces of P and Q respectively may move outside of Dom_i only on *different* events. As a result, composition will remove all behaviours which move outside of the domain on *any* of the events from A_i for $ep_i \in EP(P) \cap EP(Q)$: i.e. it will remove all behaviours which move outside of the domain on events on which we have to synchronize during the composition. $Proj_{EP(Q)}$, from the following definition, is used to give the set of actions from the entire process Q on which we may legitimately move outside of the domain $Dom_{EP(Q)}$.

Definition 4.7. *The following hold by definition, where Q is an implementation process and $ep_i \in EP(Q)$:*

1. *If $A_i \subseteq Fvis$ then $Proj_i \triangleq \varnothing$.*
2. *If $A_i \cap Fvis = \varnothing$,*

$$Proj_i \triangleq \begin{cases} \Theta_i & \text{if } Comm(A_i, Q) = Left \\ A_i - \Theta_i & \text{if } Comm(A_i, Q) = Right \end{cases}$$

3. *$Proj_{EP(Q)} \triangleq \bigcup\{Proj_i \mid ep_i \in EP(Q)\}$.*

[10]Note that a single composition need not remove all non-domain behaviours; however, all such behaviours will have been removed by the time that the implementation network under consideration engages only in finally visible events. Note also that the use of the network composition operator — instead of allowing the separate application of hiding and parallel composition — means a particular set of events can only be hidden *after* we have composed in parallel on them: this is necessary in order to make the proofs work in this section now that behaviours need not be contained in $Dom_{EP(Q)}$.

Dom-T-check	If $t\lceil Proj_{EP(Q)} \in (Dom_{EP(Q)}\lceil Proj_{EP(Q)})$ for $t \in \tau Q$ then $t \in Dom_{EP(Q)}$.

Figure 4.4: A rely-guarantee condition in the traces model, where Q is an implementation process

Condition Dom-T-check from figure 4.4 is then imposed on implementation processes; it is essentially a *rely-guarantee* condition in the sense of [18]. Provided that it holds of an implementation process Q, if we can *rely* on the fact that a particular trace $t \in \tau Q$ does not stray outside of $Dom_{EP(Q)}$ on the set of events on which it is allowed to, then we can *guarantee* that t is a member of the domain $Dom_{EP(Q)}$. By definition 4.7(1), we are able to ignore events from *Fvis* when considering Dom-T-check: this is acceptable because, by EP3-FVI, $ep.Dom = (ep.A)^*$ if $ep.A \subseteq Fvis$.

If we return to the running example, we have that $\Theta_{ack} = \varnothing$. Since $Comm(A_{ack}, LeftImpl) = Left$, then $Proj_{EP(LeftImpl)}$ is given by $\varnothing$ (recall that $\alpha in \subseteq Fvis$). Thus, Dom-T-check for *LeftImpl* becomes:

- If $\langle\rangle \in \{\langle\rangle\}$ for $t \in \tau LeftImpl$, then $t \in Dom_{EP(LeftImpl)}$.

In other words, every trace of *LeftImpl* has to be in $Dom_{EP(LeftImpl)}$ and this does, in fact, hold. Since $Comm(A_{ack}, RightImpl) = Right$, then $Proj_{EP(RightImpl)})$ is given by $A_{ack} - \Theta_{ack} = A_{ack}$ (recall that $\alpha out \subseteq Fvis$) and so Dom-T-check for *RightImpl* becomes:

- If $t\lceil A_{ack} \in (Dom_{EP(RightImpl)}\lceil A_{ack}) = Dom_{ack}$ for $t \in \tau RightImpl$, then $t \in Dom_{EP(RightImpl)}$.

This holds trivially by EP3-FVI and TR-GLOBAL1 and so Dom-T-check places no restriction at all on *RightImpl*. Because Dom-T-check requires that $t \in Dom_{EP(LeftImpl)}$ for every $t \in \tau LeftImpl$, then $t\lceil A_{ack} \in Dom_{ack}$ for all such t by TR-GLOBAL1. Thus, for $u \in \tau RightImpl$, we need not require that $u \in Dom_{EP(RightImpl)}$ — i.e. that $u\lceil A_{ack} \in Dom_{ack}$ — because no u for which this does not hold will be able to synchronize in parallel with any trace from *LeftImpl* and so any u for which it does not hold will be destroyed by composition with *LeftImpl*.

In general, Dom-T-check makes our notion of refinement-after-hiding larger than it would otherwise be: without it, we might dismiss component processes such as *RightImpl* as being incorrect, even though they may be used to build networks which are themselves correct. It is of most importance with respect to the input channels of any implementation process Q, which may be ready to receive all possible values that they can communicate, while $Dom_{EP(Q)}$ may not permit this.[11] This is acceptable in practice provided that the correponding output channel — to which the input channel will be connected — may only engage in the allowed behaviours. In such a situation, if $Comm(A_i, Q) = Left$ for $ep_i \in EP(Q)$ such that $A_i \cap Fvis = \varnothing$, we may define Θ_i to be the set of all input events from Q which are contained within A_i. Thus, Dom-T-check will allow Q to move outside the domain $Dom_{EP(Q)}$ on the events in Θ_i, while any process, P, with which Q might be composed would be allowed to move outside its domain on events in $A_i - \Theta_i$.[12] If $Comm(A_i, Q) = Right$, then we could define Θ_i to be such that $A_i - \Theta_i$ was the set of all input events from Q which are contained within A_i: i.e. Θ_i would be the set of all *output* events contained within A_i. Note, however, that we do not define Θ_{ack} to be αack, even though *ack* is an input channel in *LeftImpl* and $Comm(A_{ack}, LeftImpl) = Left$: this is because *RightImpl* fails to meet Dom-T-check when $\Theta_{ack} = \alpha ack$.

4.2.2 Defining refinement-after-hiding

The use of Dom-T-check means that, when we interpret the traces of an implementation process Q, we refer only to those which are also contained in $Dom_{EP(Q)}$ and these are given using the notation $\tau_{Dom_{EP(Q)}} Q$. (The following definition will generally be appealed to implicitly in proofs of results from this chapter.)

Definition 4.8. $\tau_{Dom_{EP(Q)}} Q \triangleq \tau Q \cap Dom_{EP(Q)}$.

The mapping $extr_{EP(Q)}$ is then overloaded to apply it to process denotations in the traces model and the fact that Q refines-after-hiding P in the traces model according to the set of extraction patterns $EP(Q)$ is denoted as $Q \sqsupseteq_T^{EP(Q)} P$. Condition TR-DEF1 from figure 4.5 shows how we interpret process denotations in the traces model and TR-DEF2 shows how refinement-after-hiding is defined here. Note that neither of these conditions contain any reference to the issue of definedness. Where Q is an implementation process, $extr_{EP(Q)}(t)$ is always defined for $t \in \tau_{Dom_{EP(Q)}} Q$ by definition 4.8 and so $extr_{EP(Q)}(\tau Q)$ is always defined by TR-DEF1.

Using these conditions, we are able to show that $\sqsupseteq_T^{EP(Q)}$ does indeed constitute a valid notion of refinement-after-hiding (see theorem 4.8). (Theorem 4.7 shows that we have generalised standard CSP refinement in the traces model, under the assumption that αQ contains only finally visible events.)

[11]This is the problem faced by *RightImpl* with respect to the channel *data* and the trace $\langle data.0, ack.no, data.1\rangle$.

[12]The events from Θ_i would occur on *output* channels in P; the events from $A_i - \Theta_i$ would occur on output channels in Q and so on input channels in P.

TR-DEF1	$extr_{EP(Q)}(\tau Q) \triangleq \{extr_{EP(Q)}(t) \mid t \in \tau_{Dom_{EP(Q)}} Q\}$.
TR-DEF2	$Q \sqsupseteq_T^{EP(Q)} P$ if and only if $extr_{EP(Q)}(\tau Q) \subseteq \tau P$ and Q meets Dom-T-check.

Figure 4.5: Defining refinement-after-hiding in the traces model, where Q is an implementation process and P is a process

Recall also before we proceed that conditions REP1 and REP2 are imposed on any implementation network $F_{impl}(Q_1, Q_2, \ldots, Q_n)$.

Theorem 4.7. *Let Q be an implementation process such that $\alpha Q \subseteq Fvis$ and let P be a process. Then $Q \sqsupseteq_T^{EP(Q)} P$ if and only if $Q \sqsupseteq_T P$.*

Theorem 4.8. *Let F_{impl} and F_{spec} be implementation and specification contexts respectively, as defined in section 4.1.2. Let $Q_1, \ldots Q_n$ be component implementation processes and $P_1, \ldots, P_n$ be processes. If $Q_i \sqsupseteq_T^{EP(Q_i)} P_i$ for $1 \leq i \leq n$ and $\alpha F_{impl}(Q_1, Q_2, \ldots, Q_n) \subseteq Fvis$, then $F_{impl}(Q_1, Q_2, \ldots, Q_n) \sqsupseteq_T F_{spec}(P_1, P_2, \ldots, P_n)$.*

4.3 Extraction patterns in the stable failures model

We now move on to consider the stable failures model and the notion of extraction pattern which must be used there. An extraction pattern in the stable failures model is a tuple

$$ep \triangleq (A, B, \Theta, dom, extr, ref),$$

where the component *dom* replaces *Dom* and we add the component *ref*. All conditions from figure 4.1 are assumed to hold as before, except that EP3-SF from figure 4.6 is used instead of EP3-T from figure 4.1; note, however, that EP3-SF implies EP3-T. Moreover, we assume that *Pref*(*dom*) returns the original *Dom* used in the traces model; in other words, we start with *Dom* and construct *dom* such that $dom \subseteq Dom$ and, for every $t \in Dom$, there exists $t \leq u \in dom$. In addition, conditions EP3A-FVI and EP5 from figure 4.6 are also imposed.

Neither *dom* nor *ref* are features of the theory from the previous chapter and both are introduced for the purposes of mapping refusal/trace pairs in practice. *dom* denotes those traces from *Dom* which are "complete" in some sense, while *ref* effectively defines a set of upper bounds on the refusals which

EP3-SF	*dom* is a non-empty set of traces over the implementation alphabet; *Dom* is given by its prefix-closure.
EP3a-FVI	If $A \subseteq \mathit{Fvis}$, then $\mathit{Dom} = \mathit{dom}$.
EP5	*ref* is a mapping defined for traces in *Dom* such that for every $t \in \mathit{Dom}$: - $\mathit{ref}(t)$ is a non-empty, subset-closed family of proper subsets of A. - if $a \in A$ and $t \circ \langle a \rangle \notin \mathit{Dom}$ then $R \cup \{a\} \in \mathit{ref}(t)$, for all $R \in \mathit{ref}(t)$.

Figure 4.6: Conditions on extraction patterns in the stable failures model

a process may exhibit after a particular trace (the roles of both of these components are discussed in more detail below). EP3A-FVI from figure 4.6 states that behaviour over finally visible events is always complete, the significance of which will become clearer below. The second part of EP5 is a counterpart in terms of refusal bounds to SF4, which guarantees that impossible events can always be refused. We now define $\mathit{extr}^{\mathit{ref}}$, the mapping applied to refusal/trace pairs whose events are contained in A (this is effectively $\lambda(R, t)$ for $\mathit{events}(t) \cup R \subseteq A' \in \mathit{MinSet}$).

4.3.1 Mapping refusals when $A \cap \mathit{Fvis} = \varnothing$

In order to define the mapping of refusal-trace pairs when $A \cap \mathit{Fvis} = \varnothing$, we use the notion of refusal bound, given by the extraction pattern component *ref*. Intuitively, refusal bounds are used to enforce progress after composition: if both participants in a parallel composition respect a particular set of bounds at some point in their joint evolution, then there will be at least one event which they both offer at that point and so on which they may synchronize in parallel (of course, due to the use of the network composition operator, this event would immediately be hidden after synchronization had occurred).[13] If a bound should be breached by a process Q — i.e. if Q should refuse too much at a particular point in time — then this places an obligation on the corresponding specification process to refuse a certain set of events. These issues are illustrated below using the running example. For reasons also explained below, the bounds used with respect to the two participants in a composition are asymmetric: that is, *ref* will be used to define the bounds

[13]See section 4.4.1 for a discussion of the significance of enforcing progress in this way.

for one of the participants, while another set of bounds is generated for the other process. Thus, for $t \in Dom$, we also introduce the notation $\overline{ref}(t)$ and define it as follows.

Definition 4.9. $\overline{ref}(t) \triangleq \{X \subseteq A \mid (\forall\, Y \in ref(t))\ X \cup Y \neq A\}$.

As a result, $X \in \overline{ref}(t)$ if and only if $X \subseteq A$ and, for every $Y \in ref(t)$, there exists $a \in A - Y$ such that $a \in A - X$. In section 4.3.4 below, we define ref_{ack}, the set of refusal bounds used in the verification of the running example. $ref_{ack}(\langle\rangle)$ is given by:

$$\{R \in 2^{\alpha data \cup \alpha ack} \mid \alpha data \not\subseteq R\}.$$

In other words, in order for this set of bounds to be respected, the relevant process must offer at least one event on channel *data* if it has not yet engaged in any communication on channels *data* and *ack*:[14] intuitively, the process must be ready to output a value on *data*. By definition 4.9, $\overline{ref}_{ack}(\langle\rangle)$ is given by $\{R \mid R \subseteq \alpha ack\}$ and so an implementation process must offer both events on channel *data* if these bounds are to be respected: intuitively, it must be ready to input a value on *data*. As a result of this, if one participant in a communication on channels *data* and *ack* respects the refusal bounds defined by $ref_{ack}(\langle\rangle)$ and the other respects the bounds defined by $\overline{ref}_{ack}(\langle\rangle)$, then the processes will be able to synchronize in parallel on at least one event on channel *data*. Likewise, if one of the processes breaches its refusal bounds, then it may be the case that parallel composition leads to (a local) deadlock on channels *data* and *ack*.

The refusal bounds defined by ref and $\overline{ref}$ are then used in the definition of $extr^{ref}$, the mapping applied to refusal-trace pairs whose events are contained in A. For $t \in Dom$, $R \subseteq A$ and implementation process Q such that $ep \in EP(Q)$, $extr^{ref}(R, t, Q)$ is defined as follows.

Definition 4.10. *If* $A \cap Fvis = \varnothing$:

$$extr^{ref}(R,t,Q) \triangleq \begin{cases} \varnothing & \text{if } (Comm(A,Q) = Left \ \wedge\ R \in ref(t)) \vee \\ & \quad\ (Comm(A,Q) = Right \ \wedge\ R \in \overline{ref}(t)) \\[1ex] B & \text{if } (Comm(A,Q) = Left \ \wedge\ R \notin ref(t)) \vee \\ & \quad\ (Comm(A,Q) = Right \ \wedge\ R \notin \overline{ref}(t)) \end{cases}$$

This definition is asymmetric in the sense that the outcome of applying the mapping to R, t is affected by whether Q is the "left" or "right" participant in any communication using the events A. In particular, the refusals

[14] That the process has not yet engaged in any communication on channels *data* and *ack* is indicated by the use of $\langle\rangle$ as an argument to ref_{ack}.

of Q will be considered with respect to ref if it is the left-hand participant and with respect to $\overline{ref}$ if it is the right-hand participant. Moreover, if a particular process breaches the bounds imposed on it, then $extr^{ref}$ will force the corresponding specification process to refuse B at the appropriate point: we have seen above in reference to the running example that a breach of refusal bounds may lead, after composition, to deadlock on the set of events A_{ack} and so $extr^{ref}$ would force the specification to deadlock on B_{ack} if this should occur.[15]

That the definition of $extr^{ref}(R, t, Q)$ is asymmetric in the sense described above is necessary since communication is generally asymmetric in nature. If b is regarded as an input channel in a particular process, then that process will usually offer all of the events on b; if it is an output channel, then the process may only offer a single event on b. As a result, the two parties to a communication may refuse very different sets of events from αb and so may respect very different refusal bounds, while still guaranteeing the progress after composition which we want.

This fact of asymmetricity is one of the reasons why we compose processes using only the network composition operator. Consider two implementation processes P and Q, where $ep_i \in EP(P) \cap EP(Q)$, $Y = \alpha P \cap \alpha Q$ and $Comm(A_i, P) \neq Comm(A_i, Q)$. Then it is not clear whether we should take $Comm(A_i, (P \parallel_Y Q)) = Left$ or $Comm(A_i, (P \parallel_Y Q)) = Right$. In fact, neither makes much sense and so $\parallel_Y$ may not be used in the definition of implementation contexts. In $P \otimes_Y Q$, by contrast, all events from A_i are hidden[16] and, by proposition 4.5, $ep_i \notin EP(P \otimes_Y Q)$.

We now consider how the definition of $extr^{ref}$ relates to the conditions given in figure 3.6. We first note that it meets implicitly condition SFS3. Condition SFS4 may be translated here to mean that $extr^{ref}(A, t, Q) = B$ and it is easy to show that it is met. By EP5 from figure 4.6, any set in $ref(t)$ must be a proper subset of A and so $A \notin ref(t)$. By definition 4.9, it cannot be the case that $A \in \overline{ref}(t)$. Hence, $extr^{ref}(A, t, Q) = B$ whether $Comm(A, Q) = Left$ or $Comm(A, Q) = Right$. Condition SFS5 on the monotonicity of the mapping in the refusal argument is met since both $ref(t)$ and $\overline{ref}(t)$ are subset-closed (the former follows by EP5 and the latter by definition 4.9).

That processes may be combined using only the network composition operator means condition SFS6 is not meaningful in the approach we take.[17]

[15]In some circumstances, we will actually disallow any breaching of refusal bounds: see condition Dom-SF-check in figure 4.9 and the related discussion in section 4.4.1.

[16]This is similar to the approach taken in CCS (see [50]): communication there is asymmetric since we can only synchronize an action a with its complement $\overline{a}$ and the result of the synchronization is hidden from view.

[17]Let P, Q be implementation processes. $\lambda(R \cup S, t)$ from SFS6 refers to R and S where

EP5-FVI	Let Q be an implementation process. If $A \subseteq \mathit{Fvis}$ and $\mathit{events}(t) \cup R \subseteq A$, then $\mathit{extr}^{\mathit{ref}}(R, t, Q) \triangleq R$.

Figure 4.7: Mapping refusals when $A \subseteq \mathit{Fvis}$

However, $\mathit{extr}^{\mathit{ref}}$ meets a similar condition which is sufficient in view of the restrictions placed on process composition. The condition is the following, where P, Q are implementation processes, $ep \in EP(P) \cap EP(Q)$, $\mathit{Comm}(A, P) \neq \mathit{Comm}(A, Q)$, $R \cup S \subseteq A$ and $t \in \mathit{Dom}$:

$$\text{if } R \cup S = A, \text{ then } \mathit{extr}^{\mathit{ref}}(R, t, P) \cup \mathit{extr}^{\mathit{ref}}(S, t, Q) = B.$$

If $R \cup S = A$ then, by definition of ref and $\overline{\mathit{ref}}$, either $\mathit{extr}^{\mathit{ref}}(R, t, P) = B$ or $\mathit{extr}^{\mathit{ref}}(S, t, Q) = B$. A possible alternative means of defining the mapping applied to refusals is given in section 4.8 below. It was suggested by the nature of condition SFS6 and so does meet it.

4.3.2 Mapping refusals when $A \subseteq \mathit{Fvis}$

In this case, the necessary definition is given by condition EP5-FVI in figure 4.7. It is a direct counterpart of condition SFS2 from figure 3.6.

4.3.3 From "local" to "global" definitions

As with Dom in TR-GLOBAL1, the notion of dom is lifted to the set of extraction patterns $EP(Q)$. It is also necessary to do the same with $\mathit{extr}^{\mathit{ref}}$. The relevant definitions are given in figure 4.8. Condition SF-GLOBAL2 is a counterpart of SFS7 from figure 3.6. When dealing with *sets* of extraction patterns, note that the components dom, ref, $\overline{\mathit{ref}}$ and $\mathit{extr}^{\mathit{ref}}$ may all be subscripted as in the traces model in order to avoid ambiguity.

4.3.4 Running example

We extend here the extraction pattern ep_{ack} so that it may be used to interpret in the stable failures model the processes *LeftImpl* and *RightImpl* from figure 1.1. dom_{ack} is defined as follows:

$$\begin{aligned} \mathit{dom}_{ack} \triangleq\ & \{\langle \mathit{data}.0, \mathit{ack}.\mathit{yes}\rangle, \langle \mathit{data}.0, \mathit{ack}.\mathit{no}, \mathit{data}.0\rangle, \\ & \langle \mathit{data}.1, \mathit{ack}.\mathit{yes}\rangle, \langle \mathit{data}.1, \mathit{ack}.\mathit{no}, \mathit{data}.1\rangle\}^*. \end{aligned}$$

R is part of a refusal from P, S is part of a refusal from Q and $R \cup S$ is part of a refusal from $P \parallel_Y Q$ for $Y = \alpha P \cap \alpha Q$.

SF-GLOBAL1	$dom_{EP(Q)}$ is the set of $t \in (A_1 \cup \ldots \cup A_m)^*$ such that $t\lceil A_i \in dom_i$ for $1 \le i \le m$.
SF-GLOBAL2	Let $R \subseteq \alpha Q$ and $t \in Dom_{EP(Q)}$. Then $extr^{ref}_{EP(Q)}(R, t, Q) \triangleq \bigcup_{1 \le i \le m} extr^{ref}_i(R \cap A_i, t\lceil A_i, Q)$.

Figure 4.8: Global definitions in the stable failures model, where Q is an implementation process

"Completeness" here — i.e. membership of dom_{ack} — means that a particular communication over the channels *data* and *ack* has been completed. Since Dom_{ack} is given as the prefix-closure of dom_{ack}, its definition has not changed (see section 4.1.7). The component ref_{ack}, where $t \in dom_{ack}$ and $t \circ u \in Dom_{ack}$, is defined as:

$$ref_{ack}(t \circ u) \triangleq \begin{cases} 2^{\alpha data} & \text{if } u = \langle data.v \rangle \\ \{R \in 2^{\alpha data \cup \alpha ack} \mid \alpha data \not\subseteq R\} & \text{if } u = \langle\rangle \\ \{R \in 2^{\alpha data \cup \alpha ack} \mid data.v \notin R\} & \text{if } u = \langle data.v, ack.no \rangle \end{cases}$$

Recall that $Comm(\alpha data \cup \alpha ack, LeftImpl) = Left$ and $Comm(\alpha data \cup \alpha ack, RightImpl) = Right$. This means that *LeftImpl* is considered with respect to ref_{ack} and *RightImpl* is considered with respect to $\overline{ref}_{ack}$. Assuming that both processes always respect the refusal bounds, then the following will hold. When behaviour is complete, *LeftImpl* will offer at least one event on the channel *data*; *RightImpl* will be ready to receive any event on that channel (see statement of $\overline{ref}_{ack}(\langle\rangle)$ in section 4.3.1 above). After a single data transmission, *LeftImpl* will accept any event on channel *ack* and so, by definition 4.9, *RightImpl* need only offer one of those events. Finally, after a negative acknowledgement has been communicated, *LeftImpl* will offer the necessary data retransmission and *RightImpl* will be ready to receive it. Thus, if both implementation components respect the refusal bounds, they will always synchronize on at least one event from the channels *data* and *ack* when they are composed in parallel.

4.4 Refinement-after-hiding in the stable failures model

We now move on to consider how the above detail may be used to define a notion of refinement-after-hiding in the stable failures model. Before proceeding, note that all definitions, conditions and restrictions from sections

Dom-SF-check	Let $(t, R) \in \phi_{Dom_{EP(Q)}} Q$ be such that $R \subseteq \alpha Q$. Let $ep_i \in EP(Q)$ be such that $A_i \cap Fvis = \varnothing$. If $extr_i^{ref}(R \cap A_i, t\lceil A_i, Q) = B_i$ then $t\lceil A_i \in dom_i$.

Figure 4.9: Extra condition on failures, where Q is an implementation process

4.1 and 4.2 are assumed to still apply in this model. Where Q is an implementation process, we shall not wish to consider those failures of Q whose trace component is not in $Dom_{EP(Q)}$. In addition, it will also be necessary to isolate those failures whose trace component is from $dom_{EP(Q)}$. We therefore introduce the following notation. (As for definition 4.8, this definition will generally be appealed to implicitly in proofs of results from this chapter.)

Definition 4.11. *Let Q be an implementation process. Then the following hold by definition.*

1. *$\phi_{Dom_{EP(Q)}} Q$ is the set of those stable failures of Q in which the trace component belongs to $Dom_{EP(Q)}$.*
2. *$\phi_{dom_{EP(Q)}} Q$ is the set of those stable failures of Q in which the trace component belongs to $dom_{EP(Q)}$.*

In chapter 3, $\lambda(\phi Q)$ was defined by applying λ to R, t for all $(t, R) \in \phi Q$ (provided that $R \subseteq \alpha Q$). Here, however, we shall apply $extr_{EP(Q)}^{ref}$ only to refusal/trace pairs from failures in $\phi_{dom_{EP(Q)}} Q$: that is, those failures whose trace components are complete in some sense. It is for this reason that the *dom* component is introduced (see section 4.4.1 for more details). In tandem with taking this approach, we also require that condition Dom-SF-check from figure 4.9 is met. As a result of this, the notion of refinement-after-hiding used in the stable failures model — $\sqsupseteq_{SF}^{EP(Q)}$ — is as defined in figure 4.10. ($extr_{EP(Q)}$ is again overloaded and may be applied to denotations in the stable failures model or to sets of stable failures.) As in the traces model, and for similar reasons, we do not consider explicitly issues of definedness in the conditions in figure 4.10. Note also that, by proposition B.14 in appendix B, $dom_{EP(Q)} \subseteq Dom_{EP(Q)}$ for any implementation process Q and so $extr_{EP(Q)}^{ref}(R, t, Q)$ is defined in the statement of SF-DEF2.

4.4.1 Role of Dom-SF-check

We first discuss briefly how the condition Dom-SF-check actually works before considering why it is necessary to use it. This will highlight the need for the *dom* component.

SF-DEF1	$extr_{EP(Q)}([\![Q]\!]_{SF}) \triangleq (extr_{EP(Q)}(\tau Q), extr_{EP(Q)}(\phi Q))$.
SF-DEF2	$extr_{EP(Q)}(\phi Q) \triangleq \{(extr_{EP(Q)}(t), X) \mid (t, R) \in \phi_{dom_{EP(Q)}} Q \wedge R \subseteq \alpha Q \wedge X \subseteq extr^{ref}_{EP(Q)}(R, t, Q) \cup (\Sigma - extr^{set}(\alpha Q))\}$.
SF-DEF3	$Q \sqsupseteq^{EP(Q)}_{SF} P$ if and only if $extr_{EP(Q)}([\![Q]\!]_{SF}) \subseteq [\![P]\!]_{SF}$ and Q meets Dom-T-check and Dom-SF-check.

Figure 4.10: Defining refinement-after-hiding in the stable failures model, where Q is an implementation process and P is a process

Dom-SF-check amounts to the requirement that refusal bounds may only be breached when behaviour is complete (recall that, by EP3A-FVI, behaviour over finally visible events is always complete).[18] This forces progress — that is, the enabling of at least one event — after parallel composition on the set of events $ep.A$, where $ep.A \cap Fvis = \varnothing$, when behaviour is not complete with respect to $ep.dom$.[19] Since all events on which synchronization occurs are immediately hidden, this enabled event will be hidden and so the corresponding state will not contribute to a stable failure of the composition. This is why, by SF-DEF2, it is only necessary to find a matching failure in the specification component when behaviour in the implementation is complete according to $dom_{EP(Q)}$.

It would be possible to define a sound notion of refinement-after-hiding in this model where $dom_{EP(Q)}$ in condition SF-DEF2 in figure 4.10 was replaced by $Dom_{EP(Q)}$ and condition Dom-SF-check was dispensed with. Were we to do such a thing, however, then the practical applicability of the method would be much reduced.[20] This can be illustrated using the following small example. We assume a (deterministic) specification process which executes the trace $\langle in.0, send.0, out.0\rangle$ and which refuses all other events. All events on the channels *in* and *out* are assumed to be contained in *Fvis*. Consider an (deterministic) implementation process, Q, which executes the trace $\langle in.0, a_1, \ldots, a_k, out.0\rangle \in Dom_{EP(Q)}$ and refuses all other events, where $\langle a_1, \ldots, a_k\rangle$ implements $\langle send.0\rangle$ in some way. In the general case, where $t = \langle in.0, a_1, \ldots, a_l\rangle$ for $l < k$, it is possible that $extr_{EP(Q)}(t) = \langle in.0, send.0\rangle$. In other words, it may be the case that we interpret *send*.0 as having oc-

[18] If $extr^{ref}_i(R \cap A_i, t\lceil A_i, Q) = B_i$ as in the statement of the condition, then this signifies, by definition 4.10, that the relevant refusal bounds have been breached.

[19] See the discussion in section 4.3.4 of the refusal bounds used in relation to the running example.

[20] A similar issue is considered in [56] in a bisimulation-type setting; see chapter 5 for further details.

curred before its implementation has actually completed; although this seems counter-intuitive, it can be necessary in practice and examples of the need for this can be seen in chapter 7. After executing a_l, the implementation process refuses all events but a_{l+1}. If we consider R to be the largest set refused after t in Q such that $R \subseteq \alpha Q$, then $out.0 \in R$. It follows by SF-GLOBAL2 and EP5-FVI that $out.0 \in extr^{ref}_{EP(Q)}(R, t, Q)$. But the specification does not refuse $out.0$ after $extr_{EP(Q)}(t) = \langle in.0, send.0 \rangle$ and so this verification would fail. However, if we use condition Dom-SF-check and a set of refusal bounds which allow us to refuse after t everything but a_{l+1} then the verification could succeed.[21]

The problem therefore stems from the interaction between the possibility of interpreting that a high-level event has occurred before its low-level implementation has completed and the fact that finally visible events must be preserved when refusal sets have the mapping applied to them. Thus, we only relate implementation to specification failures when behaviour is complete according to $dom_{EP(Q)}$. Behaviours over finally visible events are regarded as always complete by EP3A-FVI since this problem cannot arise with regard to them.

4.4.2 Soundness of refinement-after-hiding

We now proceed to show that $\sqsupseteq^{EP(Q)}_{SF}$, defined in figure 4.10, does indeed constitute a valid notion of refinement-after-hiding in the stable failures model, after first showing that it generalises standard CSP refinement when $\alpha Q \subseteq Fvis$.

Theorem 4.9. *Let Q be an implementation process such that $\alpha Q \subseteq Fvis$ and let P be a process. Then $Q \sqsupseteq^{EP(Q)}_{SF} P$ if and only if $Q \sqsupseteq_{SF} P$.*

Theorem 4.10. *Let F_{impl} and F_{spec} be implementation and specification contexts respectively, as defined in section 4.1.2. Let $Q_1, \ldots Q_n$ be component implementation processes and $P_1, \ldots, P_n$ be processes. If $Q_i \sqsupseteq^{EP(Q_i)}_{SF} P_i$ for $1 \leq i \leq n$ and $\alpha F_{impl}(Q_1, Q_2, \ldots, Q_n) \subseteq Fvis$, then $F_{impl}(Q_1, Q_2, \ldots, Q_n) \sqsupseteq_{SF} F_{spec}(P_1, P_2, \ldots, P_n)$.*

4.5 The failures divergences model

We finally consider the case of the failures divergences model. We assume that an extraction pattern is defined here as in the stable failures model and that all conditions and restrictions imposed in that model still hold here. We

[21] We would assume that behaviour in the implementation was incomplete according to $dom_{EP(Q)}$ *after* the execution of a_1 and *prior* to the execution of a_k.

EP6	Let $ep \in EP$. If $\ldots, t_j, \ldots$ is an ω-sequence in *Dom*, then $\ldots, extr(t_j), \ldots$ is also an ω-sequence.

Figure 4.11: A final condition on extraction patterns

then impose an extra condition on the mapping over traces, as a counterpart to condition FDS2 in figure 3.8. This condition is given in figure 4.11.

Behaviours in this model are interpreted in a manner directly analogous to that given in definition 3.13 in the previous chapter: see figure 4.12 for details.[22] We denote the fact that Q refines-after-hiding P in the failures divergences model as $Q \sqsupseteq_{FD}^{EP(Q)} P$; its definition is given as condition FDEF4 in figure 4.12. We show that $\sqsupseteq_{FD}^{EP(Q)}$ generalises standard CSP refinement when $\alpha Q \subseteq Fvis$ before showing that it constitutes a valid notion of refinement-after-hiding in the failures divergences model.

Theorem 4.11. *Let Q be an implementation process such that $\alpha Q \subseteq Fvis$ and let P be a process. Then $Q \sqsupseteq_{FD}^{EP(Q)} P$ if and only if $Q \sqsupseteq_{FD} P$.*

Theorem 4.12. *Let F_{impl} and F_{spec} be implementation and specification contexts respectively, as defined in section 4.1.2. Let $Q_1, \ldots Q_n$ be component implementation processes and $P_1, \ldots, P_n$ be processes. If $Q_i \sqsupseteq_{FD}^{EP(Q_i)} P_i$ for $1 \leq i \leq n$ and $\alpha F_{impl}(Q_1, Q_2, \ldots, Q_n) \subseteq Fvis$, then $F_{impl}(Q_1, Q_2, \ldots, Q_n) \sqsupseteq_{FD} F_{spec}(P_1, P_2, \ldots, P_n)$.*

4.6 Equivalence

Thus far, we have defined conditions in each of the three semantic models which allow us to show that an implementation network refines the corresponding specification network. If we wish to show that the two networks are equivalent, it is simply necessary to show that each *specification* component refines-after-hiding the corresponding *implementation* component, as well as the fact that each implementation component refines-after-hiding the corresponding specification component.[23] (Different — though not necessarily disjoint — sets of extraction patterns would be used in each case of

[22]As in chapter 3, the use of stable failures and minimally divergent traces allows us to build refinement-after-hiding in the failures divergences model on top of the treatment presented for the traces and stable failures models. Also as in chapter 3, it allows for a cleaner treatment due to the difficulty of making sure that any mapping used is defined over arbitrary divergent traces and with respect to arbitrary (non-stable) failures of any particular implementation process.

[23]With regard to the notion of refinement-after-hiding presented here, and as in the previous chapter, the terms "implementation component" and "specification component"

FD-DEF1	$extr_{EP(Q)}([\![Q]\!]_{FD}) \triangleq (extr_{EP(Q)}(\phi_{\perp}Q), extr_{EP(Q)}(\delta Q))$.
FD-DEF2	$extr_{EP(Q)}(\delta Q) \triangleq \{extr_{EP(Q)}(t) \circ u \mid t \in Dom_{EP(Q)} \wedge t \in min\delta Q \wedge u \in \Sigma^*\}$.
FD-DEF3	$extr_{EP(Q)}(\phi_{\perp}Q) \triangleq extr_{EP(Q)}(\phi Q) \cup \{(t, R) \mid t \in extr_{EP(Q)}(\delta Q) \wedge R \subseteq \Sigma\}$.
FD-DEF4	$Q \sqsupseteq_{FD}^{EP(Q)} P$ if and only if $extr_{EP(Q)}([\![Q]\!]_{FD}) \subseteq [\![P]\!]_{FD}$ and Q meets Dom-T-check and Dom-SF-check.

Figure 4.12: Defining refinement-after-hiding in the failures divergences model, where Q is an implementation process and P is a process

course.) When verifying a specification process against an implementation process in either the stable failures or failures divergences models, it should generally be possible to regard all behaviours as complete (i.e. to assume that $Dom_{EP(Q)} = dom_{EP(Q)}$). As a result, and by SF-GLOBAL1, Dom-SF-check would be met trivially and so could be dispensed with. This is possible since the problem which Dom-SF-check is intended to address should not arise.

4.7 The absence of *BTrace* and defining implementation alphabets

In chapter 3, *BTrace* is used to provide an exact characterisation of the sets which constitute *MinSet* and of the effect of applying λ to members of that set. These precise characterisations are necessary in order to derive certain results which are part of the theory presented in that chapter. However, when working in practice, we simply impose as a condition or definition any results which were previously derived and so these exact characterisations are no longer needed; thus, *BTrace* need not be an explicit part of the treatment in practice. For example, for $ep, ep' \in EP$ such that $ep \neq ep'$, $ep.A$ and $ep'.A$ are effectively sets from *MinSet*, while $ep.B$ and $ep'.B$ give $\lambda(ep.A)$ and $\lambda(ep'.A)$ respectively. In chapter 3, the use of *BTrace* plays an important role in the derivation of the requirement that $ep.B \cap ep'.B = \varnothing$. Here, however, we simply impose that condition directly using EP-UNI1 from figure 4.2.

indicate respectively the process whose behaviours are to be interpreted and the process in whose behaviours we will check for containment of those interpreted behaviours. They have no further significance than this and an extraction mapping may function either to make behaviours more abstract or to make them more concrete (or it may leave the level of abstraction the same if we are dealing only with relaxation of atomicity).

BTrace also plays a role in chapter 3 in defining those sets which are possible candidates for *MinSet* (and so thereby restricts the sets which can be used to parameterize the operators used to build implementation networks[24]). Nonetheless, the absence of *BTrace* as an explicit notion in practice does not indicate a mismatch with the theory. In order to explore this issue further, we consider how extraction pattern implementation alphabets — which are the counterpart here to sets from *MinSet* — might be constructed in practice.

Assume that we are engaged on the verification of an implementation process, Q, and so it is necessary to define a set of extraction patterns, $EP(Q)$, to interpret Q's behaviours. In particular, we wish to make the implementation alphabets of these extraction patterns as *small* as possible, so that the restriction on building implementation networks which is imposed by REP1 becomes as light as possible.[25] TR-GLOBAL2 from figure 4.3 consitutes one of the main conditions from this chapter which is in opposition to this desire, as it effectively places a lower bound on the size of any particular implementation alphabet. Let $t \circ \langle a \rangle \in Dom_{EP(Q)}$ be such that $a \in A_i$ for $ep_i \in EP(Q)$ and $extr_{EP(Q)}(t \circ \langle a \rangle) = extr_{EP(Q)}(t) \circ u$. Then, by TR-GLOBAL2, u is evaluated according to the following equation: $extr_i(t\lceil A_i \circ \langle a \rangle) = extr_i(t\lceil A_i) \circ u$. Thus, A_i must be large enough so that is possible to evaluate $extr_i(t\lceil A_i \circ \langle a \rangle)$ in practice. In other words, A_i must be large enough so that $t\lceil A_i$[26] gives sufficient information to allow us to intepret what additional information — i.e. u — is given to us by the occurrence of a after t (this issue was discussed briefly in section 3.4). This means that the implementation alphabets to be used in any particular verification in practice must be built around (the events of) traces in terms of which we are able to evaluate directly the result of applying the extraction mapping. Moreover, in order to make those alphabets as small as possible, the traces around which they are built must be "atoms" or "indivisible" in the sense that they cannot be decomposed further for the purposes of interpretation: this reflects the use of *BTrace* — containing traces which may be regarded as atoms or as indivisible in some sense[27] — in defining *MinSet* in section

[24]Recall that *AllSet* is defined in terms of *MinSet*.

[25]By definition 4.5, the bigger the implementation alphabets of the extraction patterns used then the bigger the alphabets of the implementation processes to be composed and so, by REP1, the bigger the sets on which they have to synchronize in parallel.

[26]Recall that $a \in A_i$ by definition.

[27]The particular intuition given in section 3.2 with respect to *BTrace* — namely that each specification action in Σ_{spec} may be implemented by a (finite) number of implementation traces and that *BTrace* consists of exactly those traces — may best be regarded as an aid to explanation. The significant property of *BTrace* is the fact that it consists only of traces which may be regarded as indivisible in *some* sense. As mentioned in section 3.7, it is this property which is necessary if the restrictions imposed in section 3.2.6 by R1 and proposition 3.11 are to make sense. Nonetheless, even if we do make the assumption that

3.2. Thus, there is no inconsistency between the use of *BTrace* as a device to aid the derivation of the theory presented in chapter 3 and the fact that it does not appear as part of the notion of refinement-after-hiding which is given in this chapter.

The previous paragraph gives some indication of how the implementation alphabets used in any particular verification may be arrived at. In practice, we would probably have a different implementation alphabet for each "portion" of communication in which our process engaged. For example, we might have (at least) a different implementation alphabet — and so a different extraction pattern — for each process with which our implementation process communicated: if that part of the behaviour of our implementation process which is visible at the relevant interface makes sense to the process on the other side of that interface, then it should usually be possible to define an extraction mapping to interpret those behaviours. However, the process of deriving implementation alphabets to be used in practice is still very much an area for further work and development of a methodology with respect to this will rely on the application of refinement-after-hiding to a far wider range of case studies than has so far been considered.

As a final comment, note that *BTrace* may appear *implicitly* in the definition of extraction mapping domains and in the definition of the mappings themselves. For example, the traces used in the definition of *Complete* in section 4.1.7 could constitute traces from *BTrace* were such a notion to be used explicitly here and both Dom_{ack} and $extr_{ack}$ are defined in terms of them. Indeed, the definition of $extr_{ack}$ shows one way of giving a compositional definition of an extraction mapping over traces whose events fall completely within a particular implementation alphabet (the formal definition of refinement-after-hiding is silent on this issue).

4.8 Discussion

We now proceed to discuss the way in which the concrete notion of refinement-after-hiding presented in this chapter relates to its predecessors and the way in which its development has been impacted on by the theory developed in the previous chapter. Predecessors of the work in this chapter include [10,12,16,39,40]. [39] and [40] treat differently the case of cyclic and acyclic implementation networks and impose on specification components a number of restrictions, such as the fact of never refusing any input. These treatments were combined into a single notion of implementation relation in

each specification action in Σ_{spec} may be implemented by a (finite) number of implementation traces, we would generally expect each such trace to be regarded as indivisible for the purposes of interpretation.

[10] (on which [12] is based) and the presentation in these latter two papers formed the beginning of the work in this thesis.

In addition to the extensions and modifications discussed below — which were contributed solely by the author — there is a major difference of presentation between the work given here and its predecessors. Previously, the treatment was confined to the failures divergences model, whereas here we deal individually with each of the three semantic models. In addition, the motivation behind this earlier work was to develop an implementation relation which could be used to relate an implementation component to its corresponding specification component in the event that the communications of the latter had been implemented in the former using some form of fault tolerance, possibly replication. In fact, considering the correctness of replicated processes was the main motivation behind [39]. The conception of the work was therefore far less general than it is here.

Mapping refusals and finally visible events The implementation relation given in [12] was lacking in one important property: it effectively failed to meet the condition that $\lambda(\phi Q) = \phi Q$ when $\alpha Q \subseteq Fvis$. This problem was solved immediately due to condition SFS2 from figure 3.6 and this constitutes one of the most important contributions made by the theory from the previous chapter. Previously, the mapping of refusal/trace pairs at the level of individual extraction patterns was carried out in the same manner whatever the implementation alphabet of the extraction pattern under consideration. More specifically, there was no equivalent of condition EP5-FVI from figure 4.7. Although given in a different form to EP5-FVI, this modification to the presentation in [12] first appeared in [13] and [16].

An alternative means of mapping refusals We consider here the issue of mapping refusal/trace pairs at the level of individual extraction patterns and consider an arbitrary extraction pattern ep_i. (Note that the following detail is simply sketching out a possible approach and is not intended to be a fully realised and formal presentation.) Although condition SFS6 from figure 3.6 deals only with refusals which are maximal in a certain sense, the means of mapping refusals given in definition 4.10 does not impose this restriction. If we remove from SFS6 the condition relating to maximality and translate to the notation used in this chapter, then we are given the following:

Let $t \in Dom_i$, $R, S \subseteq A_i$.Then $extr_i^{ref}(R \cup S, t) = extr_i^{ref}(R, t) \cup extr_i^{ref}(S, t)$.

This immediately suggests the following way of mapping refusal/trace pairs in practice, where $t \in Dom_i$ and $R \subseteq A_i$:

$$extr_i^{ref}(R,t) \triangleq \bigcup_{a \in R} extr_i^{ref}(\{a\},t).$$

Once we have adopted this compositional definition, SFS6 will be met however we proceed. Condition SFS2, requiring that $extr_i^{ref}(R,t) = R$ if $A_i \subseteq Fvis$, can be met easily by simply requiring that

$$extr_i^{ref}(\{a\},t) \triangleq \{a\} \text{ for } a \in A_i \subseteq Fvis.$$

The question remains, therefore, of what $extr_i^{ref}(\{a\},t)$ should return in the general case. Before presenting a possibility, we define the set $Rf_i(t)$, which gives all those events from B_i which cannot extend $extr_i(t)$ when we apply $extr_i$ to any $t \circ x \in Dom_i$. For $t \in Dom_i$,

$$Rf_i(t) \triangleq \{b \in B_i \mid (\nexists x \in A_i^*)\ t \circ x \in Dom_i \ \wedge \ extr_i(t) \circ \langle b \rangle \leq extr_i(t \circ x)\}.$$

As a counterpart to SF4, we first require that $Rf_i(t) \subseteq extr_i^{ref}(R,t)$ for any $R \subseteq A_i$: in other words, events which are impossible after the extraction of t must be contained in the extracted refusals. Then, for $t \circ \langle a \rangle \in Dom_i$, it is possible to relate a to the high-level event or events which it is being used to implement when it occurs after t. $extr_i^{ref}(\{a\},t)$ would return this set of high-level events along with $Rf_i(t)$. For $a \in A_i$ such that $t \circ \langle a \rangle \notin Dom_i$, $extr_i^{ref}(\{a\},t)$ would simply return $Rf_i(t)$. In terms of the running example from figure 1.1, we would take $extr_{ack}^{ref}(\{data.j\},\langle\rangle)$ to be $\{send.j\}$ for $j \in \{0,1\}$. This is because $data.j$ is being used to implement $send.j$ when it occurs after $\langle\rangle$ (of course, it is used to implement $send.j$ whenever it occurs). Moreover, $Rf_{ack}^{\langle\rangle}$ would be empty.

It is immediate that such an approach would meet SFS3, which requires that $extr_i^{ref}(R,t) \subseteq B_i$ and it is also immediate that SFS5, requiring the monotonicity of the mapping, would be met. It is also easy to show that SFS4 is met; that is, that $extr_i^{ref}(A_i,t) = B_i$. In order to show this, by SFS3 it is only necessary to show that $B_i \subseteq extr_i^{ref}(A_i,t)$. For $b \in B_i \cap Rf_i(t)$, the proof is immediate. For $b \in B_i - Rf_i(t)$, there must be an event $a \in A_i$ such that $t \circ \langle a \rangle \in Dom_i$ and a is used to implement b. Thus, $b \in extr_i^{ref}(A_i,t)$ in this case as well.

The intuition behind this possible approach is as follows: if we can refuse a after t at the implementation level, then we may be unable to offer, after $extr_i(t)$ at the specification level, any b which this occurrence of a is being used to implement. Whether this intuition makes sense in terms of practical verification is something that would need to be assessed on concrete examples. Note that refusal bounds would still need to be retained in order to deal with condition Dom-SF-check.

Dealing with divergence In [12] and [16], it is assumed that specification components are divergence-free and that they will be composed in such a way that the resulting specification network will also be divergence-free. Moreover, one of the conditions imposed on any implementation component Q is that no trace from $\tau_{Dom_{EP(Q)}}Q$ should also be a member of δQ. Finally, the condition is imposed that applying the extraction mapping over traces to an ω-sequence from $\tau_{Dom_{EP(Q)}}Q$ should return another ω-sequence. This guarantees that divergence will not be introduced on composition of the implementation components since it is not introduced when the corresponding specification components are composed.

The move from this treatment to that used here, which allows us to deal smoothly with divergences, was again indicated by the theory in the previous chapter, specifically condition FDS2 from figure 3.8 and the detail from section 3.6. Once we have removed the restriction that implementation and specification networks/component processes must be divergence-free and have applied the extraction mapping to (minimally) divergent traces, the original condition on ω-sequences is actually sufficient to give us what we need, since it is effectively equivalent to FDS2. Nonetheless, the move to imposing the condition on individual extraction patterns rather than on component implementation processes is important: it saves on verification effort by making the check part of the construction of the extraction pattern rather than part of the verification of the process. Moreover, the role played by minimally divergent traces — which do not appear in earlier work — in the proofs of results from this chapter is crucial to extending refinement-after-hiding to the failures divergences model.

Role of implementation alphabet That events occur on channels was an explicit part of the presentation in [12] and [16] and was integral to the notion of refinement-after-hiding presented there. Channels in specification components were partitioned into input and output channels and the *specification* alphabet of any particular extraction pattern could only contain events from input channels *or* events from output channels but not both. This led to two main differences to the treatment given here. Firstly, it was not necessary to introduce communication capabilities or the fact that composition over a particular implementation alphabet could only occur if one participant was denoted as *Left* and the other as *Right*; composition was controlled implicitly, at the specification level, by the fact that input channels may only be connected to output channels and vice versa. In addition, the extraction pattern component Θ was not used: an extraction pattern dealing with input events allowed deviation from the domain on any of the events in its implementation alphabet, while an extraction pattern dealing with output events did not allow any deviation at all.

The decision was made to abandon this treatment for two connected reasons, one more theoretical and the other practical. Condition TS4 from figure 3.4 was present in the predecessors of the work in this chapter (it is given here as TR-GLOBAL2 in figure 4.3). However, its appearance in the theory as a necessary condition served to highlight with a certain amount of clarity the role played by the implementation alphabets of the extraction patterns used in any particular verification (represented in the previous chapter by the sets contained in *MinSet*). Assume that Q is an implementation process, $t \circ \langle a \rangle \in Dom_{EP(Q)}$ and $a \in A_i$ for $ep_i \in EP(Q)$. By TR-GLOBAL2, we define $extr_{EP(Q)}(t \circ \langle a \rangle)$ in terms of $extr_{EP(Q)}(t)$ and $extr_i((t \circ \langle a \rangle)\lceil A_i)$. In a sense, as discussed in section 3.4, $extr_i((t \circ \langle a \rangle)\lceil A_i)$ is a function from a trace, $t\lceil A_i$, and an event, a, to the trace extension u which is used in the statement of TR-GLOBAL2. This means that A_i tells us what we need to know of t in order to determine the additional information which is given to us by the occurrence of a after t.

As a result, the aforementioned partitioning of extraction patterns only makes sense if we can always interpret a when it is used to implement input events by only knowing about the other events contained in t which are also used to implement input events, and similarly for output events. In the general case, however, this is not true and a specific example arose during the verification of the asynchronous communication mechanism described in chapter 7. At the specification level, the mechanism engages in read (output) and write (input) events: however, in order to interpret events used to implement a write it is necessary to know about the read behaviour of the mechanism so far and similarly for events used to implement a read. Using the approach from [12] and [16], it was therefore not possible to successfully verify the mechanism.

Equivalence Although the detail does not appear in the conference paper [12], [10] defines additional extraction pattern components and extra conditions which can be used to prove "equivalence" of an implementation and specification network.[28] An additional trace mapping, *inv* — a partial inverse of the extraction mapping — is provided to relate abstract traces to concrete traces. It is required that it is a trace homomorphism: in other words, $inv(\langle a_1, \ldots, a_n \rangle) = inv(\langle a_1 \rangle) \circ \ldots \circ inv(\langle a_n \rangle)$. The restriction is placed on specification components that after any trace, on any input channel, either all events must be refused or all events must be offered. Finally, an additional condition relating to the mapping of refusals from the specification is also imposed. Together, these last two conditions amount to using the standard

[28]The equivalence which may be proved is not standard CSP equivalence because of issues relating to mapping refusals, as detailed above.

notion of refusal bounds but with a fixed set of bounds to be applied regardless of the particular specification process under consideration. It became clear in deriving the conditions in chapter 3 that there were no theoretical reasons for treating the verification of specification against implementation differently from that of implementation against specification. Moreover, the requirement on *inv* that it be a homomorphism would be too restrictive if we were to deal with equivalence during the verification presented in chapter 7: there, communication events in the specification are implemented in different ways depending on the (specification) events which have preceded them. In other words, the full generality of a mapping from traces to traces is needed.

4.8.1 General role of the theory

We have highlighted above a number of areas in which the notion of refinement-after-hiding presented here differs from earlier versions, which changes were all introduced under the influence of the work in the previous chapter. In general, where changes were not suggested, the conditions derived in chapter 3 were useful in that they confirmed the soundness of earlier work and clarified our understanding of that work. For example, the results on mapping operators by simply mapping the sets with which they are parameterised, and on the way in which the mapping is to be applied to such sets, were very important. They confirmed that the approach taken in practice in earlier work was correct and not subject to alteration. Where the change required was not so great, such as in the case of divergences, the impact of the theoretical treatment should not be underestimated since it pinpointed exactly the approach to be taken and saved time and effort in the search for possible alternative ways of proceeding.

The issue of mapping refusals perhaps illustrates most fully the role which the theory can play. It highlighted immediately a solution to an earlier problem. It gave a framework for analysis within which the current use of refusal bounds in the mapping of refusal/trace pairs could be easily assessed and understood. Finally, the theoretical treatment suggested a new approach to the mapping of refusals and gave, in conditions SFS2-6, a straightforward way of assessing the soundness of any new method chosen.

Chapter 5

Related work

When behaviour decomposition and relaxation of atomicity are referred to in what follows, it is assumed we mean the *external* forms of these types of reification, since the internal versions can always be dealt with quite straightforwardly (at least in CSP).

5.1 Action refinement and related approaches

Action refinement (see [25] for a survey of this approach) is one of the main approaches which allows us to perform behaviour decomposition in a process algebraic setting and a great deal of work has been done in this area. In general, each action in the specification process under consideration is replaced by a precise, low-level behaviour which is defined by a *refinement function*. In this respect, it is related to the idea of top-down design in sequential systems, where high-level instructions are expanded into a lower-level module until the result is an implementation that may be executed. Action refinement is primarily concerned, therefore, with the derivation of an (correct) implementation and not with verification after the fact, in contrast to our notion of refinement-after-hiding. It suffers from two problems, however. Only a single implementation is possible for any particular specification and refinement function pair. Moreover, causal relations between events in the specification must be preserved in the implementation. For example, if a precedes b in the specification, then all of the events which form the implementation of a must precede in the implementation all of those events which form the implementation of b. Some forms of action refinement are able to perform relaxation of atomicity to the extent that, if a and b occur concurrently in the specification, then their respective implementations may interleave. However, this is the limit of the relaxation of atomicity which is possible.

[32] and [60, 71] use a dependency relation and a *weak* form of sequential

composition in order to allow additional relaxation of atomicity in an action refinement framework. Any action of the first argument to the new composition operator must precede all actions from the second argument with which it is dependent. However, an action from the second argument may precede one from the first provided that the two actions are independent. Nonetheless, there is still only one possible implementation for a particular specification (for any given refinement function and dependency relation).

In order to deal with these twin problems of action refinement — only a single implementation possible and only restricted relaxation of atomicity allowed — Rensink and Gorrieri presented the work in [59] (earlier versions appeared as [57] and [58]; an alternative presentation of some of the material appeared in [25]). The paper [59] works in the process algebraic context and details a notion of refinement termed *vertical implementation*, which may be used to verify correctness when behaviour decomposition has occurred in the move from specification to implementation, along with a degree of relaxation of atomicity. We first list some of the aims behind this work, taken from [59], to illustrate the fact that it is similar in concept to ours.

- the vertical implementation relation is parametric with respect to a mapping.
- flexibility: multiple implementations are allowed for a given specification and a strict ordering is not dictated for the low level actions implementing a high level one;
- simplicity: the introduction of a concurrency model more complex than any of the standard interleaving ones is not required;
- the vertical implementation relation "collapses" to a standard notion of refinement when the mapping is the identity;
- deadlock properties carry over from the abstract to the concrete level;
- compositional verification is allowed in the same sense as our approach.

The fact that the vertical implementation relation "collapses" to a standard notion of refinement when the mapping is the identity, along with the way in which the operators used interact with the refinement mapping, means that the notion of vertical implementation is essentially a form of refinement-after-hiding, although it is not presented in such terms. However, there are important differences in terms of concept and also in terms of technical execution which exist between the work in this thesis and that in [59].

The motivation behind our work was to explore a means of generalising refinement in CSP in order that both behaviour decomposition and relaxation of atomicity could be accommodated in the general case. Rensink and

Gorrieri start from the premise that *action refinement* has the shortcomings described above and seek to remedy them. Inherent in their treatment, therefore, is the restriction that the vertical implementation relation is to be parameterized with an *action refinement* mapping. This means that the domain of the mapping contains only individual (specification) events, although each event may be mapped to a process. Although we map to individual behaviours rather than processes, there do seem benefits to be gained from using our more general type of mapping. In particular, we can (attempt to) verify correctness in the event that relaxation of atomicity has occurred *without* behaviour decomposition: it is not obvious how the same thing could be done using an action refinement mapping. Moreover, in the general case, it need not be true that the same high-level action is always implemented in the same way wherever it occurs: this certainly applies to the asynchronous communication mechanism whose verification is considered in chapter 7. This suggests the need for an abstraction mapping and one which takes account of the history of any particular event to which it is applied. As a general point, if the aim is to allow multiple implementations for a single specification then using an abstraction mapping from possible implementations to specifications seems a more sensible way to proceed than using a mapping to make behaviours more concrete.

Where our treatment is based around a *semantic* mapping, Rensink and Gorrieri provide as a foundation of their notion of vertical implementation a set of (*syntactic*) proof rules. If a concrete relation is to be viewed as a valid vertical implementation relation, then these rules must be sound with respect to that relation. As a result, the rules define both what it means to be a vertical implementation relation and also give a proof system for any such concrete relation: this latter is something we are lacking with respect to refinement-after-hiding. Many of these rules play a similar role to the conditions RAH1-3 from chapter 3, although they are given with respect to the full range of operators. In addition, a rule is given on relaxing causality when refining actions (in addition to the interleaving which is allowed when the events being implemented are independent at the specification level).

In [59], communication is treated symmetrically: both parties to a communication are required to offer *all* of the relevant events used to implement a particular high-level event. However, [25] does introduce a version of vertical implementation whereby one party to a communication must offer all relevant events, while the other party need only offer *some* of those events. Such a treatment is similar to the use of refusal bounds in our concrete notion of refinement-after-hiding, although it is not quite as general as using (the equivalent of) refusal bounds: it seems similar to disallowing the refusal of any events on one side of the communication and simply requiring the offering of one of the possible events on the other.

Similar conditions on readiness to communicate are also a feature of the paper [56]. Using a notion of abstraction, it sets out to answer the question of when a set of fine-grain execution steps may be contracted into an abstract atomic action and so aims to explore notions of correctness which are useful when considering relaxation of atomicity. It does this in a restricted setting, considering the interactions between a client or *agent* and a server: the client may invoke a method of the server, to which invocation the server will (eventually) respond. The aim is for the relevant notion of "implements" to hold whenever the client cannot see the difference between the "atomic" and the "non-atomic" servers: that is, whenever the result of composing the respective servers with the same client is two "equivalent" systems. This is similar to our initial characterisation of refinement-after-hiding.

Of most interest, however, is the operational characterisation of the desired notion of implementation, which is based on *coupled simulation* ([52, 53]). The choice of such a basis on which to build the notion of abstraction is justified in the following terms. Intuitively, if the concrete state s "implements" the abstract state s', standard bisimulation relations such as weak bisimulation might also require that s' "implements" s. The use of coupled simulation allows that the abstract s' need only implement the concrete s when behaviour at s is *complete* in a sense similar to that used in our notion of refinement-after-hiding. This feature of the relation is not needed for soundness, rather it makes it weaker and allows more systems to be verified as correct. It seems that it may fulfil a similar requirement to the use of condition Dom-SF-Check in our work; in any case, it is an issue which invites further exploration. In particular, it may have implications for any future attempt to transfer the work presented here to a bisimulation setting.

5.2 Choosing a semantic over a syntactic mapping

In this thesis, we have chosen to use a semantic rather than a syntactic mapping. Here, we consider reasons behind this choice, with some reference to the discussion above on action refinement and vertical implementation.

Were we to use a *syntactic* mapping, it would effectively have to be defined over individual events due to the difficulty of defining it directly over arbitrary processes. As a result, using a syntactic mapping would mean using a refinement mapping to make behaviours more concrete. However, as discussed above in relation to the notion of vertical implementation, it seems more sensible to (be able to) use an abstraction mapping if we wish to allow multiple implementations for a single specification. Moreover, also as mentioned above, defining a mapping over individual events means that each specification event must be implemented in the same way whenever it occurs. This need not be true in general and is certainly not the case with regard to the asynchronous communication mechanism whose verification is considered in chapter 7. Of course, we could deal with this problem by annotating different instances of the same event and having events with different annotations implemented in different ways: however, determining how a particular event should be annotated — at least in the case of the asynchronous communication mechanism from chapter 7 — would rely on knowledge of the history of the process upto its occurrence. This would effectively require a mapping over sequences of events rather than over individual events and so would require a semantic mapping. Moreover, the same syntactic occurrence of an event may have different histories on different executions and so may be implemented in different ways in different executions: thus, it may not be possible to give a particular syntactic event a unique annotation.

5.3 External behaviour decomposition (interface refinement)

Two papers which explicitly set out to deal with the issue of interface refinement are [7] and [24].

The approach followed in [7] is termed there "interface displacement". In this approach, the aim is to take the abstract interface of two specification components, P and Q, and transform it into a more concrete one. The interface change is then encoded in a process I: I is composed in parallel with P on the set of abstract interface actions and these actions are then hidden to give the implementation process P', with the more concrete interface. In order to give to Q the concrete interface, the process I is "subtracted" from Q in a sense, giving the implementation process Q'. Q' is such that, if I were composed in parallel with it on the *concrete* actions of the (new) interface and those concrete actions hidden, then the result would be the original process Q. It is in this sense that the interface is "displaced". [7] therefore gives a means of carrying out the process of interface refinement rather than verifying its correctness after the fact. There is an interesting similarity with predecessors of the work in this thesis: namely, the interface transducers play a role comparable to that of the disturbers and extractors of [39], although there is no notion of "displacement" in [39] and all interfaces are treated in a uniform manner.

There are, however, a number of differences between the approach presented here and that in [7]. The latter is focused on the refinement of a (specification) *compound* system, where the components interact through a well-defined interface, into a compound implementation system, where that interface has been refined. This interface refinement is then justified on the grounds that the compound implementation system is guaranteed to be a correct implementation of the specification system in terms of a standard implementation relation such as traces or failures inclusion. There is no notion of being able to relate *individual* implementation and specification components — refinement of a particular process interface must be done in tandem with that of its environment — and so no notion of any sort of vertical implementation relation in the terminology of [59].

The paper [24] gives an initial formulation of a notion of interface refinement which is similar in outline to our notion of refinement-after-hiding. Working in a temporal logic framework, low-level (infinite) behaviours are related to high-level (infinite) behaviours using a device which is similar to our extraction mapping. Moreover, according to the examples discussed, it seems that the method has the power to deal simultaneously with both behaviour decomposition and relaxation of atomicity. However, it lacks the power of compositionality: that is, it is not shown that any operators on processes are monotonic with respect to this relation. This means that, if the interface between two components composed in parallel is to be refined, it is necessary to verify the *composed* implementation against the *composed* specification (the interface actions are not hidden in this composition).

It is the avowed intention of the authors in [24] to define a notion of interface refinement which is as unrestricted as possible. The result of this appears to be, however, that it is not clear what it actually means that one system implements another nor what properties are preserved from specification to implementation. Lack of compositionality may be crucial in this respect: for example, in our treatment, compositionality (conditions RAH2 and RAH3) and the condition RAH1 (expressing a kind of "collapse" property) are the criteria against which the validity of any notion of refinement-after-hiding is judged and they essentially invest such notions with meaning.

5.4 Abstraction through hiding

Looking at the general issue of behaviour abstraction, some approaches (for example, that described in [1]) describe system behaviour by sequences of state tuples with an internal component; they then require that, for every possible state sequence of a correct implementation, there should exist one of the specification such that the two sequences coincide after deleting the internal state component. A similar treatment is presented in [38], using infinite action sequences (i.e. infinite traces) instead of state sequences: the interface of the specification must be a subset of the interface of the implementation and it is required that every trace of the implementation can be turned into one of the specification by deleting actions not in the specification's interface. These two approaches and other comparable ones, such as [47], are based on abstraction by ***hiding***. In contrast, our notion of abstraction is essentially based on the ***interpretation*** of behaviours over a particular alphabet as behaviours over another alphabet. As illustrated by the example given in section 1.4.1 in chapter 1, abstraction by interpretation may not be reduced to abstraction by hiding in the general case. Nonetheless, abstraction through hiding is certainly useful and is comparable to our work to the extent that we expect to eventually hide any interpreted behaviour, leaving implementation and specification systems which may engage in the same set of visible events. However, refinement-after-hiding gives a means of verifying that abstraction through hiding will give correct behaviour *before* we actually compose the components of a system and hide the necessary behaviour: in this sense, it gives a compositional means of verifying a notion of abstraction through hiding.

5.5 Relaxation of atomicity

Some of the action refinement-related papers already discussed — such as [56], [59], [60] and [71] — have some capacity to deal with the issue of relaxation of atomicity, as did the work presented in [24]. Here we consider other work where managing relaxation of atomicity is/was the principal aim.

One of the major areas of work where this issue has been addressed is that of databases and transaction-processing systems. In this area, a specification is given in terms of a set of *sequential* behaviours: this generally means that the transactions occurring in a particular specification behaviour are totally ordered and their executions do not overlap.[1] An implementation is then allowed to execute (parts of) some transactions in parallel, provided that the resulting parallel behaviour is equivalent in some sense to an acceptable sequential behaviour. In general, this notion of "equivalence" consists of two factors: the first is that, using some suitable dependency relation, the events of the implementation behaviour under consideration may be transformed into a (sequential) specification behaviour by commuting events regarded as independent; secondly, the result of executing the implementation behaviour is the "same" as the result of executing the corresponding specification behaviour. This latter point is generally framed as the requirement that an observer would be unable to distinguish the implementation behaviour from the specificiation behaviour. (This notion of observer may not always be

[1]Note, however, that a specification will not usually be given explicitly in the database domain and "specification" behaviours are usually those sequential behaviours which are possible for the implementation.

treated explicitly, however.) Relevant notions from the literature include *serializability* (see, for example, [2]) and the notion of "atomicity" defined in [48]. Related notions in the more general context of managing parallel access to shared memory in a multi-processor architecture are *sequential consistency* ([42]) and *linearizability* ([29]). Sequential consistency requires that the dependency relation used to commute implementation events to give sequential behaviours preserves the ordering of events which occur in the same process; linearizability requires in addition that we preserve the global (external) ordering of non-overlapping operations.

Similar to an extent is the work by Lamport in [43], where executions of a low-level (non-atomic) system and those of a high-level (atomic) system are related on the basis of orderings which exist between events at a particular level of abstraction. Although Lamport's work effectively allows for the possibility of moving across levels of abstraction — we may contract a set of low-level actions into a single high-level action — that of the others does not: in general, sequential processes engage in the same events as non-atomic processes. It is difficult to compare directly our notion of refinement-after-hiding and the work discussed in this section, not least because dependencies between events do not form part of our treatment. Nonetheless, an interesting avenue for future work might be to explore possible relationships between refinement-after-hiding and such dependency-based notions of correctness: for example, since dependencies between events play such a fundamental role in the move from sequential to parallel behaviours, how do they affect our ability to derive an extraction mapping which relates those behaviours?

As indicated in chapter 1, Dingel's thesis ([21]) uses a notion of correctness-in-context based on the rely-guarantee approach as he develops a refinement calculus to be used in the derivation of parallel programs. This work obviously allows for the possibility of relaxation of atomicity. However, this relaxation is not permitted to manifest itself at the *semantic* level once those behaviours have been ignored which fail to meet the relevant *rely* condition. In other words, the behaviours of the implementation which will be preserved after it is placed in context can be compared directly with those of the specification and no alternative notion of refinement is needed.

The papers [34, 35, 37] introduce transformations on objects that increase concurrency: these increases in concurrency are possible due to restrictions imposed on the visibility of references to object instances, which restrictions limit the extent of any interference which may occur. The papers also essentially raise the challenge of proving the validity of these transformations, where a particular transformation may be regarded as "valid" if no context can tell the difference between the original sequential object and the "concurrent", transformed object.[2] [36] presents informal reasoning in support of

[2]The contexts under consideration are limited by the properties of the language in

the validity of some of these transformations, using arguments in terms of the *π-calculus* (see [67] for further details on the π-calculus).

In [66], Sangiorgi proves the correctness of one of these transformations using a typed version of the π-calculus and a typed notion of behavioural equivalence. The type discipline is that of *uniform receptiveness* (see [66] for a brief description), where the receptiveness part of the condition is similar to the input-enabledness property of the i/o automata in [47]. It is likely that the property of receptiveness plays a similar role in preserving liveness as do our conditions involving the consideration of refusal bounds, such as Dom-SF-Check.

However, the most interesting part of Sangiorgi's treatment is the use of *barbed congruence* ([51]) as the notion of behavioural equivalence. The idea behind this is to equip an observer with a *minimal* ability to observe actions and/or process states, which ability induces an equivalence relation between processes. This equivalence relation induces in turn a congruence, namely equivalence in all contexts. In the notion of barbed bisimulation, an observer can observe invisible transitions and the state they lead to; he/she can also detect when a process offers a visible event but cannot see its identity or the state to which it takes us. Barbed bisimulation is then used to induce barbed congruence and it is shown in [51] that this congruence is equivalent to strong bisimulation. What is most interesting, however, is that barbed congruence is *strictly weaker* than strong bisimulation if the set of contexts into which a process may be placed is restricted; the same applies with respect to weak bisimulation and a weak notion of barbed congruence, the latter being used in [66].[3] For example, the sequential object and the transformed, concurrent object from [66] are not even trace-equivalent; however, they can be related using the typed notion of barbed congruence.

Our goal in this thesis was to develop a notion of behaviour refinement which related behaviours much more loosely than standard CSP refinement, by effectively reducing the set of events which could be directly observed of our processes (the "directly observable" events are those that are finally visible). This has the effect of giving a notion of refinement strictly weaker than standard CSP refinement but which implies refinement according to the standard semantics after processes are composed to form a suitable network. Moreover, the contexts into which processes can be placed when using the notion of refinement-after-hiding are also restricted, which is crucial to the success of the scheme. In the restriction of what is visible and the restriction

which objects are described.

[3]In [66], Sangiorgi restricts contexts in a once-and-for-all way using a notion of typing; our contexts are restricted in the sense that Dom-T-check and Dom-SF-check must be met and an implementation context must be correct with respect to the corresponding specification context.

of acceptable contexts, typed notions of barbed congruence may be related conceptually to our notion of refinement-after-hiding, although they do not allow us to cross levels of abstraction and their development was motivated by totally separate concerns. Further work is needed to explore in more detail the relationship between the two approaches — which itself may necessitate a transference of ours to a bisimulation setting — but it may be that aspects of the barbed congruence approach can be put to good use in developing an "improved" notion of refinement-after-hiding. Some areas for future work are:

- Exploring ways in which barbed bisimulation could be adapted so that it may relate implementation and specification processes where the former has been derived from the latter using behaviour decomposition.

- Verification for typed notions of barbed congruence is carried out using laws stating equivalences between *sytactic* terms. Might it be possible to characterise typed barbed congruence in the manner of standard bisimulation, which would allow for automatic verification? Might some notion of explicit mapping be needed then to relate implementation and specification behaviours, as in the presentation of our notion of refinement-after-hiding?

- How weak can we make the restrictions on contexts and still use barbed congruence when relaxation of atomicity has occurred? How well will barbed congruence transfer to the general case of dealing with relaxation of atomicity, instead of just dealing with the specific example in [66]?

- In our approach, what it means for one process to implement another is based firmly on the standard notion of refinement in CSP, due to condition RAH1. Once contexts have been restricted and the relation given by barbed congruence is weaker than standard equivalences, what does it actually mean that one process implements another according to that relation? Might it be the case that, if all "finally invisible" events are hidden due to implementation and specification processes being placed in context, then we reclaim standard (weak) bisimulation or similar?

5.6 Data reification in Z

Z ([70]) is a specification language which considers each specification to define an abstract data type (ADT). That an abstract ADT $\mathcal{A}$ is refined by a more concrete ADT $\mathcal{C}$ is defined in terms of an input/output relation: $\mathcal{C}$

refines $\mathcal{A}$ if, for every sequence of inputs, the outputs produced by $\mathcal{C}$ after executing a particular sequence of operations are a subset of those possible for $\mathcal{A}$ after executing the sequence of corresponding operations. Refinement is therefore quantified over all possible programs (sequences of operations); its verification is made tractable using the method of *simulation* (see [22]) and it is in this sense that it may be related to process algebraic refinement. Standard data refinement in Z requires that input and output must be the same in both specification and implementation; moreover, there should be a one-to-one correspondence between abstract and concrete operations. The papers [3, 4, 19, 20] detail how these restrictions may be relaxed. It is shown how inputs and outputs may be refined so that a concrete ADT may input and output data using a different representation of it to the corresponding abstract ADT[4]; moreover, it is shown how an abstract operation may be implemented using a *sequence* of concrete operations, in a manner similar to action refinement. The setting in which this is carried out is sequential: because we are interested only in the input/output relation induced by programs, it is easy to justify splitting at the concrete level the operations which consitute those programs, as it is to justify the transformation of data in some way on the first and final steps of those (concrete) programs. In [5], results are given on representing CSP failures divergences refinement in a Z relational setting, one of the aims being to allow the use of Z refinement in a concurrent framework. It would be interesting to see the extent to which the results from [3, 4, 19, 20] might also be transferred to such a setting.

[4]The I-O transformers used for this purpose play a similar role to the extractors and disturbers from [39].

Chapter 6

Automatic verification

This chapter considers two different means of automatic verification of the notion of refinement-after-hiding presented in chapter 4. Firstly, we consider (briefly) algorithms for this purpose. Secondly, and at greater length, we consider the use of the tool FDR2 (see, for example, [63] or [64]).

6.0.1 Algorithms for automatic verification

Initial effort to verify automatically our notion of refinement-after-hiding was focused on the development of *algorithms* for this verification. Such algorithms were published in [11], referring to the implementation relation which appears in [12]. The paper [16] contains an updated set of algorithms to verify the notion of refinement-after-hiding which it describes. In these papers, both processes and extraction patterns are represented as (variants of) labelled transition systems.[1] The various components of the implementation relation are then given what is effectively a graph-theoretic restatement. For example, during the verification that $extr_{EP(Q)}(\tau Q) \subseteq \tau P$, we extract the traces of Q by taking the product of the transition system representing Q itself and those representing the necessary extraction mappings. That the extracted traces of Q are contained in those of P can then be verified using a standard algorithm to check for trace containment. (Further details can be found in [16].)

However, rather than provide an implementation of these algorithms (at least in the short term), the decision was taken to develop an alternative means of verification using FDR2. This was for a number of reasons. Developing a usable and efficient tool is a costly undertaking in terms of time and would likely have been beyond the scope of this thesis. Having been in development for ten years or more, FDR2 is mature, bug-free (in the

[1]The representations of processes are similar to those employed in FDR2, these being described in [62] for example.

author's experience) and has a number of means built-in of improving the space-efficiency of the tool. These have been of use when carrying out the verification detailed in chapter 7. Perhaps more importantly, FDR2 takes as input text files containing CSP process expressions; an implementation of the algorithms from [16] would have required the implementation of a CSP parser and compiler. Finally, the means of verification detailed below has allowed a rapid move to employing our notion of refinement-after-hiding in the verification of a real-world example: as mentioned in section 4.8 with respect to the role of implementation alphabets, this provided feedback for the development of that notion.[2]

6.0.2 Verification using FDR2

Since Dom_i for $ep_i \in EP$ is simply a set of traces, one of the most obvious ways to define that set of traces is using the CSP language itself: that is, we define a process to represent Dom_i. It turns out that it is also possible to encode the extraction mapping over traces as a CSP context. The application of the mapping to refusals cannot be handled so cleanly and some modifications need to be made to the specification under consideration. However, once these modifications have been made, we are able to encode the mapping of refusals as a CSP context. This enables us to present the checking of our notion of refinement-after-hiding in the various semantic models as a series of standard CSP refinement checks, which in turn allows us to use FDR2 for the purposes of automatic verification.[3]

Note, however, that the means of verification presented in this chapter may only be used under a certain restriction. For any implementation process, Q, where $t \circ \langle a \rangle \in Dom_{EP(Q)}$, it must be the case that:

$$|extr_{EP(Q)}(t \circ \langle a \rangle)| \;\leq\; |extr_{EP(Q)}(t)| + 1.$$

In other words, the occurrence of a further implementation event can cause the occurrence of at most one extra specification event after extraction. The extraction mapping used in chapter 7 violates this restriction in a single case and it is explained there how the problem may be addressed for that case. Note also that this restriction precludes the use of the approach described in this chapter to verify a *specification* against its *implementation*, since the occurrence of any individual specification event will usually lead to the extraction of more than one implementation event.

[2]The report [14] contains a preliminary version of the work in this chapter and this was used to verify the notion of refinement-after-hiding from [16].

[3]In the failures divergences model, we check only the restricted case that the relevant specification *component* process is divergence-free. See section 6.7 for more details.

6.1 Preliminary detail

In the remainder of this chapter, we assume the existence of a fixed implementation process, Q, and a fixed specification process P, against which Q is to be verified.

6.1.1 Useful notation

The following notation will prove to be useful.

Definition 6.1. $inv \triangleq \{i \mid ep_i \in EP(Q) \;\wedge\; A_i \cap Fvis = \varnothing\}$.

inv therefore gives the set of subscripts of those extraction patterns used to interpret behaviours over finally *invisible* events.[4] By EP1, $i \notin inv$ will be used as shorthand for the fact that $ep_i \in EP(Q)$ and $A_i \subseteq Fvis$. The extraction pattern components A, B, *Dom* and *dom* may then be lifted to the set *inv*.

Definition 6.2. *The following hold by definition:*

1. $A_{inv} \triangleq \bigcup_{i \in inv} A_i$.
2. $B_{inv} \triangleq \bigcup_{i \in inv} B_i$.

The following definition is a counterpart in terms of *inv* to conditions TR-GLOBAL1 and SF-GLOBAL1 in chapter 4.

Definition 6.3. *The following hold by definition:*

1. Dom_{inv} *is the set of* $t \in (A_{inv})^*$ *such that* $t \lceil A_i \in Dom_i$ *for* $i \in inv$.
2. dom_{inv} *is the set of* $t \in (A_{inv})^*$ *such that* $t \lceil A_i \in dom_i$ *for* $i \in inv$.

The mapping $Next_i : Dom_i \rightarrow 2^{A_i}$ is defined for every $i \in inv$ and gives the possible extensions to any trace $t \in Dom_i$ such that the resulting trace is still a member of Dom_i. It is defined as follows.

Definition 6.4. $Next_i(t) \triangleq \{a \mid t \circ \langle a \rangle \in Dom_i\}$.

Finally, $Next_i$ can be lifted to Dom_{inv} in the following way, where $Next_{inv} : Dom_{inv} \longrightarrow 2^{A_{inv}}$.

Definition 6.5. $Next_{inv}(t) \triangleq \bigcup_{i \in inv} Next_i(t \lceil A_i)$.

[4]It will not be necessary to construct a CSP representation for those extraction patterns which are used to "interpret" finally *visible* events.

6.1.2 Process alphabets

During the course of this chapter, we will present a number of different CSP processes to be used in the means of verification which is detailed here. In general, it will be necessary to define an alphabet, α, for each of these processes. We first note that αQ is as defined in definition 4.5 in section 4.1.4. For any other process, W, and by the detail in section 2.5, we are free to assign whatever alphabet we wish provided that $\beta(W) \subseteq \alpha W$. Before the alphabet of any process is used for any purpose, we will always state what we take it to be.

6.1.3 Recursive definitions

In this chapter, we define a number of different CSP processes which are parameterized by the traces in Dom_i. As an example we introduce the simplest of these processes, D_i for $i \in inv$, which is used in the next section:[5]

$$D_i \triangleq D_i(\langle\rangle) \text{ and } D_i(t) = \Box_{a \in Next_i(t)} a \to D_i(t \circ \langle a \rangle).$$

By definition 6.4 and since $\langle\rangle \in Dom_i$ by EP3-T, then $t \in Dom_i$ for any t such that $D_i(t)$ is a term used in the definition of D_i. Thus, $Next_i$ is defined whenever it is used in the definition of D_i. Moreover, it is easy to see that $\tau D_i = Dom_i$. However, since Dom_i may be an infinite set of traces in the general case, D_i is a process with a potentially infinite description. This is not a problem for the following reason. For such a definition, we assume the existence of a finite equivalence relation over the traces in Dom_i such that processes parameterized by equivalent traces have the same semantics.[6] In fact, in practice, a single distinct process name would usually be used to represent each set of processes parameterized by equivalent traces, with the result that two processes parameterized by equivalent traces will actually have the same *syntactic* definition. As a result, the definition of D_i used in practice in any verification would be finite: the representation given here is effectively the syntactic unfolding or unwinding of the definition used in practice. That the semantics of the finitely-represented process and its unwinding D_i are the same stems from the detail given in section 2.4.5 (note that D_i is guarded). Similar comments apply to the other processes which are defined in this chapter.

[5]Note that we use $\triangleq$ when we wish to assign a label to a particular syntactic term for ease of reference; as such, $\triangleq$ should not be regarded as an operator of the language under consideration and is not to be confused with the recursion operator, $=$.

[6]In the case of D_i, the equivalence relation could be induced by the nature of the (necessarily finite) description of Dom_i: see, for example, the definition of Dom_{ack} in section 4.1.7.

For example, where $ep_1 = ep_{ack}$, the process D_1 would be used in practice to represent Dom_{ack} defined in section 4.1.7:

$$D_1 = \Box_{x \in \{0,1\}} data.x \rightarrow D_1(x)$$

where

$$D_1(x) = (ack.yes \rightarrow D_1) \Box (ack.no \rightarrow data.x \rightarrow D_1).$$

Sections 6.4.3, 6.5.4 and 6.6.4 define the other processes used to verify the running example: further insight into the points made here can be gained by relating those processes to the relevant generic process definitions given in sections 6.4, 6.5 and 6.6 respectively.

6.2 Verifying Dom-T-check

We first recall the definition of condition Dom-T-check from figure 4.4 before proceeding to show how it may be verified:

Dom-T-check If $t \lceil Proj_{EP(Q)} \in (Dom_{EP(Q)} \lceil Proj_{EP(Q)})$ for $t \in \tau Q$ then $t \in Dom_{EP(Q)}$.

Before proceeding, recall that $Proj_{EP(Q)}$ gives those events on which Q may move outside of the domain $Dom_{EP(Q)}$; it is defined in definition 4.7 in section 4.2. In order to verify Dom-T-check, it is necessary to define a process to encode Dom_i for each $i \in inv$, along with another process which gives the set of traces $t \in \tau Q$ such that $t \lceil Proj_{EP(Q)} \in (Dom_{EP(Q)} \lceil Proj_{EP(Q)})$. The first process is D_i, used in the previous section as an example, and defined as follows for $i \in inv$:

$$D_i \triangleq D_i(\langle\rangle) \text{ and } D_i(t) = \Box_{a \in Next_i(t)} a \rightarrow D_i(t \circ \langle a \rangle).$$

As mentioned above, it is easy to see that $\tau D_i = Dom_i$ for $i \in inv$. The second process we must define here is Q_{Proj}, in the definition of which the following auxiliary processes are used:

$$DC_i \triangleq D_i \setminus (A_i - Proj_i) \text{ for } i \in inv \text{ and } DC \triangleq |||_{i \in inv} DC_i.$$

τDC_i for $i \in inv$ gives $Dom_i \lceil Proj_i$[7] and τDC gives $Dom_{EP(Q)} \lceil Proj_{EP(Q)}$. Q_{Proj} is then defined as follows:

$$Q_{Proj} \triangleq Q \parallel_{Proj_{EP(Q)}} DC.$$

[7]Recall that $Proj_i$ gives the set of events on which a process may move outside the domain Dom_i. It is defined in definition 4.7 in section 4.2.

The following results give two different ways of verifying Dom-T-check using these processes.

Theorem 6.1. *$Q_{Proj} \setminus Fvis \sqsupseteq_T (|||_{i \in inv} D_i)$ if and only if Q meets Dom-T-check.*

Theorem 6.2. *$Q_{Proj} \setminus (\alpha Q - A_i) \sqsupseteq_T D_i$ for every $i \in inv$ if and only if Q meets Dom-T-check.*

The second of these results would most likely be used as the basis of any verification in practice because it avoids the need to construct the interleaving of the processes D_i, the state space of which could be quite large. However, the choice is left to the user and, for smaller $(|||_{i \in inv} D_i)$, the verification of Dom-T-check may be carried out using a single refinement check by virtue of theorem 6.1. (Note that, by proposition C.1 in appendix C and PA1, *Fvis* in the statement of theorem 6.1 could be replaced with $\alpha Q \cap Fvis = \bigcup_{i \notin inv} A_i$.)

By virtue of theorem 6.2 and using the definitions given above, process D_1 defined in section 6.1.3 to encode Dom_{ack} was used to successfully verify that both *LeftImpl* and *RightImpl* from the running example meet condition Dom-T-check.

6.2.1 Alternative means of constructing Q_{Proj}

Since $\beta(DC_i) \subseteq Proj_i$ for $i \in inv$ and $\beta(DC) \subseteq Proj_{EP(Q)}$ — see proof of proposition C.1 in appendix C — we may take $\alpha DC_i = Proj_i$ for $i \in inv$ and $\alpha DC = Proj_{EP(Q)}$. Hence, by EP2 and definitions 4.5 and 4.7, $\alpha DC_i \subseteq \alpha Q$ for $i \in inv$ and $\alpha DC \subseteq \alpha Q$. Moreover, for $i, j \in inv$ such that $i \neq j$, $\alpha DC_i \cap \alpha DC_j = \varnothing$ by EP-UNI1. As a result of this and the detail in section 2.6 on the associativity of parallel composition,

$$Q_{Proj} =_T (((Q \parallel_{Proj_{j_1}} DC_{j_1}) \parallel_{Proj_{j_2}} \ldots) \parallel_{Proj_{j_k}} DC_{j_k}),$$

where $inv = \{j_1, j_2 \ldots, j_k\}$. This latter syntactic construction would be used in practice because it avoids the need to create an intermediate process DC during verification in FDR2 which may be much larger than the final process Q_{Proj}. The original construction is used in the text because it eases the necessary proofs. Similar comments apply in the remainder of this chapter wherever processes representing a particular extraction pattern component — such as *extr* or *ref* — are interleaved before being composed in parallel with another process.

6.3 Preprocessing the implementation process

In this and subsequent sections in this chapter, we assume that Q has already been shown to meet condition Dom-T-check. During verification, we are interested only in behaviours (whose trace component is) from $Dom_{EP(Q)}$. Q is therefore preprocessed in order to remove all non-domain behaviours, thereby creating a new process $\widehat{Q}$. Although this preprocessing actually adds some new failures, rather than simply taking a subset of the failures of Q, it does so in such a way that the answers to the verification questions in which we are interested are the same for Q as they are for $\widehat{Q}$ (see theorem 6.5). In order to preprocess Q, it is composed in parallel with $|||_{i \in inv} D_i$, where D_i for $i \in inv$ is as defined in the previous section:

$$\widehat{Q} \triangleq Q \parallel_{A_{inv}} (|||_{i \in inv} D_i).$$

We observe that, for $i \in inv$, $\beta(D_i) \subseteq A_i$ by EP3-T and definition 6.4. By definitions 4.3 and 4.4, $\beta(Q) \subseteq \bigcup_{1 \leq i \leq m} A_i$ and so $\beta(Q) \subseteq \beta(\widehat{Q}) \subseteq \bigcup_{1 \leq i \leq m} A_i$. Thus, the following result holds by definitions 4.3, 4.4 and 4.5.

Proposition 6.3. *The following hold:*

1. *$\widehat{Q}$ is an implementation process.*
2. *$EP(\widehat{Q}) = EP(Q)$.*
3. *$\alpha\widehat{Q} = \alpha Q$.*

We also assume that $Comm(A_i, Q) = Comm(A_i, \widehat{Q})$ for $ep_i \in EP(Q)$. The following result then characterises the behaviours of $\widehat{Q}$.

Proposition 6.4. *The following hold of $\widehat{Q}$:*

1. $\tau\widehat{Q} = \tau_{Dom_{EP(Q)}} Q$.
2. $\phi\widehat{Q} = \{(t, S \cup U) \mid (t, S) \in \phi_{Dom_{EP(Q)}} Q \wedge U \subseteq (A_{inv} - Next_{inv}(t \lceil A_{inv}))\}$.
3. $min\delta\widehat{Q} = \{t \mid t \in min\delta Q \ \wedge \ t \in Dom_{EP(Q)}\}$.

Under the assumption that Q meets Dom-T-check, the following result allows us to verify that $Q \sqsupseteq_X^{EP(Q)} P$ for $X \in \{T, SF, FD\}$ by instead verifying that the necessary conditions from figures 4.5, 4.10 and 4.12 hold of $\widehat{Q}$. In view of this and proposition 6.4, we shall always refer in what follows to $\tau\widehat{Q}$, $\phi\widehat{Q}$ and $min\delta\widehat{Q}$ in lieu of $\tau_{Dom_{EP(Q)}} Q$, $\phi_{Dom_{EP(Q)}} Q$ and $min\delta Q \cap Dom_{EP(Q)}$ respectively. (It will still be necessary to consider directly $\phi_{dom_{EP(Q)}} \widehat{Q}$ however.)

Theorem 6.5. *The following hold:*

1. $extr_{EP(Q)}(\tau Q) = extr_{EP(Q)}(\tau\widehat{Q})$.
2. *Q meets Dom-SF-check if and only if $\widehat{Q}$ meets Dom-SF-check.*
3. $extr_{EP(Q)}(\phi Q) = extr_{EP(Q)}(\phi\widehat{Q})$.
4. $extr_{EP(Q)}(\delta Q) = extr_{EP(Q)}(\delta\widehat{Q})$.
5. $extr_{EP(Q)}(\phi_{\perp} Q) = extr_{EP(Q)}(\phi_{\perp}\widehat{Q})$.

Since $EP(Q) = EP(\widehat{Q})$ by proposition 6.3(2), we shall always refer to $EP(Q)$ rather than $EP(\widehat{Q})$ in the remainder of this chapter; this serves to emphasise the fact that we do not alter any of our interpretive constructs simply because we work with $\widehat{Q}$ rather than Q.

6.4 The traces model

We now show how to verify that $extr_{EP(Q)}(\tau\widehat{Q}) \subseteq \tau P$. To do this, it is necessary to define a process context which encodes the extraction mapping over traces. $\widehat{Q}$ is then placed into that context, thus defining a process which has exactly the traces of $extr_{EP(Q)}(\tau\widehat{Q})$. It is therefore necessary to encode a function from traces to traces as a CSP context, since $extr_{EP(Q)}$ over traces is such a function.

The basic approach followed is similar to that employed to *extract* traces in the algorithms given in [11] and [16]. Intuitively, for each extraction pattern ep_i where $i \in inv$, we wish to define a process TE_i (*T*race *E*xtraction), the traces of which are essentially *pairs* of traces. The left-hand trace of each pair would be $x \in Dom_i$ and the right-hand trace would be $extr_i(x)$. In practice, of course, it is not possible to define a process which has pairs of traces. However, a similar effect can be achieved by effectively defining events as pairs. Using this approach, if $t \in \tau TE_i$ such that $|t| = k$,

$$t = \langle (x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k) \rangle.$$

We may then consider t to be given by (x, y), where $x = \langle x_1, x_2, \ldots, x_k \rangle$ and $y = \langle y_1, y_2, \ldots, y_k \rangle$; moreover, $x \in Dom_i$ and $extr_i(x) = y$.[8]

[8]Note that, for $1 \leq j \leq k$, it may be the case that y_j is effectively a "null" event. This will happen if $extr_i(\langle x_1, \ldots, x_{j-1} \rangle) = extr_i(\langle x_1, \ldots, x_j \rangle)$. In such a case, the event pair (x_j, y_j) would be represented by x_j alone and y_j would not appear in y.

Events from $\widehat{Q}$ are renamed in order that they may be synchronized during parallel composition with those from the set of TE_i for $i \in inv$. The result of this parallel composition is a process, S, where if

$$u = \langle (x_1, y_1), (x_2, y_2), \ldots, (x_l, y_l) \rangle \in \tau S$$

then $\langle x_1, x_2, \ldots, x_l \rangle \in \tau\widehat{Q}$ and $\langle y_1, y_2, \ldots, y_l \rangle = extr_{EP(Q)}(\langle x_1, x_2, \ldots, x_l \rangle)$. Finally, events in S are hidden and renamed as necessary so that, for such a $u \in \tau S$, $\langle y_1, y_2, \ldots, y_l \rangle$ is substituted for u. This means that the resulting process has exactly the traces of $extr_{EP(Q)}(\tau\widehat{Q})$.

6.4.1 Constructing the TE_i

We now show how to construct the processes TE_i for $i \in inv$. The first problem to address is the nature of the events that will be used to represent the pairs of events described above. Let (a, b) be an event pair. In the case that b is a null event — i.e. the occurrence of a does not cause the extraction of a specification event — the pair is simply encoded as a, as described above. If this is not the case, however, a *pair* of events have to be encoded by a *single* event occurring on a *single* channel. As a result, we are required to define a number of new channels, with corresponding new data types. The name of the new channel will encode the name of the channel on which the event a was transmitted; the data type of the new channel will need to represent the data type of the channel on which a was transmitted, along with both the name and data type of the channel on which b was transmitted. In general, the approach to be taken will be as follows.

Let $a = cn_a.dv_a$ and $b = cn_b.dv_b$. Thus, a is an event occurring on channel cn_a which transmits the data value dv_a. Similarly, b is an event occurring on channel cn_b which transmits dv_b. Moreover, assume that dt_a is the type of channel cn_a and dt_b is the type of cn_b. We define a new channel, called $extract_{cn_a}$, where the name of the original channel on which a occurred — i.e. cn_a — may be derived from the subscript of the new name.[9] The data type of this new channel is $dt_a.name.dt_b$, where $name$ is a data type containing a single value, namely the label of the channel cn_b.[10] As a result, the event pair (a, b) would be encoded as:

[9]Note that, in machine-readable CSP, we cannot define channel names containing subscripts. We use the device here for the purposes of presentation and, in practice, would define the new channel name to be the concatenation of the string "extract" and the string denoting the name of the original channel.

[10]Note that, in machine readable CSP, we may not actually define a data type containing a channel name as a data value: we take that approach here for ease of presentation in stating generic definitions and, in practice, *name* would store a capitalized or abbreviated version of the channel name (see the verification of the running example in section 6.4.3 below).

$$extract_{cn_a}.dv_a.cn_b.dv_b$$[11]

If we consider the running example from figure 1.1 and ep_{ack} introduced in section 4.1.7, the trace $\langle data.0, ack.yes\rangle$ extracts to $\langle send.0\rangle$. If it were possible to use the notion of event pairs directly, we could encode this extraction using the trace $\langle data.0, (ack.yes, send.0)\rangle$. (Note that $data.0$ remains in its original form since its occurrence does not cause the extraction of an additional event.) Since we cannot use such pairs directly, in the CSP representation of the extraction mapping this trace would become $\langle data.0,\ extract_{ack}.yes.Send.0\rangle$ (note that we use the label $Send$ to represent the channel name $send$ as a data value). As another example, consider $\langle data.0, ack.no, data.0\rangle$, also extracting to $\langle send.0\rangle$. In this case, using event pairs we would have $\langle data.0, ack.no,\ (data.0, send.0)\rangle$ In the extraction mapping representation the trace would become $\langle data.0, ack.no,$ $extract_{data}.0.Send.0\rangle$. Note that we have both an occurrence of the unchanged $data.0$ and also an occurrence of $data.0$ modified to allow the extraction function to be encoded.

Defining TE_i for $i \in inv$

Let $i \in inv$. Then we define the process TE_i to encode the extraction mapping $extr_i$, where $TE_i \triangleq TE_i(\langle\rangle)$ and

$$TE_i(t) = \Box_{a \in Next_i(t)} \pi_i(a,t) \rightarrow TE_i(t \circ \langle a\rangle).$$

For ease of expression, the function π_i is used here to encode the modifications that must be made to the events in A_i, although its effects must be implemented directly in any input supplied to FDR2, since it cannot be encoded as such in CSP (see the example below in section 6.4.3). It is defined as follows.[12]

Definition 6.6. *Let $t \circ \langle a\rangle \in Dom_i$ such that $a = cn_a.dv_a$ and let $b = cn_b.dv_b$. Then:*

$$\pi_i(a,t) \triangleq \begin{cases} a & \text{if } extr_i(t) = extr_i(t \circ \langle a\rangle) \\ extract_{cn_a}.dv_a.cn_b.dv_b & \text{if } extr_i(t \circ \langle a\rangle) = extr_i(t) \circ \langle b\rangle \end{cases}$$

[11]Note, of course, that alternative encodings are also possible and a slightly different one is used in the verification which is presented in chapter 7. In that chapter, the channel name cn_b is actually represented by appending it to the new channel name $extract_{cn_a}$.

[12]Recall that, by TR-GLOBAL2 and the restriction imposed in section 6.0.2, if $t \circ \langle a\rangle \in Dom_i$ is such that $extr_i(t \circ \langle a\rangle) = extr_i(t) \circ r$ for some trace r, then either $r = \langle\,\rangle$ or $r = \langle b\rangle$ for some event b.

By definition 6.4 and since $\langle\rangle \in Dom_i$ by EP3-T, $t \in Dom_i$ for any t such that $TE_i(t)$ is a term used in the definition of TE_i. Moreover, $t \circ \langle a \rangle \in Dom_i$ for any $a \in Next_i(t)$. Thus, $\pi_i(a,t)$ is defined whenever it is used in the definition of TE_i. Observe that $\pi_i(a,t)$ simply returns a if the extraction of t is identical to the extraction of $t \circ \langle a \rangle$. In the other case — namely that the extraction of t is a strict prefix of the extraction of $t \circ \langle a \rangle$ — we are effectively encoding the fact that a is the left-hand component of an event pair and b is the right-hand component. We assume that any event of the form $extract_{cn_a}.dv_a.cn_b.dv_b$ which is returned by π_i is distinct from all other events in $\bigcup_{ep_i \in EP} A_i$ (recall that EP gives the universe of all extraction patterns). This assumption is encapsulated in the following condition.

DIS Let $t \circ \langle a \rangle \in Dom_i$ for $i \in inv$ be such that $extr_i(t \circ \langle a \rangle) = extr_i(t) \circ \langle b \rangle$ for some event b. Then $\pi_i(a,t) \notin \bigcup_{ep_i \in EP} A_i$.

The following result on the events in which TE_i may engage is due to the detail in figure 2.5, definition 6.4 and the fact that $\langle\rangle \in Dom_i$ by EP3-T.

Proposition 6.6. $\beta(TE_i) = \{\pi_i(a,t) \mid t \circ \langle a \rangle \in Dom_i\}$.

Renaming functions

We now give the renaming functions[13] which, for $i \in inv$, can be used to reclaim Dom_i and $\{extr_i(t) \mid t \in Dom_i\}$ respectively from τTE_i. The renaming $domain : \bigcup_{i \in inv} \beta(TE_i) \rightarrow A_{inv}$ will return the former and $extract : \bigcup_{i \in inv} \beta(TE_i) \rightharpoonup B_{inv}$ will return the latter.

Definition 6.7. *The following hold by definition, for $i \in inv$:*

1. *Let $t \circ \langle a \rangle \in Dom_i$. Then $domain(\pi_i(a,t)) \triangleq a$.*
2. *Let $t \circ \langle a \rangle \in Dom_i$ be such that $extr_i(t \circ \langle a \rangle) = extr_i(t) \circ \langle b \rangle$. Then $extract(\pi_i(a,t)) \triangleq b$.*

We first note that, as stated in chapter 2 (page 16), partial renamings behave as the identity mapping when applied to any event over which they are not explicitly defined. By proposition C.4(1,2) in appendix C.2, $\tau(TE_i[domain]) = Dom_i$ for $i \in inv$. *extract* is defined explicitly only for those events in $\bigcup_{i \in inv} \beta(TE_i)$ which encode an event pair with a non-null right-hand component: i.e. those event pairs (a,b) where the occurrence of a leads to the extraction of b. By proposition C.4 in appendix C.2,

[13]Although, in the general case, we use renaming relations, it happens that these are functions: this follows from definition 6.7 itself and also definition 6.6.

$\tau((TE_i \setminus A_i)[extract]) = \{extr_i(t) \mid t \in Dom_i\}$ for $i \in inv$. (The hiding of events in A_i removes all those "event pairs" encoded by a single event: i.e. those which represent an implementation event occurrence which does not lead to the extraction of a specification event.)

6.4.2 Extracting the traces of $\widehat{Q}$

It is unnecessary to define processes TE_i for ep_i such that $i \notin inv$, since $extr_i$ is the identity mapping in such cases by EP4-FVI (recall that $A_i \subseteq Fvis$ if $i \notin inv$). We thus define as follows the process TE_{inv}, which will be used to extract the traces of $\widehat{Q}$:

$$TE_{inv} \triangleq |||_{i \in inv} TE_i.$$

Once TE_{inv} has been defined, it must be composed in parallel with $\widehat{Q}$ before applying the hiding and renaming which will mimic the application of the extraction mapping. In order for $\widehat{Q}$ to synchronize in parallel with TE_{inv}, its events must be renamed: each event from A_{inv} in which $\widehat{Q}$ may engage is renamed to all those "event pairs" in $\beta(TE_{inv}) = \bigcup_{i \in inv} \beta(TE_i)$ of which it forms the left-hand component. The renaming used for this purpose is $prep : A_{inv} \times \beta(TE_{inv})$, which is defined as follows.

Definition 6.8. *Let $i \in inv$ and $a \in A_i$ be such that there exists a trace u such that $u \circ \langle a \rangle \in Dom_i$. Then:*

$$prep(a) \triangleq \{\pi_i(a,t) \mid t \circ \langle a \rangle \in Dom_i\}.$$

Note that, for $t \circ \langle a \rangle \in \tau\widehat{Q}$ such that $a \in A_i$ for $i \in inv$, $t \lceil A_i \circ \langle a \rangle \in Dom_i$ by proposition 6.4(1) and TR-GLOBAL1. The following process, *TraceExtract*, then has exactly the traces of $\widehat{Q}$ after extraction, which fact is shown by theorem 6.7.

$$TraceExtract \triangleq ((\widehat{Q}[prep] \parallel_{prep(A_{inv})} TE_{inv}) \setminus A_{inv})[extract]$$

Theorem 6.7. $extr_{EP(Q)}(\tau\widehat{Q}) = \tau\, TraceExtract.$

Corollary 6.8. $extr_{EP(Q)}(\tau\widehat{Q}) \subseteq \tau P$ *if and only if* $TraceExtract \sqsupseteq_T P$.

Corollary 6.8 therefore allows us to verify automatically using FDR2 that $extr_{EP(Q)}(\tau\widehat{Q}) \subseteq \tau P$.

6.4.3 Example

Here we show how to apply the results in this section to verify that the extracted traces of *LeftImpl* are contained in those of *LeftSpec*. (Note that the components defined here can be used without modification to verify that the extracted traces of *RightImpl* are contained in those of *RightSpec*.) Let $\widehat{Q}$ be *LeftImpl* after the application of the necessary preprocessing. Let ep_1 be ep_{ack} defined in section 4.1.7. Recall that $A_1 = \alpha data \cup \alpha ack$ and $B_1 = \alpha send$. We then define TE_1 to encode $extr_1$:

$$TE_1 = \Box_{x \in \{0,1\}} data.x \rightarrow T_1(x)$$

where $T_1(x)$ is defined as:

$$((extract_{ack}.yes.Send.x \rightarrow TE_1) \,\Box\, (ack.no \rightarrow extract_{data}.x.Send.x \rightarrow TE_1)).$$

The concrete renamings *prep* and *extract* used in this example are as follows:

$$\begin{aligned} prep \;\triangleq\; & \{(ack.yes, extract_{ack}.yes.Send.0), \\ & (ack.yes, extract_{ack}.yes.Send.1), (data.0, data.0), \\ & (data.1, data.1), (ack.no, ack.no), \\ & (data.0, extract_{data}.0.Send.0), (data.1, extract_{data}.1.Send.1)\}. \end{aligned}$$

$$\begin{aligned} extract \;\triangleq\; & \{(extract_{ack}.yes.Send.0, send.0), \\ & (extract_{ack}.yes.Send.1, send.1), \\ & (extract_{data}.0.Send.0, send.0), \\ & (extract_{data}.1.Send.1, send.1)\}. \end{aligned}$$

Note that here $TE_{inv} = TE_1$ and $A_{inv} = A_1$. Using the process expressions defined here, we were able to define *TraceExtract* as above and, by virtue of corollary 6.8, verify automatically using FDR2 that the extracted traces of *LeftImpl* are contained in those of *LeftSpec*. In a similar manner, we were able to verify that the extracted traces of *RightImpl* are contained in those of *RightSpec*. Thus, since both *LeftImpl* and *RightImpl* meet Dom-T-check, since $\alpha ImplNet \subseteq Fvis$[14] and by theorem 4.8, we may infer that

$$ImplNet \sqsupseteq_T SpecNet.$$

[14]Recall that $\alpha in \cup \alpha out \subseteq Fvis$. Recall also that

- $ImplNet \triangleq LeftImpl \otimes_{(\alpha data \cup \alpha ack)} RightImpl$.
- $SpecNet \triangleq LeftSpec \otimes_{\alpha send} RightSpec$.

6.4.4 Further comments on defining D_i and TE_i

Determinism Due to the need to calculate here the semantics of processes D_i and TE_i from their syntactic representation, they are effectively presented in a particular normal form, namely using only (indexed) deterministic choice, the prefix operator and recursion. Provided that a CSP process P to be used with FDR2 is *deterministic* then there is a process P' constructed using only indexed deterministic choice[15], prefix and recursion which is semantically indistinguishable from P (see condition DE in section 2.9). As a result, candidates for D_i and TE_i respectively which are to be used in practice need only be deterministic, rather than defined using this restrictive syntax. (FDR2 can be used to check the determinism of CSP processes.) Of course, candidates for TE_i should still be defined so that they engage only in event pairs which may be generated by π_i.

Defining extraction mappings In practice, it need not be the case that the extraction pattern used in the verification of any particular implementation process will exist prior to that verification. In other words, extraction patterns may simply be created according to our needs during any particular verification. This is what happens in the verification of the asynchronous communication mechanism described in chapter 7. As a result, the only direct definition of Dom_i for $i \in inv$ which we have may be given by the traces of the implementation process under consideration: this is one reason why it may not actually be possible to define D_i and TE_i using only choice, prefix and recursion. In the verification in chapter 7, a single extraction pattern, which we shall denote ep_i, is used in the verification of the implementation process described there, which process we shall call Q'. Since we have no direct statement of Dom_i we take it to be $\tau Q'$, which guarantees that any mapping used will be defined for all traces of Q'. Assuming the existence of a suitable renaming *domain*, it is difficult to define TE_i such that $\tau(TE_i[domain]) = \tau Q'$, since to do so would require direct syntactic modification of the implementation process itself. The approach taken, therefore, is to define TE_i such that $\tau Q' \subset \tau(TE_i[domain])$. Using such an approach, it is as if we take some larger mapping and assume that $extr_i$ is defined as its restriction to the domain Dom_i. The restriction to Dom_i will then occur automatically when the process TE_i is composed in parallel with the renamed Q'.

[15]If we index the deterministic choice operator with the empty set then this is semantically equivalent to $STOP$.

6.5 Verifying Dom-SF-check

We now move on to consider the verification of condition Dom-SF-check for $\widehat{Q}$:

Dom-SF-check Let $(t, R) \in \phi\widehat{Q}$ be such that $R \subseteq \alpha\widehat{Q}$. Let $ep_i \in EP(Q)$ be such that $A_i \cap Fvis = \varnothing$. If $extr_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) = B_i$ then $t\lceil A_i \in dom_i$.

We show in this section how the verification of this condition can be transformed into a check for deadlock freedom[16] on a (number of) process(es) derived from $\widehat{Q}$.[17] This transformation is based on a simple consideration of the respective definitions of $extr^{ref}$, ref and $\overline{ref}$. (Recall that, for $i \in inv$ and $t \in Dom_i$, $\overline{ref}_i(t)$ gives the set of all $X \subseteq A_i$ such that, for every $Y \in ref_i(t)$, $X \cup Y \neq A_i$.) Before proceeding we define $RefSet_i$ for $i \in inv$, which is used extensively in what follows:

Definition 6.9. *Let $i \in inv$. Then:*

$$RefSet_i \triangleq \begin{cases} ref_i & \text{if } Comm(A_i, \widehat{Q}) = Right \\ \overline{ref}_i & \text{if } Comm(A_i, \widehat{Q}) = Left \end{cases}$$

Note that $RefSet_i$ associates the communication capability *Left* with $\overline{ref}_i$ and *Right* with ref_i for $i \in inv$; in contrast, the definition of $extr_i^{ref}$ in definition 4.10 associates *Left* with ref_i and *Right* with $\overline{ref}_i$. This will prove to be crucial in what follows: in particular, it allows us to characterise in terms of $RefSet_i$ whether or not Dom-SF-check is met.

Let $i \in inv$ and $(t, R) \in \phi\widehat{Q}$ such that $R \subseteq \alpha\widehat{Q}$ and $t\lceil A_i \notin dom_i$. If $\widehat{Q}$ meets Dom-SF-check, then $extr_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) = \varnothing$. In the case that $Comm(A_i, \widehat{Q}) = Right$, this means that $R \cap A_i \in \overline{ref}_i(t\lceil A_i)$ by definition 4.10. Hence, by definitions 4.9 and 6.9, there does not exist $X \in ref_i(t\lceil A_i) = RefSet_i(t\lceil A_i)$ such that $X \cup (R \cap A_i) = A_i$. In the case that $Comm(A_i, \widehat{Q}) = Left$ then $R \cap A_i \in ref_i(t\lceil A_i)$ and so, again by definitions 4.9 and 6.9, there does not exist $Y \in \overline{ref}_i(t\lceil A_i) = RefSet_i(t\lceil A_i)$ such that $Y \cup (R \cap A_i) = A_i$.

Consider, then, the case that $\widehat{Q}$ does *not* meet Dom-SF-check. Then there exists $(t, R) \in \phi\widehat{Q}$ such that $R \subseteq \alpha\widehat{Q}$ and $i \in inv$ such that $t\lceil A_i \notin dom_i$, where $extr_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) = B_i$. Hence, by definitions 4.9, 4.10 and 6.9, there exists $X \in RefSet_i(t\lceil A_i)$ such that $X \cup (R \cap A_i) = A_i$. In a

[16]Recall that a process W is deadlock-free if and only if $R \subset \Sigma$ for every $(t, R) \in \phi W$.

[17]This is similar in some respects to the use of tester processes in [6], where the question of whether one process implements another is transformed into a question of deadlock-freedom of the implementation composed in parallel with the tester process.

suitable process, deadlock will occur due to the refusal of all events in A_i, thus indicating that $extr_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) = B_i$ and so Dom-SF-check has been breached. We therefore define a process (to be interpreted in the stable failures model) for each $i \in inv$ such that its refusals after $t \in (Dom_i - dom_i)$ are given by $RefSet_i(t)$ and where $t \in dom_i$ does not form the trace component of any failure. Each such process for $i \in inv$ is composed in parallel with a modified $\widehat{Q}$ so that the resulting process deadlocks — due to refusing all events in A_i — if and only if condition Dom-SF-check is *not* met with respect to ep_i (i.e. $extr_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) = B_i$ while $t\lceil A_i \notin dom_i$).

6.5.1 Defining the "tester" process

We first show how to define the "tester" process DSF_i for $i \in inv$ (we need not define such a process for $i \notin inv$ by the definition of Dom-SF-check). This definition makes use of the semantic definition of DIV — the immediately diverging process — in the stable failures model: recall that the meaning of DIV in this model is $(\{\langle\rangle\}, \varnothing)$. Where P is a process and by the semantic definition of $\Box$ in figure 2.3, we have that:

$$\phi(P \Box DIV) = \{(t, R) \in \phi P \mid t \neq \langle\rangle\}.$$

For example, $\phi((a \rightarrow STOP) \Box DIV) = \{(\langle a\rangle, R) \mid R \subseteq \Sigma\}$ (see the use of DIV in the definition of processes TP and FP in section 2.10). We may therefore use DIV to obscure failures in which we are not interested, specifically those where the trace component is $t \in dom_i$ since these can be ignored by the definition of Dom-SF-check. We therefore define the process DSF_i for $i \in inv$, the failures of which are defined by $Dom_i - dom_i$ and $RefSet_i$ (see lemma 6.9 below).

Let $i \in inv$. For $t \in Dom_i$, we define $ref_i^M(t)$ as the set of sets from $ref_i(t)$ which are maximal in the subset-ordering.

Definition 6.10. $ref_i^M(t) \triangleq \{R \in ref_i(t) \mid (\nexists S \in ref_i(t))\ R \subset S\}$.

We first give the definition of DSF_i when $Comm(A_i, \widehat{Q}) = Right$. (Note that $t \notin dom_i$ is used as shorthand for the fact that $t \in Dom_i - dom_i$.) In this case, $DSF_i \triangleq DSF_i^R(\langle\rangle)$ and:

$$DSF_i^R(t) = \begin{cases} (\Box_{a \in Next_i(t)} a \rightarrow DSF_i^R(t \circ \langle a\rangle)) \Box DIV & \text{if } t \in dom_i \\ ((\Box_{a \in Next_i(t)} a \rightarrow DSF_i^R(t \circ \langle a\rangle)) \Box DIV) \sqcap & \text{if } t \notin dom_i \\ (\sqcap_{R \in ref_i^M(t)} (\Box_{a \in (A_i - R)} a \rightarrow DIV)) & \end{cases}$$

We now give the definition of DSF_i when $Comm(A_i, \widehat{Q}) = Left$. In this case, $DSF_i \triangleq DSF_i^L(\langle\rangle)$ and:

$$DSF_i^L(t) = \begin{cases} (\Box_{a \in Next_i(t)} a \rightarrow DSF_i^L(t \circ \langle a \rangle)) \ \Box \ DIV & \text{if } t \in dom_i \\ ((\Box_{a \in Next_i(t)} a \rightarrow DSF_i^L(t \circ \langle a \rangle)) \ \Box \ DIV) \ \sqcap & \text{if } t \notin dom_i \\ (\Box_{R \in ref_i^M(t)} (\sqcap_{a \in (A_i - R)} a \rightarrow DIV)) & \end{cases}$$

Note that, by EP5 and definition 6.10, $\sqcap$ is never indexed by the empty set in either of these definitions; similarly, $\Box$ is never indexed by the empty set in the last line of the definition of $DSF_i^L(t)$ (this latter is important in the proof of lemma 6.9). By definition 6.4 and since $\langle\rangle \in Dom_i$ by EP3-T, $t \in Dom_i$ for any t used to parameterise a term in the definition of DSF_i. Hence, ref_i^M is defined wherever it is used. The two versions of DSF_i are almost identical: in fact, the only difference is the ordering of the operators $\Box$ and $\sqcap$ in the last line of the respective definitions. This line is used to encode (the refusals from) $RefSet_i$ and the difference reflects the fact that it is given by ref_i in one case and by $\overline{ref}_i$ in the other. The stable failures of DSF_i are characterised by the following lemma.

Lemma 6.9. *The following holds, for $i \in inv$:*

$\phi DSF_i = \{(t, X \cup Y) \mid t \in Dom_i - dom_i \land X \in RefSet_i(t) \land Y \subseteq (\Sigma - A_i)\}$.

Examples of such processes DSF_i can be seen below in section 6.5.4.

6.5.2 Transforming the implementation process

Let $i \in inv$. We now show how to transform the implementation process $\widehat{Q}$, such that its failures are projected onto A_i: that is, if $(t, R) \in \phi\widehat{Q}$ then $(t\lceil A_i, (R \cap A_i) \cup (\Sigma - A_i))$ is a failure of the transformed process. This transformation is effected using the process $Proc_i$.

$$Proc_i = ((\Box_{a \in \alpha\widehat{Q}} a \rightarrow Proc_i) \ \Box \ DIV) \ \sqcap \ (\Box_{a \in A_i} a \rightarrow DIV).$$

$\widehat{Q}$ is composed in parallel with $Proc_i$ with the result that, for every failure $(t, R) \in \phi\widehat{Q}$, the refusal R has $\alpha\widehat{Q} - A_i$ added to it. This means that the refusal $R \cap A_i$ will survive the hiding of the events in $\alpha\widehat{Q} - A_i$: i.e. $(t\lceil A_i, R \cap A_i)$ will be a failure of the process resulting from the hiding of $\alpha\widehat{Q} - A_i$. We therefore define the following process:

$$\widehat{Q}_i \triangleq (\widehat{Q} \parallel_{\alpha\widehat{Q}} Proc_i) \setminus (\alpha\widehat{Q} - A_i).$$

The stable failures of $\widehat{Q}_i$ are given by the following result.

Lemma 6.10. *The following holds, for $i \in inv$:*

$$\phi\widehat{Q}_i = \{(t\lceil A_i, R) \mid (\exists(t, X) \in \phi\widehat{Q}) \ R \subseteq (X \cap A_i) \cup (\Sigma - A_i)\}.$$

6.5.3 The verification

We are now in a position to define the process $\mathit{FinalImple}_i$, for $i \in \mathit{inv}$, upon which the check for deadlock-freedom[18] will be carried out:

$$\mathit{FinalImple}_i \triangleq \widehat{Q}_i \parallel_{A_i} \mathit{DSF}_i.$$

Theorem 6.11. *$\widehat{Q}$ meets condition Dom-SF-check if and only if, for every $i \in \mathit{inv}$, $\mathit{FinalImple}_i$ is deadlock-free.*

The above result allows us to proceed to automatic verification of condition Dom-SF-check.

6.5.4 Example

We now show how to define the relevant process expressions used to verify that *LeftImpl* and *RightImpl* respectively meet condition Dom-SF-check. Note that, in both cases, the condition need be checked only with respect to the extraction pattern ep_{ack}, the relevant components of which are restated here.

dom_{ack} is defined as follows:

$$\begin{aligned} dom_{ack} \triangleq\ & \{\langle data.0, ack.yes\rangle, \langle data.0, ack.no, data.0\rangle, \\ & \langle data.1, ack.yes\rangle, \langle data.1, ack.no, data.1\rangle\}^*. \end{aligned}$$

Recall that Dom_{ack} is given by the prefix-closure of dom_{ack}. The ref_{ack} component, where $t \in dom_{ack}$ and $t \circ u \in Dom_{ack}$, is given as:

$$ref_{ack}(t \circ u) \triangleq \begin{cases} 2^{\alpha data} & \text{if } u = \langle data.v\rangle \\ \{R \in 2^{\alpha data \cup \alpha ack} \mid \alpha data \not\subseteq R\} & \text{if } u = \langle\rangle \\ \{R \in 2^{\alpha data \cup \alpha ack} \mid data.v \notin R\} & \text{if } u = \langle data.v, ack.no\rangle \end{cases}$$

We assume ep_1 is ep_{ack}, where $A_1 = \alpha data \cup \alpha ack$.

Verifying *RightImpl*:

In this case, $\mathit{Comm}(A_1, \mathit{RightImpl}) = \mathit{Right}$. Assume that $\widehat{Q}$ is *RightImpl* after the application of preprocessing as described in section 6.3. We also assume the existence of an extraction pattern ep_2 to "interpret" the events occurring on channel *out* (recall that $\alpha out \subseteq \mathit{Fvis}$), where $A_2 = \alpha out$. Thus, by proposition 6.3(2) and definition 4.5, $\alpha\widehat{Q} = \alpha data \cup \alpha ack \cup \alpha out$. Assume that Proc_1 is defined according to the template given above in section 6.5.2,

[18]Recall that a process W is deadlock-free if and only if $R \subset \Sigma$ for every $(t, R) \in \phi W$.

by substituting $\alpha data \ \cup \ \alpha ack \ \cup \ \alpha out$ for $\alpha\widehat{Q}$ and $\alpha data \ \cup \ \alpha ack$ for A_i. We then define $\widehat{Q}_1 \triangleq (\widehat{Q} \parallel_{\alpha\widehat{Q}} Proc_1) \setminus \alpha out$. The tester process DSF_1 used here is defined in terms of two auxiliary processes $DSF'_1(x)$ and $DSF''_1(x)$.

$$
\begin{aligned}
DSF_1 &= (\Box_{x\in\{0,1\}}(data.x \rightarrow DSF'_1(x)) \Box DIV \\
DSF'_1(x) &= ((ack.yes \rightarrow DSF_1 \Box ack.no \rightarrow DSF''_1(x)) \Box DIV) \sqcap \\
&\quad (\sqcap_{R\in\{\alpha data\}}(\Box_{y\in(A_1-R)}\ y \rightarrow DIV)) \\
DSF''_1(x) &= ((data.x \rightarrow DSF_1) \Box DIV) \sqcap \\
&\quad (\sqcap_{R\in\{A_1-\{data.x\}\}}(\Box_{y\in(A_1-R)}\ y \rightarrow DIV)).
\end{aligned}
$$

From these process expressions, we were able to define $FinalImple_1$ for $ep_1 = ep_{ack}$ and $RightImpl$, and check it for deadlock freedom using FDR2, as a result of which we successfully verified condition Dom-SF-check for *RightImpl*.

Verifying *LeftImpl*:

In this case, $Comm(A_1, LeftImpl) = Left$. Assume that $\widehat{Q}$ is *LeftImpl* after preprocessing. We also assume the existence of an extraction pattern ep_2 to "interpret" the events occurring on channel *in* (recall that $\alpha in \subseteq Fvis$), where $A_2 = \alpha in$. Thus, by proposition 6.3(2) and definition 4.5, $\alpha\widehat{Q} = \alpha data \ \cup \ \alpha ack \ \cup \ \alpha in$. Assume that $Proc_1$ is defined according to the template given above, by substituting $\alpha data \ \cup \ \alpha ack \ \cup \ \alpha in$ for $\alpha\widehat{Q}$ and $\alpha data \ \cup \ \alpha ack$ for A_i. We then define $\widehat{Q}_1 \triangleq (\widehat{Q} \parallel_{\alpha\widehat{Q}} Proc_1) \setminus \alpha in$. The tester process DSF_1 used here is defined in terms of two auxiliary processes $DSF'_1(x)$ and $DSF''_1(x)$.

$$
\begin{aligned}
DSF_1 &= (\Box_{x\in\{0,1\}}(data.x \rightarrow DSF'_1(x)) \Box DIV \\
DSF'_1(x) &= ((ack.yes \rightarrow DSF_1 \Box ack.no \rightarrow DSF''_1(x)) \Box DIV) \sqcap \\
&\quad (\Box_{R\in\{\alpha data\}}(\sqcap_{y\in(A_1-R)}\ y \rightarrow DIV)) \\
DSF''_1(x) &= ((data.x \rightarrow DSF_1) \Box DIV) \sqcap \\
&\quad (\Box_{R\in\{A_1-\{data.x\}\}}(\sqcap_{y\in(A_1-R)}\ y \rightarrow DIV)).
\end{aligned}
$$

From these process expressions, we were able to define $FinalImple_1$ for $ep_1 = ep_{ack}$ and *LeftImpl*, and check it for deadlock freedom using FDR2, as a result of which we successfully verified condition Dom-SF-check for *LeftImpl*.

6.6 The stable failures model

We now consider how to verify that $extr_{EP(Q)}([\![\widehat{Q}]\!]_{SF}) \subseteq [\![P]\!]_{SF}$, which requires the interpretation of both the traces and stable failures of $\widehat{Q}$. Section 6.4 showed how to do the former. The challenge faced here, therefore, is to encode as a CSP context the mapping applied to refusal/trace pairs. Before we show how to do this, an important issue must be raised.

Condition SF4 given in section 2.4.2 defines the following relationship which exists between the traces and failures of any process W in the stable failures model:

$$(t, R) \in \phi W \ \wedge\ t \circ \langle a \rangle \notin \tau W \Rightarrow (t, R \cup \{a\}) \in \phi W.$$

But $extr_{EP(Q)}(\tau\widehat{Q})$ and $extr_{EP(Q)}(\phi\widehat{Q})$ may not respect this relationship. Consider a refusal-maximal failure $(w, X) \in extr_{EP(Q)}(\phi\widehat{Q})$. Then, by SF-DEF2, there exists $(t, R) \in \phi_{dom_{EP(Q)}}\widehat{Q}$ such that

$$extr_{EP(Q)}(t) = w \text{ and } X = extr^{ref}_{EP(Q)}(R, t, \widehat{Q}) \cup (\Sigma - extr^{set}(\alpha\widehat{Q})).$$

It may be the case that, for $ep_i \in EP(Q)$ where $i \in inv$, $X \cap B_i = \varnothing$ since $extr^{ref}_i(R \cap A_i, t\lceil A_i, \widehat{Q}) = \varnothing$. However, it need not hold that $extr_{EP(Q)}(t) \circ \langle b \rangle \in extr_{EP(Q)}(\tau\widehat{Q})$ for every $b \in B_i$ and usually it will not. This means that, in the general case, there need not be a (syntactic) process the semantics of which is given by $extr_{EP(Q)}([\![\widehat{Q}]\!]_{SF})$. One consequence of this is that, although the extracted traces of $\widehat{Q}$ are represented directly in the syntactic term we define, its extracted failures are represented in an encoded form (see section 6.6.1 for further details). Another consequence is that the specification process P has to be modified before the verification question under consideration here can be framed as a refinement check in FDR2 (see section 6.6.2).

The interdependency between traces and failures also gives rise to another problem. In particular, if we apply to $\widehat{Q}$ the machinery necessary to extract traces then this will modify the failures of the resulting process: we no longer have a (syntactic) record of the original failures of $\widehat{Q}$ and so would be unable to apply further syntactic manipulations in order to encode the extracted failures of $\widehat{Q}$. Similarly, syntactic manipulations which are necessary to encode the extraction of the failures of $\widehat{Q}$ will have possibly undesirable effects on the traces of the resulting process. For example, consider the simple implementation process $W = a \rightarrow STOP$, where:

- $\tau W = \{\langle\rangle, \langle a \rangle\}$ and
- $\phi W = \{(\langle\rangle, R) \mid R \subseteq \Sigma \ \wedge\ a \notin R\} \cup \{(\langle a \rangle, R) \mid R \subseteq \Sigma\}$.

In the event that $extr_{EP(W)}(\langle a \rangle) = \langle\,\rangle$, we would have to hide $\{a\}$ in order to extract the traces of W. However, the only failure of the resulting process would be $\{(\langle\,\rangle, R) \mid R \subseteq \Sigma\}$. In the event that $extr_{EP(W)}(\langle a \rangle) = \langle b \rangle$, then a would eventually be renamed to b and this would alter the refusals of the process. Similarly, any attempt to extract the refusals of W would involve some manipulation of the event a and so the traces of the process would also be affected.

In order to solve this problem, we introduce the notion of *primed* events: this allows us to separate the events used to generate the traces of any process under consideration from those used to generate the refusals. For example, we could represent W as

$$W' = ((a \rightarrow STOP) \;\Box\; DIV) \;\sqcap\; (a' \rightarrow DIV),$$

where

- $\tau W' = \{\langle\,\rangle, \langle a \rangle, \langle a' \rangle\}$ and
- $\phi W = \{(\langle\,\rangle, R) \mid R \subseteq \Sigma \;\wedge\; a' \notin R\} \cup \{(\langle a \rangle, R) \mid R \subseteq \Sigma\}$.

Thus, the event a could be manipulated as necessary in order to extract the traces of W' but this would not affect the *refusals* of the process (it would, of course, affect the trace component of one of the *failures* but that trace component would need to be extracted as well anyway). Similarly, a' could be manipulated during the extraction of refusals but this would not affect the trace $\langle a \rangle$ (the additional trace $\langle a' \rangle$ can effectively be ignored). This approach of using primed events to generate refusals and unprimed events to generate traces is employed below in the definition of the processes RE_i for $i \in inv$ described in section 6.6.1 (these processes are used to extract the refusals of $\widehat{Q}$).

Priming events and related issues

Before proceeding, we introduce three renaming relations which will allow us to prime events.

Definition 6.11. *The following hold by definition:*

1. *If $a \in A_{inv} \cup B_{inv}$ then $prime(a) \triangleq a'$.*
2. *If $a \in A_{inv}$ then $p^Q(a) \triangleq \{a, a'\}$.*
3. *If $a \in B_{inv}$ then $p^P(a) \triangleq \{a, a'\}$.*

prime converts an event from $A_{inv} \cup B_{inv}$ into its primed counterpart. p^Q returns both the original event and its primed version for finally invisible *implementation* events; p^P does the same for finally invisible *specification* events. It is assumed that the set of primed events contains only "fresh" events: i.e. it does not contain any events already used in defining $\widehat{Q}$, P or $EP(Q)$, or which are used in any other capacity as part of the verification of $\widehat{Q}$.

The act of priming an event cannot be done directly in (machine-readable) CSP and so the approach taken is as follows. We take the event to be primed and define a new channel with the same type as the original channel on which the event occurred and whose name is a concatenation of the original name and some other "reserved" word, such as *prime*. The new event will then occur on the new channel, whilst communicating the same data value as the original event. For example, if we were to "prime" the event *data*.0, the result could be $data_{prime}.0$ (see section 6.6.4 for further examples). Note, of course, that we cannot use channel names containing subscripts in machine-readable CSP: they are used here simply for the purposes of presentation and, in practice, we would use something like *dataprime*.

In the course of extracting the refusals of $\widehat{Q}$, it will also be necessary to rename each event in $prime(A_i)$ for $i \in inv$ to a distinguished event d_i.[19] The set of such events is labelled d_{inv}.

Definition 6.12. $d_{inv} \triangleq \{d_i \mid i \in inv\}$.

The renaming of events in $prime(A_i)$ to d_i for $i \in inv$ is carried out using the renaming σ^Q. (σ^P is also defined, to rename events in $prime(B_i)$ to d_i: it is used in preprocessing the specification).

Definition 6.13. *Let $i \in inv$.*

1. *If $a \in prime(A_i)$ then $\sigma^Q(a) \triangleq d_i$.*
2. *If $a \in prime(B_i)$ then $\sigma^P(a) \triangleq d_i$.*

The events d_i for $i \in inv$ are used to encode the extraction of refusals. In order to relate this encoding to $extr^{ref}_{EP(Q)}$, we introduce the mapping $extrFDR^{ref}_{EP(Q)}$.

Definition 6.14. *Let $t \in Dom_{EP(Q)}$ and $R \subseteq \alpha\widehat{Q}$. Then*

$$extrFDR^{ref}_{EP(Q)}(R, t, \widehat{Q}) \triangleq \bigcup_{1 \leq i \leq m} extrFDR^{ref}_i(R \cap A_i, t\lceil A_i, \widehat{Q}), \text{ where:}$$

[19]Each such d_i is assumed to be a "fresh" event not used or introduced elsewhere.

1. *Let $i \notin inv$. Then $extrFDR_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) \triangleq R \cap A_i$.*

2. *Let $i \in inv$. Then:*

   (a) *if $extr_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) = B_i$ then $extrFDR_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) \triangleq \{d_i\}$.*

   (b) *if $extr_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) = \varnothing$, $extrFDR_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) \triangleq \varnothing$.*

In relation to definition 6.14(1), recall that $A_i \subseteq Fvis$ if $i \notin inv$ and so $extr_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) = R \cap A_i$ by EP5-FVI.

6.6.1 Interpreting the behaviours of $\widehat{Q}$

In this section, we show how to simultaneously extract the traces and failures of $\widehat{Q}$.

Extracting refusals

The extraction of refusals uses a set of processes RE_i for $i \in inv$ which are similar to the DSF_i defined in the previous section for verifying Dom-SF-check. The only differences are that refusals are generated here using primed events and only traces in dom_i form the trace components of the stable failures of the process. This latter is because, by SF-DEF2 in figure 4.10, we are ultimately only interested in failures the trace component of which is in $dom_{EP(Q)}$.

Let $i \in inv$ and note that $t \notin dom_i$ is again used as a shorthand for the fact that $t \in (Dom_i - dom_i)$. We first give the definition of the process RE_i when $Comm(A_i, \widehat{Q}) = Right$. In this case, $RE_i \triangleq RE_i^R(\langle\rangle)$ and

$$RE_i^R(t) = \begin{cases} (\Box_{a \in Next_i(t)} a \to RE_i^R(t \circ \langle a \rangle)) \Box DIV & \text{if } t \notin dom_i \\ ((\Box_{a \in Next_i(t)} a \to RE_i^R(t \circ \langle a \rangle)) \Box DIV) \sqcap & \text{if } t \in dom_i \\ (\sqcap_{R \in ref_i^M(t)} (\Box_{a \in prime(A_i - R)} a \to DIV)) & \end{cases}$$

We now give the definition of RE_i when $Comm(A_i, \widehat{Q}) = Left$. In this case, $RE_i \triangleq RE_i^L(\langle\rangle)$ and

$$RE_i^L(t) = \begin{cases} (\Box_{a \in Next_i(t)} a \to RE_i^L(t \circ \langle a \rangle)) \Box DIV & \text{if } t \notin dom_i \\ ((\Box_{a \in Next_i(t)} a \to RE_i^L(t \circ \langle a \rangle)) \Box DIV) \sqcap & \text{if } t \in dom_i \\ (\Box_{R \in ref_i^M(t)} (\sqcap_{a \in prime(A_i - R)} a \to DIV)) & \end{cases}$$

Similar comments apply to these definitions as were made with respect to the definitions of DSF_i in the previous section. In particular, by EP5 and definition 6.10, $\sqcap$ is never indexed by the empty set in either of these definitions; also, $\Box$ is never indexed by the empty set in the last line of the definition of $RE_i^L(t)$. By definition 6.4 and since $\langle\rangle \in Dom_i$ by EP3-T, then $t \in Dom_i$ for any t used to parameterise a term in the definition of RE_i and so ref_i^M is defined wherever it is used. Also as before, the two definitions of RE_i are similar: again, the only difference is the ordering of the operators $\Box$ and $\sqcap$ in the last line of the respective definitions, due to the fact that this line encodes (the priming of) $RefSet_i$.

We also define the process, *Trim*.

$$Trim = (\Box_{a \in A_{inv}} a \rightarrow Trim) \Box (\Box_{a \in prime(A_{inv})} a \rightarrow DIV).$$

Trim is composed in parallel with $|||_{i \in inv} RE_i$ in order to ensure that primed events only appear at the *end* of traces in the resulting process, RE_{inv}, which process will be used to extract the refusals of $\widehat{Q}$:[20]

$$RE_{inv} \triangleq (|||_{i \in inv} RE_i) \parallel_{A_{inv} \cup prime(A_{inv})} Trim.$$

The following result characterises the behaviours of RE_{inv} (recall that, by definition 6.3, $Dom_{inv} \triangleq (|||_{i \in inv} Dom_i)$ and $dom_{inv} \triangleq (|||_{i \in inv} dom_i)$).[21]

Lemma 6.12. *The following hold of RE_{inv}:*

1. $\tau RE_{inv} = Dom_{inv} \cup T$, *where* $T \subseteq \{t \circ \langle prime(a)\rangle \mid t \in Dom_{inv} \wedge ((\exists i \in inv)\ a \in A_i)\}$.

2. $$\begin{aligned} \phi RE_{inv} = \ & \{(t, prime(X) \cup Y) \mid t \in dom_{inv} \wedge X \subseteq A_{inv} \wedge \\ & Y \subseteq (\Sigma - prime(A_{inv})) \wedge \\ & ((\forall i \in inv)\ X \cap A_i \in RefSet_i(t \lceil A_i))\}. \end{aligned}$$

Before composing RE_{inv} in parallel with $\widehat{Q}$, it is necessary to prime the events of $\widehat{Q}$ using p^Q. This essentially gives two copies of every event from $\widehat{Q}$ which is in A_{inv}: after composition with RE_{inv}, the unprimed events effectively define the traces of the resulting process and the primed events define the refusals, as they do in RE_{inv}. Prior to composition with RE_{inv} it is also necessary to compose the renamed $\widehat{Q}$ with the following process, *TrimTwo*, to create the process *Interim*. *TrimTwo* plays a role similar to *Trim* defined

[20] If primed events could appear elsewhere in the traces of RE_{inv}, it would interfere with the extraction of the traces of $\widehat{Q}$.

[21] Note that, in the definition of T in lemma 6.12(1), dom_{inv} could have been used in place of Dom_{inv} without invalidating the result. We use Dom_{inv} because it is sufficient for our purposes in the remainder of this section and because it eases the proof of both this result and a later result.

above: it ensures that events from $prime(A_{inv})$ only occur at the end of traces in *Interim*. We therefore define:

$$TrimTwo = (\Box_{a \in \alpha \widehat{Q}} a \to TrimTwo) \ \Box \ (\Box_{a \in prime(A_{inv})} a \to DIV)$$

and

$$Interim \triangleq \widehat{Q}[p^Q] \parallel_{\alpha \widehat{Q} \cup prime(A_{inv})} TrimTwo.$$

Consider a failure $(t, R) \in \phi_{dom_{EP(Q)}} \widehat{Q}$ and let $i \in inv$. Once RE_{inv} has been composed in parallel with *Interim*, the resulting process will refuse $prime(A_i)$ after t if $extr_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) = B_i$. (This behaviour is similar to that induced when DSF_i is composed in parallel with $\widehat{Q}_i$, as described in section 6.5.) Similarly, only a strict subset of $prime(A_i)$ will be refused after t if $extr_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) = \varnothing$. The events from $prime(A_i)$ are then renamed to d_i using σ^Q. This has the result that $\{d_i\}$ is refused after t if $extr_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) = B_i$; moreover, $\{d_i\}$ is *not* refused after t if $extr_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) = \varnothing$.[22]

We therefore define *PreImple*, a process which has the extracted *refusals* of $\widehat{Q}$ but whose *traces* have not yet been extracted:

$$PreImple \triangleq (Interim \parallel_{A_{inv} \cup prime(A_{inv})} RE_{inv})[\sigma^Q].$$

We introduce the notation A_{Fvis} as follows.

Definition 6.15. $A_{Fvis} \triangleq \bigcup_{i \notin inv} A_i = \alpha\widehat{Q} \cap Fvis$.

The behaviours of *PreImple* are then given by the following result.

Lemma 6.13. *The following hold:*

1. $\tau PreImple = \tau\widehat{Q} \cup T$, *where* $T \subseteq \{t \circ \langle d_i \rangle \mid t \in \tau\widehat{Q} \ \wedge \ d_i \in d_{inv}\}$.
2. $\phi PreImple = \{(t, X \cup Y) \mid (\exists (t, R) \in \phi_{dom_{EP(Q)}} \widehat{Q}) \ R \subseteq \alpha\widehat{Q} \ \wedge$
 $X \subseteq extrFDR^{ref}_{EP(Q)}(R, t, \widehat{Q}) \ \wedge \ Y \subseteq (\Sigma - (A_{Fvis} \cup d_{inv}))\}$.

[22]According to the detail in section 2.4.2, for some process W and renaming G,

$$\phi(W[G]) \triangleq \{(s', X) \mid (\exists s) \ s \ G \ s' \ \wedge \ (s, G^{-1}(X)) \in \phi W\}.$$

Since $(\sigma^Q)^{-1}(\{d_i\}) = prime(A_i) \cup \{d_i\}$ for $i \in inv$, then $prime(A_i)$ must be refused *before* renaming with σ^Q if $\{d_i\}$ is to be refused *after* renaming has been applied (note that $\{d_i\}$ itself is *always* refused before application of the renaming).

Extracting traces

We now define the process *FinalImple*. This will constitute the implementation process supplied to the refinement check in FDR2 which is used to verify whether or not $extr_{EP(Q)}([\![\widehat{Q}]\!]_{SF}) \subseteq [\![P]\!]_{SF}$. This final syntactic transformation is used to "extract" the traces of *PreImple*. (Note that *extract*, *prep* and TE_{inv} are as defined in section 6.4.)

$$FinalImple \triangleq (((PreImple[prep]) \parallel_{prep(A_{inv})} TE_{inv}) \setminus A_{inv})[extract]$$

The behaviours of *FinalImple* are given by the following result.

Lemma 6.14. *The following hold:*

1. $\tau FinalImple = extr_{EP(Q)}(\tau\widehat{Q}) \ \cup \ T$, *where* $T \subseteq \{extr_{EP(Q)}(t) \circ \langle d_i \rangle \mid t \in \tau\widehat{Q} \ \wedge \ d_i \in d_{inv}\}$.

2. $\phi FinalImple = \{(extr_{EP(Q)}(w), X \cup Y) \mid (\exists (w,R) \in \phi_{dom_{EP(Q)}}\widehat{Q})$
   $R \subseteq \alpha\widehat{Q} \ \wedge \ X \subseteq extrFDR^{ref}_{EP(Q)}(R, w, \widehat{Q})$
   $\wedge \ Y \subseteq (\Sigma - (A_{Fvis} \cup d_{inv}))\}$.

6.6.2 Preprocessing the specification process P

Before proceeding, we introduce an additional notation which will prove useful in characterising the behaviours of the preprocessed specification.

Definition 6.16. *Let* $(t,R) \in \phi P$. *Then* $DB(R) \triangleq \{d_i \in d_{inv} \mid B_i \subseteq R\}$.

In addition, we assume the alphabet of P to be as follows:

$$\alpha P = \beta(P) \cup extr^{set}(\alpha\widehat{Q}).$$

As indicated above, it is necessary to modify the specification process P before it is used in any refinement check in FDR2. There are two related reasons for this. Firstly, as indicated by lemma 6.14(1), certain traces from $\tau FinalImple$ may be ended by an event $d_i \in d_{inv}$. It is therefore necessary to add d_i for $i \in inv$ to the end of every trace in the specification. Secondly, refusals in *FinalImple* are effectively defined only in terms of finally visible events and the events from d_{inv}; all other events are always refused. The refusals from the specification therefore have to be modified in order to reflect this fact. These changes are effected using the auxiliary process *Proc*. (Recall that $B_{inv} \triangleq \bigcup_{i \in inv} B_i$ by definition 6.2(2).)

$$\begin{aligned} Proc \ = \ & ((\Box_{a \in \alpha P} a \rightarrow Proc)) \ \Box \ DIV) \ \sqcap \\ & (\Box_{a \in ((\alpha P - B_{inv}) \cup prime(B_{inv}))} a \rightarrow DIV) \ \sqcap \\ & ((\Box_{y \in d_{inv}} y \rightarrow DIV) \ \Box \ DIV). \end{aligned}$$

The events from P are primed using p^P in order to give two copies of each event in B_{inv} — the primed events will be used to define refusals and the unprimed events to define traces — and *Proc* is then composed in parallel with the renamed P, synchronizing on $\alpha P \cup prime(B_{inv})$. We refer to the resulting process as W. For every $(t, R) \in \phi Proc$, $t \in (\alpha P)^*$ and

$$R \subseteq (\Sigma - ((\alpha P - B_{inv}) \cup prime(B_{inv}))).$$

Thus, W will always refuse all events in B_{inv} since $B_{inv} \subseteq extr^{set}(\alpha\widehat{Q}) \subseteq \alpha P$; however, refusals from P which are contained in B_{inv} will appear in a primed form in W. In other words, if $(t, R) \in \phi P$ is such that $R \subseteq B_{inv}$, then $(t, prime(R)) \in \phi W$. Finally, the renaming σ^P is applied to W with the effect that all events from $prime(B_i)$ for $i \in inv$ are renamed to d_i. This means that the new specification process, *NewSpec*, will refuse $\{d_i\}$ for $i \in inv$ after a trace t if and only if P could previously refuse B_i after t (see lemma 6.15(2) and recall the definition of DB in definition 6.16). *NewSpec* is therefore defined as follows.

$$NewSpec \triangleq (P[p^P] \parallel_{\alpha P \cup prime(B_{inv})} Proc)[\sigma^P]$$

NewSpec will constitute the specification process supplied to the refinement check in FDR2 which is used to verify whether or not $extr_{EP(Q)}([\![\widehat{Q}]\!]_{SF}) \subseteq [\![P]\!]_{SF}$. Its behaviours are characterised by the following result.

Lemma 6.15. *The following hold:*

1. $\tau NewSpec = \tau P \cup \{t \circ \langle d_i \rangle \mid t \in \tau P \ \wedge \ d_i \in d_{inv}\}$.
2. $$\begin{aligned}\phi NewSpec = \{(t, R) \mid \ & (\exists (t, X) \in \phi P) \\ & R \subseteq (X \cap (\alpha P - B_{inv})) \cup DB(X) \cup \\ & (\Sigma - ((\alpha P - B_{inv}) \cup d_{inv}))\}\end{aligned}$$

In this result, $\alpha P - B_{inv}$ denotes a combination of finally visible events and "other" events. This reflects the fact that P may engage in events other than those in the specification alphabets of the extraction patterns used to interpret $\widehat{Q}$ (i.e. other than those in $\bigcup_{1 \leq i \leq m} B_i$).

6.6.3 Final result

We now give the final result which lets us verify using FDR2 whether or not $extr_{EP(Q)}([\![\widehat{Q}]\!]_{SF}) \subseteq [\![P]\!]_{SF}$.

Theorem 6.16. *$extr_{EP(Q)}([\![\widehat{Q}]\!]_{SF}) \subseteq [\![P]\!]_{SF}$ if and only if FinalImple $\sqsupseteq_{SF}$ NewSpec.*

6.6.4 Example

We now show how the results given above can be used to define inputs to FDR2 to verify automatically that the extracted behaviours of *LeftImpl* (respectively *RightImpl*) in the stable failures model are contained in the behaviours of *LeftSpec* (respectively *RightSpec*) in the same model.

In both cases — i.e. in the verification of both *LeftImpl* and *RightImpl* — let ep_1 be ep_{ack}. We therefore have that $inv = \{1\}$. Also, $d_{inv} = \{d_1\}$, $A_1 = A_{inv} = \alpha data \cup \alpha ack$ and $B_1 = B_{inv} = \alpha send$. Assuming the existence of suitable extraction patterns to "interpret" events occurring on channel *in* in *LeftImpl* and on channel *out* in *RightImpl*,[23] we have:

- $\alpha RightSpec = \alpha out \cup \alpha send$.
- $\alpha LeftSpec = \alpha in \cup \alpha send$.

We define new channels $data_{prime}$, ack_{prime} and $send_{prime}$ with the same types as *data*, *ack* and *send* respectively on which will occur the necessary primed events. The renaming *prime* which is used here is defined as:[24]

$$\begin{aligned} prime \triangleq\ & \{(send.0, send_{prime}.0), (send.1, send_{prime}.1),\\ & (data.0, data_{prime}.0), (data.1, data_{prime}.1),\\ & (ack.yes, ack_{prime}.yes), (ack.no, ack_{prime}.no)\}. \end{aligned}$$

The renaming p^Q used here is defined as:

$$\begin{aligned} p^Q \triangleq\ & \{(data.0, data.0), (data.0, data_{prime}.0),\\ & (data.1, data.1), (data.1, data_{prime}.1), (ack.yes, ack.yes),\\ & (ack.yes, ack_{prime}.yes), (ack.no, ack.no), (ack.no, ack_{prime}.no)\}. \end{aligned}$$

The renaming p^P used here is defined as:

$$\begin{aligned} p^P \triangleq\ & \{(send.0, send.0), (send.0, send_{prime}.0), (send.1, send.1),\\ & (send.1, send_{prime}.1)\}. \end{aligned}$$

The renaming σ^Q used here is defined as:

$$\begin{aligned} \sigma^Q \triangleq\ & \{(data_{prime}.0, d_1), (data_{prime}.1, d_1), (ack_{prime}.yes, d_1),\\ & (ack_{prime}.no, d_1)\}. \end{aligned}$$

[23]These are the extraction patterns whose existence was assumed in section 6.5.4.

[24]Note that all renamings used here are the same whether we are verifying *LeftImpl* or *RightImpl*.

The renaming σ^P used here is defined as:

$$\sigma^P \quad \triangleq \quad \{(send_{prime}.0, d_1), (send_{prime}.1, d_1)\}.$$

Using the detail above, *NewSpec* could be constructed for both *LeftSpec* and *RightSpec* after taking account of the following important point. In the generic syntactic definitions given in this section, the renaming *prime* is applied to different sets of events. However, renamings may *not* be applied to sets in FDR2 (that approach is used in definitions for ease of expression). In practice, we supply directly the events from the primed set under consideration. For example, where a deterministic choice operator in *Proc* is indexed by $a \in (\ldots \cup prime(B_{inv}))$, we would instead give directly the set of primed events: in the example here, we would use $a \in (\ldots \cup \alpha send_{prime})$ rather than $a \in (\ldots \cup prime(\alpha send))$. In the definition of RE_i in section 6.6.1, a choice operator is indexed by $R \in ref_i^M(t)$ and then a subsequent choice operator is indexed by $a \in prime(A_i - R)$. In practice, we take advantage of the fact that $prime(A_i - R) = prime(A_i) - prime(R)$. The first of the two choice operators is therefore indexed over the set of $prime(R)$ such that $R \in ref_i^M(t)$, where the $prime(R)$ are supplied directly. The second choice operator is then indexed by $a \in prime(A_i) - X$, where $prime(A_i)$ is supplied directly and X represents a member of the set of $prime(R)$ such that $R \in ref_i^M(t)$. This approach is illustrated by the definition of the processes RE_1 below.

Below, $X \triangleq \alpha ack_{prime} \cup \{data_{prime}.0\}$ and $Y \triangleq \alpha ack_{prime} \cup \{data_{prime}.1\}$. These sets are used to represent the *primed* maximal refusals bounds when behaviour is complete (recall that $ref_{ack}(t) = \{R \in 2^{\alpha data \cup \alpha ack} \mid \alpha data \not\subseteq R\}$ for $t \in dom_{ack}$).

Verifying *RightImpl*:

In this case, $Comm(A_1, RightImpl) = Right$. Assume that $\widehat{Q}$ is *RightImpl* after the application of preprocessing as described in section 6.3. Then, as shown in section 6.5.4, $\alpha\widehat{Q} = \alpha data \ \cup \ \alpha ack \ \cup \ \alpha out$. The process RE_1 for the extraction pattern $ep_1 = ep_{ack}$ is defined in terms of two auxiliary processes $RE'_1(x)$ and $RE''_1(x)$:

$$\begin{aligned}
RE_1 \quad &= \quad ((\Box_{x \in \{0,1\}} data.x \rightarrow RE'_1(x)) \Box DIV) \sqcap \\
&\qquad (\sqcap_{R \in \{X,Y\}} (\Box_{y \in ((\alpha data_{prime} \cup \alpha ack_{prime}) - R)} \ y \rightarrow DIV)). \\
RE'_1(x) \quad &= \quad (ack.yes \rightarrow RE_1 \Box ack.no \rightarrow RE''_1(x)) \Box DIV. \\
RE''_1(x) \quad &= \quad (data.x \rightarrow RE_1) \Box DIV.
\end{aligned}$$

Verifying *LeftImpl***:**

In this case, $\mathit{Comm}(A_1, \mathit{LeftImpl}) = \mathit{Left}$. Assume that $\widehat{Q}$ is *LeftImpl* after preprocessing. Then, as shown in section 6.5.4, $\alpha\widehat{Q} = \alpha\mathit{data} \cup \alpha\mathit{ack} \cup \alpha\mathit{in}$. The process RE_1 for the extraction pattern $ep_1 = ep_{ack}$ is defined in terms of the following two auxiliary processes $RE_1'(x)$ and $RE_1''(x)$:

$$\begin{aligned}
RE_1 &= ((\Box_{x\in\{0,1\}}\mathit{data}.x \rightarrow RE_1'(x)) \Box \mathit{DIV}) \sqcap \\
&\quad (\Box_{R\in\{X,Y\}}(\sqcap_{y\in((\alpha\mathit{data}_{prime}\cup\alpha\mathit{ack}_{prime})-R)}\, y \rightarrow \mathit{DIV})). \\
RE_1'(x) &= (\mathit{ack.yes} \rightarrow RE_1 \Box \mathit{ack.no} \rightarrow RE_1''(x)) \Box \mathit{DIV}. \\
RE_1''(x) &= (\mathit{data}.x \rightarrow RE_1) \Box \mathit{DIV}.
\end{aligned}$$

Using the components defined above, along with TE_1, *prep* and *extract* described in section 6.4.3, we were able to define all necessary process expressions needed for the current verification. By supplying them as inputs to FDR2, we were then able to verify automatically that the extracted behaviours of *LeftImpl* (respectively *RightImpl*) in the stable failures model are contained in the behaviours of *LeftSpec* (respectively *RightSpec*) in the same model.

Thus, since both *LeftImpl* and *RightImpl* meet Dom-T-check and Dom-SF-check, since $\alpha\mathit{ImplNet} \subseteq \mathit{Fvis}$ and by theorem 4.10, we may infer that

$$\mathit{ImplNet} \sqsupseteq_{SF} \mathit{SpecNet}.$$

6.7 The failures divergences model

Finally,we show how to verify automatically that $\mathit{extr}_{EP(Q)}([\![\widehat{Q}]\!]_{FD}) \subseteq [\![P]\!]_{FD}$ when $\delta P = \varnothing$.[25] By working under this restriction, the condition that is verified here is similar to the notion of refinement-after-hiding presented in [16]. Before proceeding to the verification of the condition proper, it is necessary to show how to verify condition Ep6.

[25]This restriction is imposed because it lets us verify the condition while still working in the stable failures model: the use of *DIV* in extracting refusals would distort the outcome of any verification check in the failures divergences model. It is a minor restriction in any case, since one would usually expect a (component) specification process to be divergence-free. Moreover, it should be stressed that the specification ***network*** may still contain divergent traces.

Verifying EP6

We recall condition EP6 from section 4.5 and observe that it must be verified for every extraction pattern $ep_i \in EP(Q)$ such that $A_i \cap Fvis = \varnothing$, that is, such that $i \in inv$ (it is met trivially when $i \notin inv$ by EP4-FVI).

EP6 Let $ep \in EP$. If $\ldots, t_j, \ldots$ is an ω-sequence in *Dom*, then $\ldots, extr(t_j), \ldots$ is also an ω-sequence.

In practice, EP6 would be verified of the extraction patterns independently of the actual system verification: in other words, it would be verified of any particular extraction pattern when that extraction pattern was first created. The following result shows how this verification may be carried out, where TE_i is as defined in section 6.4.

Theorem 6.17. *Let $ep_i \in EP(Q)$ be such that $i \in inv$. Then ep_i meets* EP6 *if and only if $\delta(TE_i \setminus A_i) = \varnothing$.*

The verification proper

We assume that $\widehat{Q}$ has already been shown to meet conditions Dom-T-check and Dom-SF-check. As is shown by the following result, two checks are then required in FDR2 to verify that $extr_{EP(Q)}([\![\widehat{Q}]\!]_{FD}) \subseteq [\![P]\!]_{FD}$ when $\delta P = \varnothing$ (*NewSpec* and *FinalImple* are as defined in section 6.6).

Theorem 6.18. *Let $\delta P = \varnothing$ and assume that $\widehat{Q}$ meets conditions Dom-T-check and Dom-SF-check. Then $extr_{EP(Q)}([\![\widehat{Q}]\!]_{FD}) \subseteq [\![P]\!]_{FD}$ if and only if $\delta\widehat{Q} = \varnothing$ and FinalImple $\sqsupseteq_{SF}$ NewSpec.*

Using the above detail, we were able to verify automatically that *LeftImpl* refines-after-hiding *LeftSpec* and *RightImpl* refines-after-hiding *RightSpec* in the failures divergences model. Thus, by theorem 4.12, we may infer that *ImplNet* $\sqsupseteq_{FD}$ *SpecNet*.

6.8 Conclusion

We have presented here a means of automatic verification of our notion of refinement-after-hiding, albeit under certain restrictions.[26] Moreover, it has been built on top of an existing industrial-strength tool, with all of the benefits which that confers. This means of verification is used in the next chapter

[26] The extraction mappings which may be used must be restricted as described in section 6.0.2 and component specification processes must be divergence-free when working in the failures divergences model.

to verify the correctness of an algorithm for asynchronous communication and we postpone until then a more detailed discussion of it.

Chapter 7

Case study

Thus far, we have considered the theory behind notions of refinement-after-hiding in CSP, presented a concrete such notion and described a means of automatically verifying it using a pre-existing tool. The next step is to apply these latter two things in practice. To do this, we attempt to verify the correctness of a particular *asynchronous communication mechanism* or ACM.[1]

7.1 Asynchronous communication

In an ideal world, where we could guarantee instantaneous, atomic[2] data transfer — whatever the type of the data being transferred — shared memory communication between two concurrent processes could be implemented directly using single variables or registers, without any attendant access control policies or mechanisms. However, such atomic data transfers are not possible and if, for example, a reader and writer process were allowed unconstrained access to such a variable or register, interference would occur due to the overlapping of read and write events.

Usually, if communication is to take place between two concurrent processes via a shared memory area, some form of synchronization[3] will be required in order to avoid interference. Such synchronization may take the form of a *critical section* or *handshake communication.* However, this may

[1]Although the purpose of this case study is to apply in practice the machinery developed in previous chapters, it is also intended as an illustration of the general power of refinement-after-hiding: the results given in this chapter are therefore important in their own right.

[2]In this chapter, we shall describe (sequences of) events as "atomic" (with respect to each other) if their occurrences do not overlap in time and so their respective executions cannot interfere with each other.

[3]Synchronization here means that the two communicating processes have to co-ordinate their activities in some way, possibly via a third-party mechanism such as a critical section. It does not refer to communication which is regulated by some sort of global clock.

force one or both of the communicating processes to wait or block while the other completes a data transfer; this may be undesirable, particularly in a real-time environment. Even a buffer, unless of infinite capacity, is not fully asynchronous: if it becomes full, a writer process may have to wait and, if it becomes empty, a reader process may have to wait.

It is to solve this problem that asynchronous communication mechanisms or ACMs have been introduced. Such mechanisms are characterised by the fact that, if used by a single reader and writer, neither the reader nor the writer will ever have to wait before it is allowed to interact with the mechanism. As a result, a writer may always write to an ACM and the reader may always read from it: that is, writes are destructive and may overwrite data already written, while reads are non-destructive and so re-reading is allowed. In order to allow such unconstrained access, despite the reality of non-atomic data transfer, ACMs combine some sort of access logic with multiple data slots. The multiple data slots allow a read and a write to proceed concurrently without interfering with each other, while the access logic ensures the reader and writer processes never access the same slot at the same time. The specific ACM we consider here is Simpson's 4-slot mechanism ([68]).

7.1.1 Simpson's 4-slot mechanism

The software version of the mechanism from [68] is given in figure 7.1; we assume that it will be used to manage data transfer between a single reader and a single writer, which communicate with the mechanism using the read and write procedures respectively. It contains — as the name suggests — four data slots, arranged into two *pairs* of two *slots*. Each of these slots stores a value of type *datatype*[4] and together they constitute a 2-dimensional array, *data*, which is a global variable. The first dimension of the array represents the pair, the second the two slots within that pair. Intuitively, the writer tries to avoid the reader as it seeks to write into the mechanism, while the reader chases after the writer in order to read the last piece of data written.

Three global variables are used in order to manage the behaviour of the reader and writer respectively. These are *latest*, *reading* and *slot*. *latest* is a bit variable indicating the pair to which the writer last wrote, while *slot[i]* tells the reader which slot was last written to in pair *i*. The bit variable *reading* tells the writer the pair from which the reader is about to read or from which it has just read. Note also that *pair* and *index* are local variables.

The behaviour of the reader is relatively straightforward to understand. It ascertains the pair to which the writer last wrote and places this value in

[4]In the general case, *datatype* will be a complex type whose reading and writing are not guaranteed to be atomic by the underlying system on which the 4-slot mechanism is implemented.

```
Global variables:  $reading, latest : bit$
                   $slot : array\ of\ bit$
                   $data : array\ of\ (array\ of\ datatype)$

  procedure  $write$   $(item : datatype);$
             $var$     $pair,\ index : bit;$
             $begin$
                       $pair := not(reading);$
                       $index := not(slot[pair]);$
                       $data[pair, index] := item;$
                       $slot[pair] := index;$
                       $latest := pair;$
             $end;$

  procedure  $read :$  $datatype;$
             $var$     $pair,\ index : bit;$
             $begin$
                       $pair := latest;$
                       $reading := pair;$
                       $index := slot[pair];$
                       $read := data[pair, index];$
             $end$
```

Figure 7.1: Simpson's 4-slot mechanism

the local variable *pair*. It then indicates to the writer that it is going to read from this pair by storing the value in *reading*, before discovering the slot last written to in the pair by interrogating the variable *slot*. Finally, it reads the data item stored in *data* at the relevant pair and slot. Note that the data transfer from *data* which represents this read will *not* occur atomically in the general case.

As indicated above, the writer aims to avoid the slot and pair combination in which the reader finds itself. It first decides to write to the pair in which the reader has *not* indicated an interest via *reading* (we assume that $not(0) = 1$ and $not(1) = 0$). It then decides to write to the slot in that pair which contains the oldest value. This means that it is impossible to immediately overwrite the last data value written into the mechanism. It also means that the writer avoids the reader in the event that the latter is reading from this pair. (This may happen despite the efforts of the writer to choose the alternative pair due to the arbitrary interleaving of the commands contained in the respective read and write procedures.) The relevant data value is then written — non-atomically — into the correct pair and slot combination. *slot* is updated to indicate which slot was written to in the relevant pair before, finally, *latest* is updated to indicate to the reader the pair in which the last write occurred.

As indicated above, a call to the read procedure and a call to the write procedure may proceed concurrently and so the commands they contain can be arbitrarily interleaved. This is obviously necessary if we are to have non-blocking — and so asynchronous — communication. And it is this fact of arbitrary interleaving, along with the fact that data transfers are non-atomic, which leads to the need for verification to ensure that the mechanism does, indeed, behave as desired.

7.2 Verifying the 4-slot mechanism

As hinted above, the 4-slot is intended to mimic the functionality of a register despite the fact that we cannot guarantee atomicity of read and write operations (see figure 7.2 for a procedure-based representation of a register, where it is assumed that the read and write procedures *do* execute atomically). In moving from the register to the 4-slot, a variety of types of reification have occurred:[5]

- *Data reification*: The single memory slot of the register has been replaced by four data slots, along with a number of variables to control access to those slots.

[5]It is exactly this combination of different types of reification, exhibited by a mechanism whose definition is relatively concise, which led us to choose the 4-slot as a case study.

```
Global variable: data : datatype

procedure  write   (item : datatype);
           begin
                   data := item;
           end;

procedure  read :  datatype;
           begin
                   read := data;
           end
```

Figure 7.2: A register

- *(External) behaviour decomposition*: The register transfers data in terms of individual read and write events, while the 4-slot uses a number of different events to implement a read or a write.[6]
- *(External) relaxation of atomicity*: In the register, reads and writes are atomic; in the 4-slot, the read and write procedures may proceed concurrently.

Due to the nature of this reification, standard CSP refinement could not be used to verify that (our CSP representation of) the 4-slot is a correct implementation of (our CSP representation of) the register. Using the concrete notion of refinement-after-hiding from chapter 4, however, we *are* able to show that the 4-slot implements the register and the remainder of this chapter is concerned with doing so. Before proceeding, we look briefly at some other approaches which have been used to verify the correctness of the 4-slot; some of the concepts introduced thereby will be useful in what follows. We will also take advantage of one of the results that has been shown, in order to simplify our verification.

7.2.1 Standard approaches

The standard approach taken in the literature is not to consider correctness with respect to a register. Rather, certain intuitive properties are identified

[6]The 4-slot mechanism given in figure 7.1 and the register from figure 7.2 present the same procedural interface to the outside world and so it does not seem that external behaviour decomposition has occurred. However, as can be seen below in section 7.3, we actually represent the 4-slot as a CSP process in which the events used to implement the read and write procedures are *all* externally visible. The reasons for this are discussed in section 7.3.4.

which it is felt must hold of the 4-slot[7] if it is to work in an acceptable manner, which properties are expressed at the level of abstraction of the 4-slot itself. Arguably the three most important such properties are *data coherence*, *data freshness* and *data sequencing* (see, for example, [17]).

Data coherence Data coherence is preserved if and only if a reader process and a writer process may not simultaneously access the same slot in the 4-slot. It is essentially a mutual exclusion property — only one of the processes may be in a particular slot at any one time — and is used to guarantee that the 4-slot behaves as if data items were actually transferred atomically. This means that reads and writes at the 4-slot level will behave as if they had been ordered atomically in some sequence. In this respect, the requirement for data coherence is similar to that of *serializability* in databases (see, for example, [2]).

Data freshness When a read is executed, it is not enough to guarantee that the value read is a genuine value written into the mechanism; we also need to guarantee that it was written into the mechanism as recently as possible. The property of data freshness is therefore as follows: the oldest value which may be read by a read procedure is that written into the mechanism by the most recent write procedure whose execution had completed by the time at which the execution of this read procedure began.[8]

Data sequencing This condition is concerned with the order in which values are read from the 4-slot. It stipulates that, once we have read a particular value, x, we cannot subsequently read a value, y, which was written into the mechanism earlier than x. Note that data freshness does not imply data sequencing, due to the fact that we can read an "old" value and still meet the data freshness condition.

7.2.2 Checking these conditions

A number of authors have considered the problem of checking these conditions for the 4-slot and have approached it in various ways; moreover, the conditions have all been shown to be met, under the assumption that the

[7]These properties have been used in the verification of a number of different ACMs but we shall concentrate here on the 4-slot.

[8]We cannot simply give the definition as the read procedure must read the last value written into the mechanism: the last-but-one value written may be read if the read procedure begins interrogation of the necessary control variables before the current write has finished updating them after writing a particular data item into the mechanism.

control variables used in the 4-slot will never suffer from *metastability* (see below).

Simpson himself developed and presented the *role-model* approach in [69], which involves dynamically allocating (possibly multiple) 'roles' to the pairs and slots in the array which stores the data written into and read from the 4-slot. On the occurrence of an event used in the implementation of a read or write procedure, the role of any pair or slot may change. A transition system is then constructed, the states of which are given by the roles allocated to the slots and pairs. Model-checking is carried out on this state space in order to determine that the relevant properties hold: for example, if the property of data coherence is met, no reachable state can exist where both reading and writing roles are assigned to the same slot. In his PhD thesis ([17]), Clark set out to present a unified means of checking the correctness of various ACMs (see [17] also for a survey of other approaches to this problem). This was accomplished using an approach involving *Petri Nets* ([55]) and the resulting method was used to successfully verify all three of the above properties for the 4-slot. In [65], Rushby uses model-checking to verify the same properties.

All three of these approaches have in common the fact that they abstract from the data values transmitted by the 4-slot mechanism and specify the properties to be checked independently of these values. Since we verify the 4-slot against a specification *process*, we are unable to employ data abstraction in such a way and have to address squarely the issue of the data values which we shall communicate in our model of the 4-slot (see section 7.3.4).

These three authors also all highlight the issue of whether accesses to the control variables used in the 4-slot algorithm are atomic. The phenomenon of *metastability* (see [17] for an explanation and a list of references) ensures that they are not atomic in the general case. However, Simpson takes an engineering view and states that, in practice, it is possible to design and implement underlying hardware so that metastability is a negligible problem, from which the 4-slot can recover immediately in any case. As a result, he works from the assumption that accesses to control variables are atomic. Both Clark and Rushby carry out verification without this assumption and show that the 4-slot mechanism is not correct. (They relax it in different ways: Clark allows for the possibility of metastability while Rushby assumes that control variables are built using registers which may return any valid value if reads and writes overlap.)[9] The choice which we make in our modelling of the 4-slot is discussed in section 7.3.4.

The recent papers [26, 27] also concern themselves with the fact of verifying that the 4-slot respects the property of data coherence. The first of these

[9]Their results are challenged, however, by the paper [54]: the authors of this paper claim that a more accurate modelling of metastability allows them to show that the 4-slot is, in fact, correct under metastable operation.

uses data refinement in VDM ([33]) to carry out this verification, although it restricts the degree to which the events of the read and write procedures in the 4-slot may be interleaved. The second paper uses a rely-guarantee proof method in conjunction with data refinement and this allows the restrictions from the previous paper to be lifted.

7.3 Modelling the 4-slot in CSP

In order to verify the 4-slot using our notion of refinement-after-hiding, we need to render both it and the register in CSP.[10] In our modelling of the 4-slot, we assume that the problem of metastable operation will not arise (see section 7.3.4 below). Under such an assumption, all of the authors discussed in the previous section have shown that the 4-slot enjoys the property of data coherence. As a result, the mechanism behaves as if it transfers data items atomically and we use this fact to simplify our CSP model: i.e. our model will transmit data items atomically. This simplifies considerably the extraction mapping which is needed to interpret the behaviours of the 4-slot.

7.3.1 The process used

Six basic processes are used in the CSP representation of the 4-slot. There is a process to represent each of the global variables *latest*, *reading* and *slot*, while another process represents the data array. Finally, two processes are used to impose the necessary ordering of event executions.

Figure 7.3 details most of the data types and channels which are required in the construction of the processes which we shall use. Channel *data* is used for (atomic) data transmission: it has fields to indicate whether a read or write operation is occurring, to indicate the pair/slot combination where the data will be written to or read from and, finally, a field to store an integer value from *dataint*.[11] The channels *latest*, *reading* and *slot* are used to communicate with the control variables.[12] Channel *slot* has an additional field to indicate which of the processes is carrying out the relevant action, since both the reader and writer need to read from the global variable *slot*. The operation *not* is defined as *not*(*first*) = *second* and *not*(*second*) = *first*.

[10]The CSP model of the 4-slot is based partly on a model produced by Rod White of Matra BAe Dynamics.

[11]*first* and *second* are used in place of 0 and 1 to indicate a particular pair or slot in order to make process definitions easier to follow.

[12]They are each given the name of the control variable that they are used to communicate with as this makes clearer the connection between the events of the CSP representation of the 4-slot and the events used in the description of the 4-slot in figure 7.1.

- $dataint = \{0..5\}$
- $datatype\ slots\ =\ first \mid second$
- $datatype\ ops = rd \mid wr$
- $datatype\ user\ =\ reader \mid writer$
- $datatype\ dataslot = ops.slots.slots.dataint$
- $channel\ data\ :\ dataslot$
- $channel\ reading,\ latest\ :\ ops.slots$
- $channel\ slot\ :\ user.slots.ops.slots$

Figure 7.3: Data type and channel definitions

The process to represent the variable *reading* is as follows:

- $BitReading = Reading(first)$
- $$\begin{aligned} Reading(x)\ =\ & reading.rd.x \rightarrow Reading(x) \\ & \Box \\ & reading.wr?y \rightarrow Reading(y) \end{aligned}$$

The process to represent the variable *latest* is as follows:

- $BitLatest = Last(first)$
- $Last(x) = latest.rd.x \rightarrow Last(x) \Box latest.wr?y \rightarrow Last(y)$

Figure 7.4: Representing the bit variables *latest* and *reading*

- $SLOT(x,Y) =$

```
let S(y)=
    slot.writer.x.rd.y→S(y)
    □
    slot.writer.x.wr?val→S(val)
    □
    slot.reader.x.rd.y→S(y)
    □
    slot.reader.x.wr?val→S(val)
within S(Y)
```

- $Slots = |||_{x \in \{first,second\}} SLOT(x,first)$

Figure 7.5: Representing *slot*

Figure 7.4 details the processes used to represent the control variables *latest* and *reading* respectively.[13] The representation of the array *slot* can be seen in figure 7.5. It is given by interleaving the two processes *SLOT(first,first)* and *SLOT(second,first)*, each of which represents an element of the array (a similar technique is used to represent the array *data*). Its definition relies on the fact that we may define generic processes which are parameterized by a data value or values indicating a particular position in an array. The channel on which this generic process communicates with the environment is also parameterized with these same values (see section 2.12). Therefore, we may communicate with the process of our choice — i.e. access the desired position in the array — by making sure that we communicate the "identifier/s" for our desired process when we communicate over the relevant channel. For example, the writer is connected to the first position in the array (for the purposes of reading) by "channel" *slot.writer.first.rd.* It is connected to the second position in the array (for the purposes of reading) by *slot.writer.second.rd* and so on. In other words, the parameter x in the definition of *SLOT* denotes the position in the array which is represented by *SLOT*.

The representation of the array *data* also consists of a number of processes, each representing a particular position in the array. The definition of the relevant processes is given in figure 7.6. The process *DataSlot* is parameterized by three values. The first two of these, labelled by x and y, denote the slot in the array which this particular process will represent: we create a *DataSlot* process for every pair and slot combination. For example, *DataSlot(first,second,0)* will be the second slot in the first pair; similarly,

[13]See section 2.12 for an explanation of the syntax used to represent multi-directional communication.

- $DataSlot(x,y,V)=$
  $\quad let\ D(v)=$
  $\qquad data.wr.x.y?val \rightarrow D(val)$
  $\qquad \Box$
  $\qquad data.rd.x.y.v \rightarrow D(v)$
  $\quad within\ D(V)$

- $Data = |||_{x \in \mathcal{A}} DataSlot(fst(x),sec(x),0)$, where $\mathcal{A} \;=\; \{(first, first), (first, second), (second, first), (second, second)\}$, $fst((x, y)) = x$ and $\mathrm{sec}((x, y)) = y$.

Figure 7.6: Representing the data array

DataSlot(second,second,0) is the second slot in the second pair. Each slot in the array — and so each *DataSlot* process — then contains a single integer value, denoted by the variable V (or v in the local definition D).

The processes described so far — namely, *BitReading*, *BitLatest*, *Slots* and *Data* — represent the global variables of the 4-slot mechanism. The final requirement is to provide a communication interface with these processes which reflects the nature of the read and write procedures given in figure 7.1. The processes used for this are given in figure 7.7: their purpose is to impose an ordering on the events offered by the global variables. All of these processes are then composed in parallel, synchronizing on common actions, to give the process *FSlot*, the CSP representation of the 4-slot:

$$((BitReading\ |||\ BitLatest\ |||\ Slots\ |||\ Data)\ \|_A\ Writer)\ \|_B\ Reader$$

where

- $A = \alpha reading.rd\ \cup\ \alpha slot.writer\ \cup\ \alpha data.wr\ \cup \alpha latest.wr.$
- $B = \alpha latest.rd\ \cup\ \alpha reading.wr\ \cup\ \alpha slot.reader\ \cup \alpha data.rd.$

In practice, we use the following equivalent construct to give *FSlot*, in order to avoid the state explosion which would arise from the interleaving of four different processes:

$$((((BitReading\ \|_A\ Writer)\ \|_B\ Reader)\ \|_C\ BitLatest)\ \|_D\ Slots)\ \|_E\ Data$$

where $A = \alpha reading.rd$, $B = \alpha reading.wr$, $C = \alpha latest$, $D = \alpha slot$ and $E = \alpha data$.

Ordering writer-side behaviour:

$$\begin{aligned} \mathit{Writer} = \; & \mathit{reading.rd?p} \rightarrow \\ & \mathit{slot.writer.not(p).rd?i} \rightarrow \\ & \mathit{data.wr.not(p).not(i)?val} \rightarrow \\ & \mathit{slot.writer.not(p).wr.not(i)} \rightarrow \\ & \mathit{latest.wr.not(p)} \rightarrow \mathit{Writer} \end{aligned}$$

Ordering reader-side behaviour:

$$\begin{aligned} \mathit{Reader} = \; & \mathit{latest.rd?p} \rightarrow \\ & \mathit{reading.wr.p} \rightarrow \\ & \mathit{slot.reader.p.rd?i} \rightarrow \\ & \mathit{data.rd.p.i?val} \rightarrow \mathit{Reader} \end{aligned}$$

Figure 7.7: Ordering behaviour of global variables

7.3.2 A simple environment

We also present a simple environment with which *FSlot* might be composed. The purposes of this are twofold. Firstly, it allows us to carry out a basic compositional verification. Secondly, consideration of an environment such as the one we propose is very useful (at least in this case) in determining that the traces extraction mapping we have developed is acceptable: see section 7.4 for a discussion of this issue. Figure 7.8 describes this environment: it may take a value from *dataint* on the channel *in* and write it into *FSlot*; it may also read a value from *FSlot*, before outputting the result on channel *out*. (Note that the channels *in* and *out* used here are assumed to be different to those used in the definition of the processes from the running example in figure 1.1.)

7.3.3 The register and a corresponding environment

The CSP version of the register is presented in figure 7.9. The variable *data* from figure 7.2 is represented as a parameter to the process. Since individual CSP events occur instantaneously and cannot occur concurrently, we are guaranteed to have atomic transfers of data. Figure 7.10 defines the specification environment for which the 4-slot environment is an implementation: it is essentially a pair of single-slot buffers to be placed on the read and write channels of the register. Note that the events on channels *in* and *out* are regarded as finally visible; all other events — i.e. all those in both the register

- $channel\ in,\ out : dataint.$
- $WriteEnviron = in?val \rightarrow reading.rd?p \rightarrow slot.writer.not(p).rd?i \rightarrow data.wr.not(p).not(i).val \rightarrow slot.writer.not(p).wr.not(i) \rightarrow latest.wr.not(p) \rightarrow WriteEnviron$
- $ReadEnviron = latest.rd?p \rightarrow reading.wr.p \rightarrow slot.reader.p.rd?i \rightarrow data.rd.p.i?val \rightarrow out.val \rightarrow ReadEnviron$
- $FourSlotEnviron = WriteEnviron \mid\mid\mid ReadEnviron$

Figure 7.8: An environment for *FSlot*

- $channel\ read, write : dataint$
- $Register = Reg(0)$
- $Reg(x) = read.x \rightarrow Reg(x) \ \Box\ write?y \rightarrow Reg(y)$

Figure 7.9: A CSP version of the register

- $RegWriteEnviron \ = \ in?val \rightarrow write.val \rightarrow \ RegWriteEnviron.$
- $RegReadEnviron \ = \ read?val \rightarrow out.val \rightarrow \ RegReadEnviron.$
- $RegisterEnviron \ = \ RegWriteEnviron \ ||| \ RegReadEnviron.$

Figure 7.10: A corresponding environment for the register

and *FSlot* — are finally invisible.

7.3.4 Issues related to modelling the 4-slot in CSP

We now consider issues relating to some of the choices we have made in modelling the 4-slot mechanism in CSP.

The 4-slot, its environment and inter-process communication

Although the 4-slot algorithm as presented in figure 7.1 implies that reader and writer processes would communicate with the mechanism using (possibly remote) procedure calls, we have taken a different approach with our CSP model. Essentially, we have assumed that all events of the 4-slot are visible to the environment — i.e. both events effecting data transfer and those concerned with manipulating control variables — and that the environment engages in both types of event whenever it wishes to transfer data. It is possible to model the 4-slot and the register in CSP using a procedural interface which gives them both the same set of visible events.[14] However, relaxation of atomicity in the 4-slot means that its procedure invocations and returns may interleave in ways not possible for the register. This means that standard CSP refinement could not be used for verification here and so we need to use refinement-after-hiding. As can be seen in section 7.5, we always extract on the occurrence of events which either write to or read from a control variable, meaning that these events must be visible in our CSP representation of the 4-slot. As a result, it seems we need to see more than would be visible with a procedural interface if verification is to succeed. And since the CSP version of the 4-slot no longer has a procedural interface, there is no reason to retain such an interface in the CSP version of the register.

This of course raises the question of the validity of the results generated here with respect to any real system which might use the 4-slot to transfer data. Although we have not explored this issue formally, we make the following points with respect to any environment with which the 4-slot might

[14]A procedure is modelled externally as an invocation event and a corresponding return event, each of which may communicate data as necessary.

be composed. A procedural interface would be represented in CSP using an event to denote the procedure call and a corresponding event to denote the procedure return. Having made a procedure call, one would assume that the environment would always be ready to receive the return until it actually occurred (this is similar to the property of *receptiveness* which is described in [67]). Moreover, due to the assumption of a single reader and a single writer, the call event of a particular procedure would not be allowed if a return event for that procedure was pending. With our model, it is as if we have substituted for the call and return events all of the events in the relevant procedure: we would do this in general by assuming that the environment would always be ready to accept the next event from a procedure once it had begun to execute; moreover, only one event from a particular procedure would be enabled at any one time. (This is the approach followed in defining the environment *FourSlotEnviron*.)

The extra events added by eschewing a procedural interface would not actually interact at all with any other events in the environment: they would neither enable such events nor cause them to be disabled. Since all such interface events would be hidden anyway in the final network, the change in modelling approach should not have any impact in the traces model and it would certainly fail to introduce any new divergences. In the stable failures model, whichever modelling approach we used, no state between the start and termination of a particular procedure would contribute a stable failure due to offering at least one event which would be hidden in the final network. As a result, whichever modelling approach we used the (CSP) behaviour of any network built using the 4-slot should not change to any significant degree. (See also comments on this issue in section 7.9 at the end of this chapter.)

Instantaneous events but no simultaneous events

In [17], Clark raises the possibility that certain ACMs may execute two events a, b simultaneously with a different result to that which arises by executing either a then b or b then a. This is of significance because we have no means of modelling in CSP the actual concurrent execution of two different events. However, Clark showed that such a problem does not arise with respect to the 4-slot.

Metastability

In section 7.2, we mentioned the problem of metastability in relation to the question of whether or not control variables are modelled as being capable of atomic data transfers. Our CSP model of the 4-slot assumes that such data transfers will be atomic. This is for two main reasons: the first is that we accept Simpson's view that the problem of metastability is negligible in practice (or can be made so). Secondly, we are concerned here with exploring how our notion of refinement-after-hiding may be applied in practice and modelling the possibility of metastability would complicate our model to a large degree.

A concrete data type

For the purposes of verification, specifically the need to use FDR2, it was necessary to choose a concrete data type to be written to and read from both the register and the 4-slot. We have chosen (a subset of) the integers — i.e. $\{0..5\}$ — because they are a basic type. Since we are carrying out model-checking, it is also necessary to choose a finite type and the limits of the hardware on which FDR2 was run dictated the size of the type used. This issue is discussed at greater length in section 7.8, after the presentation of the processes used in the verification.

7.4 Restricting the (traces) extraction mapping

Before proceeding to the verification proper and the derivation of a suitable extraction pattern, it is necessary to consider an important methodological point regarding the use of refinement-after-hiding in practice. Namely, the verification of a particular implementation component may be regarded as ultimately successful only if we are able to verify the correctness of the environment with which the component is to be composed. And the extraction pattern/s used to verify a particular component may have a significant impact on the possibility of successful verification of the environment. This issue is considered in section 7.7.1 with regard to the refusal bounds which are used in the verification of the 4-slot. However, there are properties of sufficient importance that we need to *guarantee* they hold of our extraction pattern (specifically, of the mapping over traces).

In verifying that *FSlot* refines-after-hiding the (CSP) register in the traces model, every trace of *FSlot* has to be mapped to a trace of the register. Since *FSlot* does not engage in any finally visible events, the only restrictions on the mapping used are that it must be strict, monotonic and return a trace over the alphabet of the register when applied to any trace from *FSlot*. As a result, it would be possible to define a mapping which simply returned the empty trace $\langle\rangle$ for any trace to which it was applied. We could then very easily show that the extracted traces of *FSlot* were contained in those of the register. However, such a mapping would cause problems when it came to verifying any meaningful environment with which *FSlot* might be composed. By definition, the mapping used to interpret the traces of *FSlot* will also be used in the interpretation of the traces of the environment and that environment will contain finally visible events in the general case. The presence of these events, along with the fact that finally visible events must be left unaltered by any extraction mapping — see conditions EP4-FVI and TR-GLOBAL2 in chapter 4 — will impose further restrictions on the mapping used in the verification of *FSlot*, which restrictions should be anticipated when that mapping is being developed.

It is possible to define an environment which simply carries out a direct translation to and from the behaviours of *FSlot*: this is the environment presented in figure 7.8.[15] (The corresponding specification environment is given in figure 7.10.) In order for any extraction mapping to allow the successful verification of this environment, the way in which the mapping interprets the traces of *FSlot* must be consistent with the way in which the environment translates to and from those behaviours: this is illustrated by the following discussion.

In the 4-slot environment, every (low-level) write[16] to *FSlot* will be preceded by an event $in.x$; moreover, the low-level write will also transmit the value x. Similarly, every (low-level) read from *FSlot* which transmits the value x will be followed by the event $out.x$. In the specification environment, every event $write.x$ will be preceded by the event $in.x$; similarly, every event $read.x$ will be followed by the event $out.x$. Since the events on channels *in* and *out* are finally visible, they must be unaltered by the application of any extraction mapping. This means that, if verification of the environment is to succeed, each low-level write must extract to $write.x$. Likewise, each low-level read must extract to $read.x$. In other words, each low-level read or write must be extracted to exactly one high-level data transmission; moreover, that high-level data transmission must communicate the same data value as was transmitted by the low-level read or write.

These conditions are therefore required to hold of any mapping we develop here, since we should always expect to be able to verify successfully an environment of the simplicity of that in figure 7.8; moreover, if they do hold it is unlikely that verification of a more complex environment would fail simply because the extraction mapping developed to verify *FSlot* was unsuitable. That they do hold can be checked by attempting to verify the implementation environment from figure 7.8 against the specification environment from

[15]It plays a role similar to that of the extractors and disturbers in [39].

[16]We shall use *low-level write* to mean the execution of the events in *FSlot* which implement a call to the write procedure of the 4-slot; similarly, *low-level read* will be used to mean the execution of the events in *FSlot* which implement a call to the read procedure of the 4-slot. A high-level write will then simply be an event occurring on channel *write*; a high-level read will be an event occurring on channel *read*.

figure 7.10, using the mapping under consideration. It may seem that this can also be checked simply by inspecting the mapping itself. However, as can be seen in the next section, high-level read events are always extracted to *before* the relevant data value has been transmitted at the lower-level. In such a case, it is no longer straightforward to see that the event extracted will transmit the correct data.[17]

7.5 The traces model

We now move on to consider the extraction pattern needed for verification of refinement-after-hiding in the traces model. A single extraction pattern, denoted ep_{ar}, is used to relate the behaviours of *FSlot* to those of the register, where *ar* denotes the fact that we interpret behaviours of an ACM as behaviours of a Register.[18] As a result, $EP(FSlot) = \{ep_{ar}\}$. A_{ar}, B_{ar}, Θ_{ar} and Dom_{ar} for ep_{ar} are defined as follows:

- $A_{ar} = \alpha latest \cup \alpha reading \cup \alpha slot \cup \alpha data$.
- $B_{ar} = \alpha read \cup \alpha write$.
- $\Theta_{ar} = \varnothing$.
- $Dom_{ar} = \tau FSlot$.

First note that all events in A_{ar} are assumed to be ***finally invisible*** — i.e. $A_{ar} \cap Fvis = \varnothing$; we also assume that $Comm(A_{ar}, FSlot) = Left$. Since $EP(FSlot) = \{ep_{ar}\}$, we shall use Dom_{ar} in lieu of $Dom_{EP(FSlot)}$ by TR-GLOBAL1. Note also that, by TR-GLOBAL2, $extr_{EP(FSlot)}$ is equivalent to $extr_{ar}$ in the case that the former is being used to denote the mapping over individual traces. That $Dom_{ar} = \tau FSlot$ means Dom-T-check is met trivially by *FSlot*.[19] It also means that *FSlot* does not need to be preprocessed as described in section 6.3 when we consider automatic verification (recall that

[17] An early version of the extraction mapping derived here was used in a successful verification of *FSlot* but verification of the environment failed for this reason.

[18] A single extraction pattern (and so a single traces mapping) is used to interpret both read and write events because the point at which write events are extracted depends partly on the behaviour of the reader and the point at which read events are extracted depends partly on the behaviour of the writer.

[19] Since $\Theta_{ar} = \varnothing$ and $Comm(A_{ar}, FSlot) = Left$, then Dom-T-check requires that $t \in Dom_{ar}$ for every $t \in \tau FSlot$: i.e. there are no events on which *FSlot* is allowed to go outside the domain. This then means that Dom-T-check does not place any restrictions at all on *FourSlotEnviron*. (See related discussion in section 4.2 with respect to verification of the running example and see also section 7.6.3, where the verification of *FourSlotEnviron* is considered.)

this preprocessing would simply remove all traces of *FSlot* which are not contained in Dom_{ar}).

7.5.1 The extraction mapping

According to the conditions discussed in section 7.4, our choice in defining $extr_{ar}$ is restricted to finding the particular event in each low-level read and write on the occurrence of which we will extract to the relevant high-level event (which extracted event must transmit the same data value as the corresponding low-level event). In addition, that we are mapping traces of *FSlot* to those of the register means any high-level read event to which we extract must transmit the same data value as the last high-level write to which we extracted.

Mimicking the behaviour of the register after application of the mapping is complicated by two main factors (detail on how the relevant situations may arise can be found in section 7.5.2):

- A low-level read may actually read data written into *FSlot* by a low-level write that has not yet completed (that is, it has not yet updated both control variables to fully indicate where it wrote the data).

- The slot and pair from which data is to be read on a particular low-level read may be fully determined before the identity of the relevant slot and pair has been discovered from the control variables.

The first point has the consequence that we cannot always extract to a high-level write event at exactly the point at which the low-level write has completed (i.e. we cannot always extract on the occurrence of the event which updates the variable *latest*). If we were to do this, the reader side may have already read and extracted the value written and, at the specification level, we will get a trace which apparently manages to read a value before it has been written. However, we must also be careful not to extract the current write yet if the reader could still read the value written by the previous write.

As soon as it is fully determined which slot and pair the reader will read from, the value to be read is also fully determined. This is because the writer will not be able to access the relevant slot of the data array until this read has finished, since the 4-slot maintains the property of data coherence. As a result, by the second point above, we may know exactly which value the reader is to read *before* it has completed interrogating the necessary control variables. And we must extract to a high-level read as soon as the value to be read is determined: if we did not do this, the reader could wait until an arbitrary number of further writes had been completed and extracted and only then complete and extract this read. This would give the apppearance

The Writer:

```
        begin
1           $pair := not\ reading$;
2           $index := not\ slot[pair]$;
3           $data[pair, index] := item$;
4           $slot[pair] := index$;
5           $latest := pair$;
        end;
```

The Reader:

```
        begin
1           $pair := latest$;
2           $reading := pair$;
3           $index := slot[pair]$;
4           $read := data[pair, index]$;
        end
```

Figure 7.11: Simpson's 4-slot mechanism annotated

of reading an old value and so of having more memory than the single slot of the register.

Before proceeding, it is also necessary to observe that the event on the occurrence of which we actually extract to a high-level write is not always the same and depends on the way in which low-level reads and writes have been interleaved; a similar comment applies with regard to extraction to high-level read events.

7.5.2 Defining $extr_{ar}$

Figure 7.11 presents an annotated version of the 4-slot mechanism, using numbers to indicate positions within the read and write procedures.[20] These annotations are used both in the presentation of the extraction mapping and in its explanation. Before presenting the mapping, we consider in greater detail the points at which a particular low-level read or write should be extracted.

Considering the writer side in more detail

We cannot extract a write by positions 2 and 3 in the writer, since we do not yet know the value to be written. If the writer is at position 4, it is impossible for the reader to read what has just been written since data coherence is preserved. Finally, we must have extracted once we return to position 1. This means that, if we have not already extracted on the current call to write, we must do so on the occurrence of *latest.wr.not(p)*.

We therefore consider position 5 in the write procedure and the conditions under which we need to have extracted a high-level write event by the time that we reach it: in other words, when do we extract a write event on the occurrence of *slot.writer.not(p).wr.not(i)*. In general, we need to have extracted by this point if the reader already knows, or can discover without any further writer action, the pair into which the writer has just written. If this is the case, the reader can proceed to find out which slot in the pair was written to and so read and extract the value just written. This can happen in the following circumstances:

- If we have already extracted in the reader and the global variable *latest* stores the same value as the variable *pair* in the writer. (The value of *pair* in the writer tells us the pair which the writer has just written to.)
- If we are at position 1 in the reader and the global variable *latest* stores the same value as the variable *pair* in the writer.
- If we are at position 2 or 3 in the reader but have not extracted yet, and the value of *pair* in the reader is the same as the value of *pair* in the writer. (In the corresponding conditional branches in the extraction mapping definition given in figure 7.12, we do not actually state explicitly the requirement that the reader has not yet extracted. This is simply because, if the value of *pair* in the reader is the same as

[20] It is easier to annotate the original definition of the mechanism than the CSP version of it; in any case, the connection between this annotated version and the CSP version should be clear enough.

the value of *pair* in the writer, then the reader cannot have extracted yet. This can be seen from an inspection of the conditions below which let the reader be at either position 2 or 3 and have extracted by that point.[21])

Note that the reader must always have extracted by the time that it reaches position 4.

Considering the reader side in more detail

Recall that we will extract to a high-level read event as soon as we are certain of the pair and slot combination from which we will read on the current call to read.

By position 2 in the reader, we know the pair we must read from. In order for it to be fully determined by this point the slot from which we will read, it has to be the case that the writer is unable to write again to this pair before we have completed the current read. (If the writer could write to this pair again, it would first write to the other slot of the pair, to which element the reader could then be directed.) If the writer is to be unable to write to this pair, it is necessary that the value of *pair* in the reader is the same as the value of *reading*. We therefore have to have extracted a read by position 2 in the reader — that is, extracted on the occurrence of *latest.rd.p* — in the following circumstances:

- If the writer is at position 1 or position 5, and *pair* in the reader has the same value as *reading*.
- If the writer is at positions 2, 3 or 4, the value of *pair* in the writer is *not* the same as the value of *pair* in the reader and *pair* in the reader has the same value as *reading*.

In order to check these conditions in practice, we would use the value stored in *latest* in place of that stored in *pair* in the reader: the conditions

[21]First note that the decision on whether or not we will extract on the occurrence of *slot.writer.not(p).wr.not(i)* is taken when the writer is at position 4. By the detail on extracting read events, we consider each of two cases in which the reader may have already extracted and be at either position 2 or position 3. In the first case, the writer is at either position 1 or position 5 and the value of *pair* in the reader is the same as the value of *reading* when the extraction occurs. As a result, by the time that the writer reaches position 4 on this or any subsequent call to write (while the reader is still at position 2 or position 3), it will have set the value of *pair* in the writer to the "negation" of *reading* and so to the "negation" of *pair* in the reader. A similar argument applies in the second case, when the writer is at position 2, 3 or 4 when the extraction of the read event occurs, except that here we start out with the fact that *pair* in the reader does not have the same value as *pair* in the writer.

must be checked at position 1, when *pair* has not yet been updated with the value of *latest.*

By position 3, we know the pair we will read from and have also indicated this to the writer. We have to have extracted by position 3 if the writer is at position 1 or position 5 or if the value of *pair* in the reader is not the same as the value of *pair* in the writer. These conditions are essentially the same as those given for position 2, when we bear in mind the fact that we have just assigned the value of *pair* in the reader to *reading.*

Finally, we must always have extracted by position 4 since, at this point, we know both the slot and pair of the data item which we shall read.

It can be seen from the above discussion that the position of the writer plays a role in whether or not we extract a read event. And, in fact, the writer *moving* to position 5 may necessitate the extraction of a read event. This means that the event $slot.writer.not(p).wr.not(i)$ will, in some cases, be extracted to both a read *and* a write event. This can be seen in the definition of the extraction mapping in figure 7.12.

The mapping

We now proceed to define $extr_{ar}$. Before giving the definition of this mapping, it is necessary to introduce some auxiliary notation.

- For any trace $t \in \tau FSlot$:
  - we take $exR(t) = yes$ if and only if we have already extracted a *read* event during the current call to read and take $exR(t) = no$ otherwise.
  - we take $exW(t) = yes$ if and only if we have already extracted a *write* event during the current call to write and have $exW(t) = no$ otherwise.
- *late* gives the current value stored by the control variable *latest.*
- *rp* gives the current value of the variable *pair* in the reader and *wrp* gives the value of the variable *pair* in the writer.
- *rPos* gives the current position of the reader and *wPos* gives the current position of the writer.
- *rdng* gives the value currently stored in the variable *reading.*
- $slotVal[i]$ gives the value currently stored at position i in the array *slot.*
- *wVal* gives the last value written into the mechanism.

- $rVal[i][j]$ gives the data value stored by the mechanism in pair i, slot j.

We then have that $extr_{ar}(\langle\rangle) \triangleq \langle\rangle$ and, for $t \circ \langle a\rangle \in \tau FSlot$,

$$extr_{ar}(t \circ \langle a\rangle) \triangleq extr_{ar}(t) \circ u,$$

where u is as defined in figure 7.12.

A brief comment is required on the clauses used for the extraction of write events, since at first sight some of them may not appear to be mutually exclusive. That they are mutually exclusive follows from the fact that, if the reader is at position 1, then it cannot yet have extracted, and, as observed above, the reader cannot yet have extracted if the value of *pair* in the reader is the same as the value of *pair* in the writer.

7.6 Automatic verification in the traces model using FDR2

We now move on to consider how we may verify automatically — using FDR2 and the approach of chapter 6 — that $extr_{EP(FSlot)}(\tau FSlot) \subseteq \tau Register$. The first step is to represent the traces mapping $extr_{ar}$ as a CSP process and to define the renamings which are also needed for the verification. This detail is given in appendix D. Before looking at that chapter, the reader is advised to first read the following comments on deriving extraction mappings.

7.6.1 Deriving extraction mappings

We comment on the methodology used to develop the extraction mapping $extr_{ar}$. Due to the complexity of the mapping required, itself a consequence of the complexity of the behaviours of *FSlot*, it is virtually impossible to look at any candidate mapping and make a decision on its suitability solely by inspection. As a result, verification in FDR2 played an integral role in determining the mapping to be used: essentially, when a mapping was developed which allowed us to successfully verify both *FSlot* and its environment then that was the mapping to be used. In other words, the (CSP version of the) mapping was partly the outcome of a process of trial and error: verification was attempted using a particular mapping, verification failed, debugging information was inspected to find the cause of the failure, the mapping was modified, verification was attempted again. When verification succeeded, we had our mapping.

Although we were in possession of some of the intuition given above to explain $extr_{ar}$ *before* embarking on the verification, a large part of that insight

$$u \triangleq \begin{cases}
\langle write.wVal \rangle & \text{if } (a = slot.writer.x.wr.y) \wedge \\
 & (rPos = 1 \ \wedge \ late = wrp) \\[1ex]
\langle write.wVal \rangle & \text{if } (a = slot.writer.x.wr.y) \wedge \\
 & (exR(t) = yes \ \wedge \ late = wrp) \\[1ex]
\langle write.wVal \rangle & \text{if } (a = slot.writer.x.wr.y) \wedge \\
 & (rPos = 2 \ \vee \ rPos = 3) \wedge \\
 & (rp = wrp \ \wedge \ rp \neq rdng) \\[1ex]
\langle write.wVal \rangle & \text{if } (a = latest.wr.x) \wedge \\
 & (exW(t) = no) \\[1ex]
\langle write.wVal, read.wVal \rangle & \text{if } (a = slot.writer.x.wr.y) \wedge \\
 & (rPos = 2 \ \vee \ rPos = 3) \wedge \\
 & (rp = wrp \ \wedge \ rp = rdng) \\[1ex]
\langle read.(rVal[x][slotVal[x]]) \rangle & \text{if } (a = latest.rd.x) \wedge \\
 & (wPos = 1 \ \vee \ wPos = 5) \wedge \\
 & (late = rdng) \\[1ex]
\langle read.(rVal[x][slotVal[x]]) \rangle & \text{if } (a = latest.rd.x) \wedge \\
 & (wPos = 2 \ \vee \ wPos = 3 \vee \\
 & \ wPos = 4) \wedge \\
 & (wrp \ ! = late \ \wedge \ late = rdng) \\[1ex]
\langle read.(rVal[x][slotVal[x]]) \rangle & \text{if } (a = reading.wr.x) \wedge \\
 & (exR(t) = no) \wedge \\
 & (wPos = 1 \ \vee \ wPos = 5 \vee \\
 & \ wrp \ ! = rp) \\[1ex]
\langle read.(rVal[x][y]) \rangle & \text{if } (a = slot.reader.x.rd.y) \wedge \\
 & (exR(t) = no) \\[1ex]
\langle \rangle & \text{otherwise}
\end{cases}$$

Figure 7.12: Defining $extr_{ar}$

was provided by working with FDR2. The definition of $extr_{ar}$ given in figure 7.12 was then derived from the process used in the verification and this has a most important consequence. It may not be immediately clear to the reader that the process TE_{ar} from appendix D accurately encodes the mapping $extr_{ar}$[22], which may in turn have cast doubt on the validity of the verification presented here. However, such a thing does not matter: by definition, TE_{ar} — after restriction to the appropriate domain — encodes the mapping used in the (successful) verification and the definition of $extr_{ar}$ in figure 7.12 may best be viewed as an attempt to present that mapping in a more easily understandable form.[23] On a related point, the intuition given to explain $extr_{ar}$ is not intended to be complete in the sense that it fully defines the mapping; as indicated above, it is partly an attempt to explain after the fact the mapping which verification indicated was suitable.

7.6.2 The CSP version of $extr_{ar}$ and applying it to $\tau FSlot$

The reader should now read appendix D. The process TE_{ar} is used, along with the renamings $prep_{ar}$ and $extract_{ar}$, to encode the mapping $extr_{ar}$ and so, by TR-GLOBAL2 and TR-DEF1, to encode the application of $extr_{EP(FSlot)}$ to $\tau FSlot$. *ExtFS* is used to denote *FSlot* after the application of the extraction mapping and we have

$$ExtFS \triangleq ((FSlot[prep_{ar}] \parallel_{prep_{ar}(A_{ar})} TE_{ar}) \setminus A_{ar})[extract_{ar}]$$

(Recall that $A_{ar} = \alpha data \cup \alpha reading \cup \alpha slot \cup \alpha latest$ and also that it is not necessary to preprocess *FSlot* prior to renaming with $prep_{ar}$ since $Dom_{ar} = \tau FSlot$.)

Extraction to non-singleton traces

The means of automatic verification presented in chapter 6 assumes that we can extract to at most one high-level event on the occurrence of any individual low-level event. Here, however, we do extract to more than a single event in a particular case.[24] We first discuss in more detail the problem which leads

[22] Note that the mapping represented by TE_{ar} has a domain larger than Dom_{ar}; it is assumed that it represents $extr_{ar}$ once it has been restricted to the domain Dom_{ar}, which restriction will be effected by composition in parallel with $FSlot[prep_{ar}]$ during verification.

[23] The author *is* convinced, however, that it does accurately reflect the mapping encoded by TE_{ar}!

[24] We say that an extraction mapping, *extr*, extracts to ***non-singleton*** traces if there is at least one trace $t \circ \langle a \rangle$ such that $extr(t \circ \langle a \rangle) = extr(t) \circ u$ and $|u| \geq 2$. If there is not at least one such trace then we say that the mapping extracts only to ***singleton*** traces.

to the need for this restriction, before showing that it does not arise in the verification under consideration here.[25]

During verification in the general case of an implementation process Q, we define a number of different processes TE_j such that $j \in inv$. Consider the case, for $i \in inv$, that $t \circ \langle a \rangle \in Dom_i$ and $extr_i(t \circ \langle a \rangle) = extr_i(t) \circ u$, where u is a non-singleton trace. In TE_i, we would therefore have a trace $v \circ \langle b_1 \rangle \circ \ldots \circ \langle b_k \rangle$, where $domain(v) = t$, $domain(v \circ \langle b_1 \rangle \circ \ldots \circ \langle b_k \rangle) = t \circ \langle a \rangle$ and $extract(\langle b_1 \rangle \circ \ldots \circ \langle b_k \rangle) = u$ (b_h for $1 < h \leq k$ would represent an event pair with a null *left-hand* component and b_1 would represent an event pair with a as the left-hand component). Using TE_i and the other TE_j such that $j \in inv$, we build TE_{inv}. Since the sets of events in which TE_i and TE_j may engage are disjoint for $i \neq j$, we cannot guarantee that $\langle b_1 \rangle \circ \ldots \circ \langle b_k \rangle$ will always execute atomically in TE_{inv} and it is extremely unlikely that it will: the events from the TE_j for $i \neq j$ will interleave with it in an arbitrary fashion. (In fact, $\langle b_1 \rangle \circ \ldots \circ \langle b_k \rangle$ may not even execute atomically in TE_i, depending on how TE_i is defined syntactically.) This may have the result that there exists $w \in \tau TE_{inv}$ such that $domain(w) \in Dom_{inv}$ but $extract(w \setminus A_{inv}) \neq extr_{EP(Q)}(domain(w))$: a will occur somewhere in $domain(w)$ where its occurrence should be extracted to u, while $b_1, \ldots, b_k$ may not occur consecutively in w and so the events of u may be distributed across a number of other events in $extract(w \setminus A_{inv})$. Even if $\langle b_1 \rangle \circ \ldots \circ \langle b_k \rangle$ executes atomically in TE_{inv}, it may not do so in $\widehat{Q}[prep] \parallel_{prep(A_{inv})} TE_{inv}$: if $\widehat{Q}$ contains finally visible events, these may interleave arbitrarily with $\langle b_1 \rangle \circ \ldots \circ \langle b_k \rangle$ and a similar problem will arise. This is why only extraction to singleton traces is allowed in the general case when we verify refinement-after-hiding using the approach from chapter 6. However, if we can guarantee that all such sequences $\langle b_1 \rangle \circ \ldots \circ \langle b_k \rangle$ *will* execute atomically in TE_{inv} and also in $\widehat{Q}[prep] \parallel_{prep(A_{inv})} TE_{inv}$ then it *is* acceptable to allow extraction to non-singleton traces.

The extraction to a non-singleton trace which is used in the verification here is effected by the use of *extractWriteSlotRead?x?y?val* followed by *extra.val* in the definition of *WrExt* in figure D.4 in the appendix. And we can always guarantee in this restricted case that, for arbitrary values c, d, e, $\langle extractWriteSlotRead.c.d.e, extra.e \rangle$ *will* execute atomically whenever it occurs. This is for the following reasons.[26] In the composition in parallel of *RdExt* and *WrExt*, it is immediate that no other event but *extra.e* will be enabled after the execution of *extractWriteSlotRead.c.d.e*: see the way in which the two events appear in *RdExt* (see figure D.5 in the appendix) and

[25]We refer now to the detail in section 6.4.

[26]Recall that TE_{ar} is built by composing *WrExt* with *RdExt*, the result with *EDATA* and the result of that composition with *SlotCopy*, synchronizing on shared events in each composition.

note also that these processes have to synchronize on $\alpha extractWriteSlotRead$. After all further parallel compositions necessary to give TE_{ar} and then $(FSlot[prep_{ar}] \parallel_{prep_{ar}(A_{ar})} TE_{ar})$, it still holds that only $extra.e$ will be enabled once we have executed $extractWriteSlotRead.c.d.e$. This is because, in each of these further compositions, synchronization occurs on all events in which the new process to be added can engage: for example, $FSlot[prep_{ar}]$ cannot engage in any events outside of $prep_{ar}(A_{ar})$. As a result, we have that $\langle extractWriteSlotRead.c.d.e, extra.e \rangle$ always executes atomically whenever it occurs and so it is acceptable to allow extraction to a non-singleton trace in this case.[27]

The final verification

In view of the points made in section 6.4.4, we observe that TE_{ar} is deterministic[28] and also that it is acceptable that TE_{ar} defines a mapping which has a domain strictly larger than $\tau FSlot$[29]: the domain of the mapping will be restricted as necessary by composition with $FSlot[prep_{ar}]$. By the above and the detail in section 6.4, we therefore have that $\tau ExtFS = extr_{EP(FSlot)}(\tau FSlot)$. As a result, we are able to show that $extr_{EP(FSlot)}(\tau FSlot) \subseteq \tau Register$ by verifying in FDR2 that $ExtFS \sqsupseteq_T Register$. Due to the successful outcome of this verification and the fact that Dom-T-check is met by $FSlot$, we have that $FSlot$ refines-after-hiding the register in the traces model.

[27]Even if this trace were not to execute atomically as described above, it would simply mean that $extr_{EP(FSlot)}(\tau FSlot)$ was a strict subset of $\tau ExtFS$. Provided that verification is successful — as it is here — we would still get the result that we desire on containment of $extr_{EP(FSlot)}(\tau FSlot)$ in $\tau Register$. In other words, once the requirement on extracting to singleton traces is relaxed, successful verification using this approach is only a ***sufficient*** indicator that the extracted traces of a particular implementation are contained in those of the corresponding specification rather than a ***necessary*** condition of it. A similar point also applies with respect to the verification of the environment which is described in section 7.6.3.

[28]FDR2 has been used to verify the determinism of *RdExt*, *WrExt*, *EDATA* and *SlotCopy*; moreover, parallel composition which synchronizes on common events cannot introduce non-determinism.

[29]That TE_{ar} does define a mapping which has a domain larger than $\tau FSlot$ has been verified in FDR2, using the renaming $domain_{ar}$ defined in figure D.9 in the appendix. In actual fact, we showed that the following holds:

$$\tau FSlot \subseteq \tau(((FSlot[prep_{ar}] \parallel_{prep_{ar}(A_{ar})} TE_{ar}) \setminus \alpha extra)[domain_{ar}]).$$

In other words, $\tau FSlot$ is contained in the domain of the mapping defined by TE_{ar} once that domain has been restricted as necessary by composition with $FSlot[prep_{ar}]$.

7.6.3 Verifying the environment

As indicated in section 7.4, it is necessary to use $extr_{ar}$ in the verification of the 4-slot environment in order to be sure that it ($extr_{ar}$) is acceptable. In actual fact, we will show that *FourSlotEnviron* refines-after-hiding in the traces model *RegisterEnviron*.

Since $Comm(A_{ar}, FSlot) = Left$, $Comm(A_{ar}, FourSlotEnviron) = Right$. Since *FourSlotEnviron* may engage in finally visible events — these are the events on channels *in* and *out* — it is necessary to assume the existence of an extraction pattern ep', where $A' = \alpha in \cup \alpha out$. We shall use *EP* as a shorthand for $EP(FourSlotEnviron) = \{ep_{ar}, ep'\}$.

Verifying Dom-T-check Since $\Theta_{ar} = \varnothing$ and $\alpha in \cup \alpha out \subseteq Fvis$, we have by definition 4.7 in chapter 4 that $Proj_{EP} = A_{ar}$. As a result, Dom-T-check is equivalent to the following:

If $t\lceil A_{ar} \in Dom_{EP}\lceil A_{ar}$ for every $t \in \tau FourSlotEnviron$, then $t \in Dom_{EP}$.

That this holds is immediate by TR-GLOBAL1, the fact that $Dom_{EP}\lceil A_{ar} = Dom_{ar}$ and the fact that $Dom' = A'^{*}$.

Extracting traces In order to extract the traces of *FourSlotEnviron*, it is first necessary to preprocess it as described in section 6.3. This is done by composing *FourSlotEnviron* in parallel with *FSlot*, synchronizing on A_{ar}. (Recall that $Dom_{ar} = \tau FSlot$; moreover, *FSlot* has been shown to be deterministic using FDR2.) The resulting process is denoted *ModEnv*. Since ep' is used to "interpret" finally visible events, it is not necessary to construct a process *TE* to represent the extraction mapping which it contains. We therefore construct the following process:

$$ExtEnv \triangleq ((ModEnv[prep_{ar}] \parallel_{prep_{ar}(A_{ar})} TE_{ar}) \setminus A_{ar})[extract_{ar}]$$

According to the discussion in section 7.6.2, we have to show that, for arbitrary c, d, e, $\langle extractWriteSlotRead.c.d.e, extra.e\rangle$ executes atomically whenever it occurs in $ModEnv[prep_{ar}] \parallel_{prep_{ar}(A_{ar})} TE_{ar}$. We already know that it executes so in TE_{ar}: this means that $extra.e$ is the only event enabled in TE_{ar} after we have executed $extractWriteSlotRead.c.d.e$. And since we synchronize on $prep_{ar}(A_{ar})$, no events from $prep_{ar}(A_{ar})$ will be enabled in $ModEnv[prep_{ar}] \parallel_{prep_{ar}(A_{ar})} TE_{ar}$ immediately after the execution of $extractWriteSlotRead.c.d.e$. Moreover, no finally visible events will be enabled then either: when $extra.e$ is enabled, the environment described by *ModEnv* must be in the middle of a call to read and in the middle of a call to write, while finally visible events are only enabled (in the environment) when there are no outstanding calls to either read or write. It follows that $\langle \mathit{extractWriteSlotRead.c.d.e}, \mathit{extra.e} \rangle$ does always execute atomically in this case. However, as discussed in section 7.6.2, even if we could not guarantee this it would not matter provided that the necessary verification was successful, as it is here.

On the basis of the detail in section 7.6.2 and that given in section 6.4, we conclude that $\tau \mathit{ExtEnv} = \mathit{extr}_{EP}(\tau \mathit{ModEnv})$ and so $\tau \mathit{ExtEnv} = \mathit{extr}_{EP}(\tau \mathit{FourSlotEnviron})$ by TR-DEF1.

Refinement-after-hiding Using FDR2, we were able to successfully verify that $\mathit{ExtEnv} \sqsupseteq_T \mathit{RegisterEnviron}$. This means that

$$\mathit{extr}_{EP}(\tau \mathit{FourSlotEnviron}) \subseteq \tau \mathit{RegisterEnviron}$$

and so *FourSlotEnviron* refines-after-hiding *RegisterEnviron* in the traces model.

Compositional verification

We observe that $\alpha(\mathit{FSlot} \otimes_{A_{ar}} \mathit{FourSlotEnviron}) \subseteq \mathit{Fvis}$. Thus, it follows by theorem 4.8 from section 4.2 that:

$$(\mathit{FSlot} \otimes_{A_{ar}} \mathit{FourSlotEnviron}) \sqsupseteq_T (\mathit{Register} \otimes_{B_{ar}} \mathit{RegisterEnviron}).$$

Despite the fact that they have very different (trace) behaviours, this result illustrates that *FSlot* is a valid (trace) implementation of a register when placed in a simple environment. This is a non-trivial result in the sense that the composition of *FSlot* with the environment does not simply deadlock or refuse to do anything: recall that calls to read and write are non-blocking (in *FSlot*). (This latter issue is treated more formally in the next section.)

7.6.4 A comment on compositionality

So far in this chapter, we have shown how *FSlot* and *FourSlotEnviron* may be verified using refinement-after-hiding and have then inferred that $\mathit{FSlot} \otimes_{A_{ar}} \mathit{FourSlotEnviron}$ refines $\mathit{Register} \otimes_{B_{ar}} \mathit{RegisterEnviron}$ in the traces model according to standard CSP refinement. That we show this compositionally — i.e. by treating separately the verification of *FSlot* and that of *FourSlotEnviron* — is not something that would have been possible with standard CSP refinement. Indeed, it is the additional degree of compositionality which refinement-after-hiding allows in comparison to standard CSP refinement which is the main benefit provided by the former over the latter.

In this case, however, it does not seem that this additional compositionality gives much of a benefit and, indeed, it is simple enough to verify directly using FDR2 that $FSlot \otimes_{A_{ar}} FourSlotEnviron$ refines $Register \otimes_{B_{ar}} RegisterEnviron$ according to standard CSP refinement. However, there are three points to be made with respect to this.

Firstly, we verify the composition $FSlot \otimes_{A_{ar}} FourSlotEnviron$ simply in order to show how a compositional verification using refinement-after-hiding might proceed. In practice, *FSlot* could be composed with a much bigger process, where direct verification using FDR2 and standard CSP refinement might be impossible due to the problem of state explosion. Moreover, it need not be the case that the composition of *FSlot* with the component process into which it is to be embedded will result in a process which engages only in finally visible events: it may be that the interface between this process and the rest of the implementation network under consideration also needs to be interpreted using refinement-after-hiding. Finally, the use of refinement-after-hiding allows *FSlot* to be verified in isolation: the direct use of standard CSP refinement would mean the duplication of effort, as *FSlot* would effectively be re-verified during the verification of each implementation network of which it was a component process.

7.7 The stable failures and failures divergences models

We now move on to the verification of refinement-after-hiding in the stable failures and failures divergences models. In order to proceed, it is first necessary to define the extraction pattern components dom_{ar} and ref_{ar} (we assume that the extraction pattern used is still denoted ep_{ar} and that A_{ar}, B_{ar}, Θ_{ar} and $extr_{ar}$ remain as before). The first of these is defined as follows:

$$\begin{aligned} dom_{ar} \;=\; & \{t \in \tau FSlot \mid (\exists x, y \in \{first, second\}) \\ & t \circ \langle reading.rd.x\rangle \in \tau FSlot \;\wedge\; t \circ \langle latest.rd.y\rangle \in \tau FSlot\}. \end{aligned}$$

In other words, behaviours are "complete" only if there is no call to either read or write currently outstanding. It is easy to see that Dom_{ar} is still given by $\tau FSlot$ even now that we have to define it as the prefix-closure of dom_{ar}.

Defining refusal bounds Two factors inform the choice of refusal bounds to be used here, one a condition to be met by any set of refusal bounds, the other based on more practical concerns. Since *FSlot* is deterministic, we have that

$$\phi FSlot = \{(t,R) \mid t \in \tau FSlot \wedge R \subseteq \{a \in A_{ar} \mid t\text{o}\langle a\rangle \notin \tau FSlot\} \cup (\Sigma - A_{ar})\}.$$

By this, EP5 and since $Dom_{ar} = \tau FSlot$, then, where $(t,R) \in \phi FSlot$, $X \cup (R \cap A_{ar}) \in ref_{ar}(t)$ for every $X \in ref_{ar}(t)$. This effectively gives us a lower bound on the size of the refusal bounds to be used. It is then sensible to make our refusal bounds as small as possible while still allowing *FSlot* to be successfully verified. This is simply because the smaller the bounds contained in ref_{ar}, the less restrictive those which appear in $\overline{ref}_{ar}$ (see definition 4.9). This means that the conditions imposed on any environment if it is to be verified successfully will also be less restrictive (see section 7.7.1 below). As a result, for $t \in Dom_{ar} = \tau FSlot$, we take

$$ref_{ar}(t) \triangleq \{R \cap A_{ar} \mid (t,R) \in \phi FSlot\},$$

meaning that *FSlot* will never breach any bound from ref_{ar}.[30]

Verifying Dom-SF-check Since $EP(FSlot) = \{ep_{ar}\}$, $\alpha FSlot = A_{ar}$ and $Dom_{ar} = Dom_{EP(FSlot)} = \tau FSlot$, Dom-SF-check reduces to the following:

- Let $(t,R) \in \phi FSlot$, where $R \subseteq A_{ar}$. If $extr^{ref}_{ar}(R,t,FSlot) = B_{ar}$ then $t \in dom_{ar}$.

By definition 4.10, $extr^{ref}_{ar}(R,t,FSlot) = \varnothing$ for every $(t,R) \in \phi FSlot$ where $R \subseteq A_{ar}$ (recall that $Comm(A_{ar}, FSlot) = Left$). As a result, Dom-SF-check is met trivially.

Extracting failures We then have that the following result holds.

Proposition 7.1. *$extr_{EP(FSlot)}(\phi FSlot) \subseteq \phi Register$.*

Proof. Let $(t,R) \in \phi_{dom_{EP(FSlot)}} FSlot$ be such that $R \subseteq \alpha FSlot = A_{ar}$. Then $extr^{ref}_{EP(FSlot)}(R,t,FSlot) = extr^{ref}_{ar}(R,t,FSlot) = \varnothing$ by SF-GLOBAL2, definition 4.10 and since $EP(FSlot) = \{ep_{ar}\}$. Moreover, since $dom_{ar} = dom_{EP(FSlot)}$ by SF-GLOBAL1 and $dom_{ar} \subseteq \tau FSlot = \{t \mid (t,R) \in \phi FSlot\}$, we have that:

$$dom_{ar} = \{t \mid (t,R) \in \phi_{dom_{EP(FSlot)}} FSlot\}.$$

[30]By EP5, it must be the case that each set in $ref_{ar}(t)$ is a *proper* subset of A_{ar}; that this is the case here follows from the fact that *FSlot* is deadlock-free (verified in FDR2 and follows automatically from the definition of *FSlot* anyway) and always refuses all events from $\Sigma - A_{ar}$.

Hence, by SF-DEF2,

$$extr_{EP(FSlot)}(\phi FSlot) = \{(extr_{EP(FSlot)}(t), Y) \mid t \in dom_{ar} \wedge Y \subseteq (\Sigma - B_{ar})\}.$$

We observe that $\tau Register = \{t \mid (t, \varnothing) \in \phi Register\}$ by proposition 2.3(2) and since $\delta Register = \varnothing$. Since $dom_{ar} \subseteq \tau FSlot$ and $extr_{EP(FSlot)}(\tau FSlot) \subseteq \tau Register$ (see section 7.6.2), then, by TR-DEF1:

$$\{(extr_{EP(FSlot)}(t), \varnothing) \mid t \in dom_{ar}\} \subseteq \phi Register.$$

That $extr_{EP(FSlot)}(\phi FSlot) \subseteq \phi Register$ follows by SF4 and since $B_{ar} = \alpha read \cup \alpha write$. □

Refinement-after-hiding We observe that

$$extr_{EP(FSlot)}([\![FSlot]\!]_{SF}) \subseteq [\![Register]\!]_{SF}$$

since $extr_{EP(FSlot)}(\tau FSlot) \subseteq \tau Register$ and by SF-DEF1 and proposition 7.1. Moreover, *FSlot* meets conditions Dom-T-check and Dom-SF-check. Hence, by SF-DEF3, *FSlot* refines-after-hiding the register in the stable failures model. That it does so in the failures divergences model follows by FD-DEF1, FD-DEF4 and the following five points:

- $extr_{EP(FSlot)}(\phi FSlot) \subseteq \phi Register$ and *FSlot* meets Dom-T-check and Dom-SF-check.
- By DR2, $\phi Register \subseteq \phi_\perp Register$.
- By FD-DEF2 and since $\delta FSlot = \varnothing$[31], then $extr_{EP(FSlot)}(\delta FSlot) = \varnothing$.
- $extr_{EP(FSlot)}(\phi_\perp FSlot) = extr_{EP(FSlot)}(\phi FSlot)$ by FD-DEF3 and since $extr_{EP(FSlot)}(\delta FSlot) = \varnothing$.
- $extr_{ar}$ meets EP6. (This has been verified using FDR2.[32])

Automatic verification using FDR2 is not needed in general here because of the nature of *FSlot* and of the refusal bounds used. In addition, the fact that *FSlot* gives the only definition of Dom_{ar} which we have would have complicated the definitions of the processes *DSF* and *RE* needed for verification here.

[31] All component processes used to build *FSlot* are guarded and so divergence-free by DF; moreover, parallel composition cannot introduce divergence. That $\delta FSlot = \varnothing$ has also been verified using FDR2.

[32] Note that we verify it by checking for the divergence-freeness of $(FSlot[prep_{ar}] \parallel_{prep_{ar}(A_{ar})} TE_{ar}) \setminus A_{ar}$ rather than the divergence-freeness of $TE_{ar} \setminus A_{ar}$.

7.7.1 Refusal bounds and environments

The refusal bounds defined here will place constraints on the form of any environment which may be successfully verified and with which *FSlot* may be composed. If we consider Q to be such an arbitrary environment and P to be the corresponding specification environment (with which the register would be composed), then Q must meet Dom-SF-check and its extracted failures must be contained in those of P (we are considering only those conditions which involve the use of refusal bounds). We show that the refusal bounds presented here place only the lightest of restrictions on the behaviours of Q.

We first observe that $Comm(A_{ar}, Q) = Right$, since $Comm(A_{ar}, FSlot) = Left$. For $t \in Dom_{ar}$, we have that

$$ref_{ar}(t) = \{R \cap A_{ar} \mid (t, R) \in \phi FSlot\} = Sub(\{\{a \in A_{ar} \mid t \circ \langle a \rangle \notin \tau FSlot\}\}).$$

By definition 4.9, $\overline{ref}_{ar}(t) = \{X \subseteq A_{ar} \mid (\forall Y \in ref_{ar}(t))\, X \cup Y \neq A_{ar}\}$. This means that for $X \in \overline{ref}_{ar}(t)$, for every $Y \in ref_{ar}(t)$ there exists $a \in A_{ar} - Y$ such that $a \in A_{ar} - X$. As a result, for $t \in Dom_{ar}$,

$$\overline{ref}_{ar}(t) = Sub(\{A_{ar} - \{a\} \mid t \circ \langle a \rangle \in \tau FSlot\}).$$

This means that the bounds given by $\overline{ref}_{ar}$ will be breached after trace w by any environment with which *FSlot* might be composed only if that environment fails to offer after w at least one event which is valid according to $\tau FSlot = Dom_{ar}$. As a result of this, Q will meet Dom-SF-check with respect to ep_{ar} provided that, whenever it is in the middle of either a call to read or a call to write (and so behaviour over A_{ar} is incomplete), it is always ready to communicate at least one event in which *FSlot* may engage at that point. And, according to the discussion in section 7.3.4, we would expect Q to be always ready to progress in some way a procedure call which it had already begun. Moreover, for $t \in dom_{EP(Q)}$, SF-DEF2 will require P to refuse after $extr_{EP(Q)}(t)$ all events on channels *read* and *write* — i.e. to refuse all communication with the register — only if Q refuses after t all communication (valid with respect to Dom_{ar}) with *FSlot*.

It can be seen by this discussion, therefore, that verification of a particular environment is unlikely to fail simply because the choice of ref_{ar} described here is inappropriate.

7.8 Data independence

In this chapter, we have shown that *FSlot* refines-after-hiding *Register* in all three CSP semantic models. From this, we would like to infer that the 4-slot is a valid implementation of the register in general. However, such an inference is subject to the *caveat* that *FSlot* and the (CSP) register communicate data from a *restricted* set — i.e. *dataint* — while the 4-slot (and the register) may transmit data from much larger sets in practice.[33]

The problem of *data-independence* with regard to refinement in CSP may be stated as follows: if processes Q and P are each parameterised by a data type T, when can we say that Q refines P whatever concrete data type is substituted for T? This problem has been considered in [46] and is also discussed in [63]: provided that values from T are used only in restricted ways in Q and P, a concrete data type containing only a small number of values — for example, one or two values — may be substituted for T in both Q and P. If Q refines P when this concrete data type is used in place of T, then we may conclude that Q refines P whatever data type is substituted for T. However, we are unable to apply the results from [46] here, nor could they be used in respect of any CSP process used to encode the extraction of a set of traces: renaming is used as part of that encoding and the renaming operator is not part of the language allowed by [46].

Nonetheless, it may be seen by inspection that $extr_{ar}$, *FSlot*, *Register* and the processes and renamings used to encode $extr_{ar}$ neither refer explicitly to values from *dataint*[34] nor do they ever take action on the basis of the values held by variables or constants of that type. In view of this and the fact that $\tau ExtFS \subseteq \tau Register$ when *dataint* is a 6-valued set, it is likely that $\tau ExtFS \subseteq \tau Register$ whatever the range of integer values represented by *dataint*. By this and similar reasoning with regard to the stable failures and failures divergences models, we may draw the tentative conclusion that *FSlot* refines-after-hiding *Register* in all three semantic models whatever the range of integer values represented by *dataint*. We therefore conclude that, according to our scheme, the 4-slot is a valid implementation of the register, while also acknowledging the need for further work to treat in a proper and formal manner the issue of data independence.

7.9 Discussion

The work in this chapter had two main purposes. The first was to verify the correctness of the 4-slot mechanism in a novel manner and to derive thereby a result which had not been shown before. The second was to explore how our notion of refinement-after-hiding and our approach to its verification using

[33]It is generally the issue of the size of the data set which is important, rather than the actual values which it contains. See, for example, [46].

[34]0 is a constant when it is used as the initial value stored in each slot in the data array or in the copy of the data array used in TE_{ar}. It could be dispensed with in any case by stipulating that the 4-slot should first complete a call to write before it is allowed to begin a call to read.

FDR2 might fare when used in practice on a real-world example. We consider each of these areas in turn.

7.9.1 What the verification means

In this chapter we have shown that (our CSP representation of) the 4-slot is a valid implementation of a register. Due to the nature of refinement-after-hiding, this means that we may build an implementation of a network which communicates data internally using a register by modifying in a suitable fashion the necessary communication interface and then substituting the 4-slot for the register. This is a significant result for a number of reasons. Firstly, that the 4-slot has more than a single memory slot may be made apparent to a user: a read may begin, interrogate the necessary control variables and then wait for an arbitrary number of writes before completing, thereby appearing to read an old value. It is not immediately clear that such behaviour should be permissible in any valid implementation of a register, which has only a single memory slot. Nonetheless, the success of the verification described here indicates that, once the 4-slot and register have been placed in suitable contexts and all communication hidden, it is effectively impossible for an observer to distinguish between them.

That the 4-slot has been shown to implement a register is also important when it comes to reasoning about systems which might be built using it (the 4-slot). If we may reason initially about a system built using a register, this is likely to be much simpler than considering directly the corresponding system built using the 4-slot. And any results proven about this simpler specification system will be valid for the corresponding system built using the 4-slot. Moreover, knowing that the 4-slot implements a register gives a much better intuition behind the behaviour which it will induce when used in a particular system than is gained by knowing that it meets the conditions of data coherence, data freshness and data sequencing.

7.9.2 Lessons learned and further work

We now consider issues which have been raised during the course of this verification.

The 4-slot was chosen as a case study primarily for the types of reification it exhibits and because it is a real-world example. It was *not* chosen because it was felt in advance that it would be especially amenable to verification (in FDR2) using refinement-after-hiding. As a result, its study has highlighted a number of areas where further work is needed with respect to our means of verification, both in terms of extending the power of the approach and also in terms of developing a methodology for its use.

Describing and deriving extraction mappings

The extraction mapping which is used to interpret traces of *FSlot* as traces of the register is relatively complex, both in its incarnation as an abstract description and in its representation as a CSP process. Although the point has been made above that any process TE_i used in a particular verification should be taken as the authoritative description of the extraction mapping $extr_i$, it would still be useful to have a more mechanical way of proceeding from an extraction mapping description to a CSP process: after all, we have to arrive somehow at an initial version of that CSP process. In order to facilitate a generic translation from mapping to CSP process, it would be useful to have a more structured notation within which extraction mappings could be expressed. In general, an extraction mapping is defined compositionally over an event a and its history t: t is then effectively mapped into a particular information domain, its representation in that domain being used to determine the event to which a must be extracted. Although different information domains would be used for different verifications, a standard translation from information domain and mapping notation to CSP process would be very useful. (Such issues are also of relevance in terms of describing the mapping *ref* and representing it as necessary as a CSP process.)

Deriving a mapping for successful verification

The extraction mapping used here was developed as the verification proceeded and was not known in advance. In such circumstances, verification may fail either because the mapping used is unsuitable or because the process being verified cannot be related to the specification under consideration. An interesting area to explore, therefore, is how it might be possible to tell that a particular verification will never succeed or, conversely, that a suitable mapping does, in fact, exist. (This is related to the issue of completeness: see brief discussion in chapter 8.) In the first instance, such work would be concerned with the notion of refinement-after-hiding itself, rather than with the actual means of verification.

In addition, the process of developing the mapping used here was not straightforward and this raises a number of important issues. In particular, it casts doubt on the ease with which refinement-after-hiding might be used in practice and suggests the need for further work to explore how the process of developing mappings might be made easier. For example, one could explore the development and use of semi-automated tools for this purpose. However, we should perhaps reserve judgement on the ease of use of refinement-after-hiding until it has been applied to a more extensive range of case studies.

Considering the environment

In section 7.4, it could be seen that the environment with which the 4-slot is to be composed plays a significant role in determining the form of the traces mapping to be used for verification. This role of the environment needs to be explored further, especially with respect to determining refusal bounds, and its consideration needs to be built firmly into any methodology for use of refinement-after-hiding in practice.

Working with procedural interfaces

Although the 4-slot is defined initially using a procedural interface, we have assumed in the verification presented in this chapter that all of its events are visible to the environment — i.e. both events effecting data transfer and those concerned with manipulating control variables — and that the environment engages in both types of event whenever it wishes to transfer data. This is because we always extract on the occurrence of events which either write to or read from a control variable, meaning that these events must be visible in our CSP representation of the 4-slot. This also means that such events should be visible in (our representation of) the environment. (This issue was discussed in section 7.3.4)

This is an instance of a general problem faced by refinement-after-hiding if it is to be used in the verification of processes which communicate using procedural interfaces. In particular, any process Q which calls a procedure in another process will engage in an event to represent the procedure call and one to represent the procedure return, but will not engage in any events from the body of the procedure. If, during the verification of Q, we need to extract on the occurrence of an event which is part of the body of the procedure then this will not be possible without modification of Q. Further work is needed both to show how this modification may be carried out automatically and also to show formally that such modification does not affect the validity of any verification which may be carried out.

Complex statement of domain of mapping

In the development of our notion of refinement-after-hiding, including the development of its predecessors, it was implicitly assumed that the method would be applied when the extraction mapping to be used was known in advance, meaning that the *domain* of the mapping would also be known in advance. Moreover, it was assumed that the latter would be expressed in a relatively straightforward, syntactically simple form. These assumptions did not hold in the verification in this chapter, however, and *FSlot* itself was the only syntactic representation available of the domain of the mapping which we used.

Due to the syntactic complexity of *FSlot*, it was difficult to modify directly the syntactic definition of Dom_{ar} given by it. As a result, the process TE_{ar} is constructed independently of *FSlot* and so defines a mapping with a domain strictly larger than $\tau FSlot$. And had we defined processes DSF_{ar} and RE_{ar} to deal with verification in the stable failures model, it would not have been possible to build them directly around *FSlot*, even though such a manner of construction is implicit in the definitions given in chapter 6.

Further work is needed in the first instance to explore the sort of implementation processes which might give rise to this problem of complex definitions of mapping domains. Specifically, we intend to consider fully the issue of constructing DSF_i and RE_i in such cases: this is complicated by the fact that certain failures are obscured in these processes, depending on whether or not their trace component is in dom_i. In addition, the formal framework for verification using FDR2 may need to be extended to deal with this issue.

Deriving refusal bounds and verifying Dom-SF-check

Again due to the fact that the extraction pattern used here did not exist prior to this verification, its ref_{ar} component was defined directly in terms of the failures of *FSlot*. A similar approach is likely to be necessary whenever we are not provided with refusal bounds in advance of a particular verification and it should guarantee that the implementation component under consideration meets condition Dom-SF-check. However, Dom-SF-check will also need to be verified of any environment with which that component may be composed and this could cause problems. If the refusal bounds to be used come directly from an implementation component, then it will be impossible in the general case to derive a statement of them without first calculating the semantics of that component. This means that it would be difficult to define directly any necessary process DSF_i as described in section 6.5. Further work is therefore needed to consider the automatic verification of Dom-SF-check in such a situation.

In addition, it is not clear in general how refusal bounds when behaviour is *complete* might be derived from the failures of the implementation component under consideration. This is due to the fact that, if verification is to be successful, the failures of the *specification* will also play a role in determining the nature of those bounds.

Extracting to non-singleton traces

An obvious area for further work is to explore how we might represent extraction mappings as CSP processes in the general case that extraction to non-singleton traces is allowed. This is especially important if we are to be able to use FDR2 when verifying *equivalence*: this necessitates interpreting abstract behaviours in a more concrete form and so is generally going to require extraction to non-singleton traces. It is for this reason that we have not attempted to verify the "equivalence" of the 4-slot and the register.

"Straightforward" case studies

A case study or studies will be explored in future where the problems and issues identified above might not be expected to arise, for example where an implementation process communicates data using certain fault tolerant mechanisms or a particular communication protocol. In such cases, the extraction mapping to be used should be determined in advance (by the nature of the fault tolerant mechanism or communication protocol) and the domain of that mapping should (hopefully) be stated explicitly. This will give a better idea of how the method of automatic verification presented in chapter 6 might perform when it is not being pushed to its limits. In addition, it should give us a better opportunity to explore the way in which automatic verification might work in the stable failures and failures divergences models, which thing is missing from the work in this chapter.

Chapter 8

Conclusion

In the process algebraic framework, the meaning of processes is based firmly on the notion of an observer and what he/she may observe of the behaviour of a particular process. As a result, we abstract from internal actions since they cannot be observed: two processes are regarded as equivalent (within a suitable semantic framework) if they have the same external behaviours, regardless of the manner in which they perform computations internally. It is arguable, however, that this notion of observability is too stringent and should be relaxed, on the basis that processes are rarely used in isolation. In other words, if we assume that an observer only observes complete systems or networks — rather than individual component processes — the set of visible events is immediately much reduced and the way is open for defining a more relaxed notion of equivalence or refinement. This suggests the development of a notion of correctness-in-context, where visible events are partitioned into those that an observer will be able to see in the final network built and those which will be invisible to him/her.

Using these notions of correctness-in-context and the partitioning of visible events, along with the device of an interpretive mapping, chapter 3 gives an abstract formal statement of what it means for a particular implementation relation to constitute a notion of *refinement-after-hiding*: this is captured in the conditions RAH1-3. From these and a number of other basic conditions, we are able to derive a set of conditions which are sufficient to define refinement-after-hiding in practice. These are put to use in chapter 4 as we modify and extend an existing such notion. Not only is the work from chapter 3 of fundamental importance in carrying out this modification, it also gives a clear and definite framework within which we are able to understand the form of our concrete notion of refinement-after-hiding and why exactly it works. The implementation relation defined in chapter 4 gives a generalisation of standard CSP refinement in all three semantic models and provides the ability to deal with a variety of types of reification in the move from specification to implementation: more specifically, it can deal with data reification, external behaviour decomposition and external relaxation of atomicity, as evidenced by the successful verification in chapter 7.

Chapter 6 defines a means of automatic verification for our concrete notion of refinement-after-hiding, using the existing industrial-strength tool FDR2. This is significant for a number of reasons: it allowed us to proceed more quickly to verification in practice of a real-world example; FDR2's state-space compression techniques are vital to our ability to perform verification, as even the representation of the 4-slot from chapter 7 initially has a large number of states; its (FDR2's) debugging facilities proved crucial in the development of the extraction mapping used in the same chapter. Finally, chapter 7 shows that Simpson's 4-slot asynchronous communication mechanism is a valid implementation of a register, using the notion of refinement-after-hiding from chapter 4 and the means of verification from chapter 6. This is an important result both in this specific case and in terms of what it shows may be possible in general: in the move from register to (CSP representation of the) 4-slot, data reification, external behaviour decomposition and external relaxation of atomicity have all occurred, yet verification may still be effected successfully.

As a final comment, we note that the work presented in chapters 3, 4 and 6 may be regarded as a whole which is greater than the sum of its parts. Using our means of automatic verification, we were able to proceed quickly to verification of a real-world example. The consideration of this example then fed back into the development of the implementation relation in chapter 4. For example, the requirement that any particular extraction pattern should deal only with input or only with output events proved to be too restrictive in practice, while the theory from chapter 3 indicated that such an approach was not necessary in order for refinement-after-hiding to work. It is also envisaged that these three components will play a similar, mutually supportive role in future work. In particular, we will consider alternative means of mapping refusals, such as that described in section 4.8 and suggested by the theory in chapter 3. Encoding any such alternative approaches as CSP processes in FDR2 (where that is possible) will allow for their rapid use in the verification of real-world examples, the success or failure of which verifications will reflect on their usefulness in practice. And if practice tells us that a particular means of mapping refusals should be modified, then the theory gives a framework within which those modifications may be assessed and carried out.

8.1 Further work

At various points throughout this thesis, we have indicated areas in which further work is needed or in which such work might yield interesting and useful results. We identify here five main areas on which we shall concentrate.

8.1.1 Refinement-after-hiding and "completeness"

Assume that $F_{impl}(Q_1, \ldots, Q_n)$ is an implementation network and the corresponding specification network is given by $F_{spec}(P_1, \ldots, P_n)$. Future work will address the question of whether it is always possible to come up with suitable extraction patterns such that Q_i refines-after-hiding P_i for $1 \leq i \leq n$ in the event that $F_{impl}(Q_1, \ldots, Q_n)$ refines $F_{spec}(P_1, \ldots, P_n)$ according to standard CSP refinement. In the event that refinement-after-hiding is not complete in this sense, we will aim to establish restrictions on implementation and specification networks such that the property of completeness is enjoyed in the restricted domain.

8.1.2 Barbed congruence and refinement-after-hiding

Chapter 5 (page 119) describes in some detail areas which might be explored concerning the relation between barbed congruence and refinement-after-hiding. In particular, the ability to verify correctness when at least relaxation of atomicity has occurred *without* the need to construct an interpretive mapping would be extremely desirable.

8.1.3 Mapping refusals

The approach to mapping refusals used in this thesis is based on the notion of refusal bounds and the treatment of communication as asymmetric in character. A possible alternative approach to mapping refusals is proposed in section 4.8. It is currently not clear what advantages one might possess over the other nor how one might decide the appropriateness of a particular approach in a particular set of circumstances. Further work, both in terms of theory and of practical examples, will look at the types of process where one approach might allow verification to succeed while the other might cause it to fail: this will help us to identify situations where it would be advisable to choose one approach over the other.

Moreover, if an alternative means of mapping refusals could be developed, its use might avoid some of the problems which may arise when the necessary extraction patterns are not known in advance of a particular verification (see discussion in section 7.9).

8.1.4 Improving the means of automatic verification

Section 7.9 contains a number of ways in which the means of automatic verification given in this thesis could be extended and improved. As a first step, a notation to represent extraction mappings will be explored, along with ways of mechanically translating from such a notation to a CSP process which represents the relevant mapping.

8.1.5 Further case studies

Finally, it is necessary to consider the verification of further example processes and systems, both in order to fully evaluate the usefulness of our notion of refinement-after-hiding and also to develop a proper methodology regarding its use.

Appendix A

Proofs from chapter 3

A.1 Proofs from section 3.2

Proof of proposition 3.3

Proof. 1.
$$\begin{aligned}\alpha(P \setminus A) &= [[\beta(P \setminus A)]] && \text{(by definition 3.10)}\\ &= [[\beta(P) - A]] && \text{(by figure 2.5)}\\ &= [[\beta(P)]] - A && \text{(by def. 3.7 and since } A \in \mathit{AllSet})\\ &= (\alpha P) - A && \text{(by definition 3.10)}\end{aligned}$$

2.
$$\begin{aligned}\alpha(P \parallel_Y Q) &= [[\beta(P \parallel_Y Q)]] && \text{(by definition 3.10)}\\ &= [[\beta(P) \cup \beta(Q)]] && \text{(by figure 2.5)}\\ &= [[\beta(P)]] \cup [[\beta(Q)]] && \text{(by definition 3.7)}\\ &= \alpha P \cup \alpha Q && \text{(by definition 3.10)}\end{aligned}$$

□

Proof of proposition 3.4

Proof. Since $A \cap B \neq \varnothing$, it follows that $B \neq \varnothing$. Hence, $B = \bigcup_{i \in I} A_i$ where I is a non-empty indexing set into *MinSet* and so $A \cap A_i \neq \varnothing$ for some $i \in I$. By definition 3.5, $A = A_i$ and so $A \subseteq B$. □

Proof of proposition 3.5

Proof. The proof is immediate in the event that $A = \varnothing$ and so we assume that $A \neq \varnothing$. We use *RHS* to denote $\bigcup\{\mathit{events}(t) \mid t \in \mathit{BTrace} \wedge \mathit{events}(t) \subseteq A\}$ in this proof. It is immediate that $\mathit{RHS} \subseteq A$. We therefore show $A \subseteq \mathit{RHS}$, by assuming there exists $a \in A - \mathit{RHS}$. By definitions 3.6, 3.5 and 3.4(1), there exists $t \in \mathit{BTrace}$ such that $a \in \mathit{events}(t)$; moreover, $\mathit{events}(t) \not\subseteq A$.

Since $A = \bigcup_{i\in I} A_i$ where I is a non-empty indexing set into *MinSet*, there exists $i \in I$ such that $a \in A_i$ but $events(t) \not\subseteq A_i$ since $events(t) \not\subseteq A$. Hence, we have a contradiction by definition 3.5(1). □

Proposition A.1. $\lambda(\varnothing) = \varnothing$.

Proof. We consider each of two cases in turn.

Case 1: $\langle\rangle \in BTrace$ and so $\lambda(\langle\rangle)$ is defined. By definition 3.8, it suffices to show that $\lambda(\langle\rangle) = \langle\rangle$. Let $STOP$ be an implementation process (recall that we may assume $STOP$ is an implementation process by definition 3.9 and since $\beta(STOP) = \varnothing \subseteq \Sigma_{impl}$). Since $\beta(STOP) = \varnothing$, then $\alpha STOP = \varnothing$ and so $\alpha STOP \subseteq Fvis$. Thus, by RAH1, $\lambda([\![STOP]\!]_X) = [\![STOP]\!]_X$ for $X \in \{T, SF, FD\}$. Hence, $\lambda(\tau STOP) = \tau STOP$ and so, by definition 3.3, $\lambda(\langle\rangle) = \langle\rangle$.

Case 2: $\langle\rangle \notin BTrace$. In this case, the proof is immediate by definition 3.8. □

Proof of proposition 3.6

Proof. In the event that $I = \varnothing$, the proof is immediate by proposition A.1 and so we consider the case that $I \neq \varnothing$. By definition 3.8, $\lambda(\bigcup_{i\in I} A_i)$ is given by:

$$\begin{aligned}
&\textstyle\bigcup\{events(\lambda(t)) \mid t \in BTrace \ \wedge\ events(t) \subseteq \bigcup_{i\in I} A_i\} \\
=&\textstyle\bigcup\{events(\lambda(t)) \mid t \in BTrace \ \wedge\ (\exists i \in I)\ events(t) \subseteq A_i\} \quad \text{(by def. 3.5(1))} \\
=&\textstyle\bigcup_{i\in I}(\bigcup\{events(\lambda(t)) \mid t \in BTrace \ \wedge\ events(t) \subseteq A_i\}) \\
=&\textstyle\bigcup_{i\in I} \lambda(A_i) \quad \text{(by definition 3.8)}
\end{aligned}$$

□

Proof of proposition 3.7

Proof. Let $A = \bigcup_{i\in I} A_i$ and $A' = \bigcup_{j\in J} A_j$, where I, J are indexing sets into *MinSet* and $I \cap J = \varnothing$ since $A \cap A' = \varnothing$ (note that $\varnothing \notin MinSet$). We consider each of two cases in turn.

Case 1: Either $A = \varnothing \neq A'$, $A \neq \varnothing = A'$ or $A = \varnothing = A'$. The proof in this case follows by proposition A.1.

Case 2: $A \neq \varnothing \neq A'$. By proposition 3.6, $\lambda(A) = \bigcup_{i\in I} \lambda(A_i)$ and $\lambda(A') = \bigcup_{j\in J} \lambda(A_j)$. Since $I \cap J = \varnothing$, $A_i \neq A_j$ for all $i \in I$, $j \in J$. Thus, by definitions 3.5(2) and 3.8, $\lambda(A_i) \cap \lambda(A_j) = \varnothing$ for all $i \in I$,$j \in J$ and so the proof in this case follows. □

Proof of proposition 3.8

Proof. Let I, J be indexing sets into *MinSet* such that $A = \bigcup_{i \in I} A_i$ and $B = \bigcup_{j \in J} A_j$.

1. It is immediate by definition 3.5 that $A \oplus B = \bigcup_{k \in I \oplus J} A_k$ and so $A \oplus B \in \mathit{AllSet}$ by definition 3.6.

2. We observe that:

$$\begin{aligned}
\lambda(A \oplus B) &= \lambda(\textstyle\bigcup_{i \in I} A_i \oplus \bigcup_{j \in J} A_j) \\
&= \lambda(\textstyle\bigcup_{k \in I \oplus J} A_k) \\
&= \textstyle\bigcup_{k \in I \oplus J} \lambda(A_k) && \text{(by proposition 3.6)} \\
&= \textstyle\bigcup_{i \in I} \lambda(A_i) \oplus \bigcup_{j \in J} \lambda(A_j) && \text{(by definitions 3.5 and 3.8)} \\
&= \lambda(A) \oplus \lambda(B) && \text{(by proposition 3.6)}
\end{aligned}$$

□

Proof of proposition 3.9

Proof. 1. The proof is immediate from definitions 3.5 and 3.6.

2. We observe that:

$$\begin{aligned}
& \textstyle\bigcup_{A \in \mathit{MinSet}} \lambda(A) \\
=\ & \lambda(\textstyle\bigcup_{A \in \mathit{MinSet}} A) && \text{(by proposition 3.6)} \\
=\ & \lambda(\Sigma_{\mathit{impl}}) && \text{(by definition 3.5)} \\
=\ & \textstyle\bigcup \{ \mathit{events}(\lambda(t)) \mid t \in \mathit{BTrace} \ \wedge \ \mathit{events}(t) \subseteq \Sigma_{\mathit{impl}} \} && \text{(by def. 3.8)} \\
=\ & \textstyle\bigcup \{ \mathit{events}(\lambda(t)) \mid t \in \mathit{BTrace} \} && \text{(by definition 3.4(1))} \\
=\ & \Sigma_{\mathit{spec}} && \text{(by definition 3.4(2))}
\end{aligned}$$

3. Let $A \in \mathit{AllSet}$. By definition 3.5, $\Sigma_{\mathit{impl}} = \bigcup_{A' \in \mathit{MinSet}} A'$ and so $A \subseteq \Sigma_{\mathit{impl}}$ by definition 3.6. By the proof of part 2 of the proposition, $\Sigma_{\mathit{spec}} = \lambda(\Sigma_{\mathit{impl}})$ and so $\lambda(A) \subseteq \Sigma_{\mathit{spec}}$ by definition 3.8 and since $A \subseteq \Sigma_{\mathit{impl}}$. □

Proof of proposition 3.10

Proof. 1. We assume that $X \subseteq A$. By definition 3.6, $[[X]] = \bigcup_{i \in I} A_i$, where I is an indexing set into *MinSet*. In the event that $[[X]] = \varnothing$, the proof is immediate and so we consider the case that $[[X]] \neq \varnothing$. Let $i \in I$. By definition 3.7, $X \cap A_i \neq \varnothing$ and so $A \cap A_i \neq \varnothing$. Thus, $A_i \subseteq A$ by proposition 3.4.

2. We show that $[[R]] \cup [[S]] \subseteq [[R \cup S]]$; the proof that $[[R \cup S]] \subseteq [[R]] \cup [[S]]$ is similar. By definition 3.6, $[[R]] = \bigcup_{i \in I} A_i$, where I is an

indexing set into *MinSet*. We show that $[[R]] \subseteq [[R \cup S]]$: that $[[S]] \subseteq [[R \cup S]]$ may be shown in a similar manner. In the event that $[[R]] = \varnothing$, the proof is immediate and so we consider the case that $[[R]] \neq \varnothing$. Let $i \in I$. By definition 3.7, $A_i \cap R \neq \varnothing$ and so $A_i \cap [[R \cup S]] \neq \varnothing$. Hence, by proposition 3.4, $A_i \subseteq [[R \cup S]]$. □

Proof of proposition 3.11

Proof. By definition of *Imp* in definition 2.3, there exists $(P \parallel_Y Q) \in Imp(F_{impl}(Q_1, \ldots, Q_n))$ such that, by R2, $Y = \alpha P \cap \alpha Q$. By definition 3.10, $\alpha P, \alpha Q \in AllSet$ and so $Y \in AllSet$ by proposition 3.8(1). □

Proposition A.2. *Let $A \in AllSet$.*

1. $\Sigma_{impl} - A \in AllSet$.

2. $\lambda(\Sigma_{impl} - A) = \Sigma_{spec} - \lambda(A)$.

Proof. 1. The proof follows by proposition 3.9(1) and proposition 3.8(1).
2. We have:

$$\begin{aligned}
\lambda(\Sigma_{impl} - A) &= \lambda(\Sigma_{impl}) - \lambda(A) && \text{(by prop.s 3.9(1) and 3.8(2))} \\
&= \lambda(\textstyle\bigcup_{A' \in MinSet} A') - \lambda(A) && \text{(by definition 3.5)} \\
&= \textstyle\bigcup_{A' \in MinSet} \lambda(A') - \lambda(A) && \text{(by proposition 3.6)} \\
&= \Sigma_{spec} - \lambda(A) && \text{(by prop. 3.9(2))}
\end{aligned}$$

□

Proof of proposition 3.12

Proof. Let $Q \triangleq FST\langle BTrace \rangle$ be a (component) implementation process. By proposition 2.12(1), $\tau Q = Pref(BTrace)$ and so, by PREF-CLOS and definition 3.3, $\lambda([\![Q]\!]_T)$ is defined. $\beta(Q) = \bigcup_{t \in BTrace} events(t)$ by proposition 2.12(2) and so $\beta(Q) = \Sigma_{impl}$ by definition 3.4(1). By definition 3.10 and proposition 3.9(1), $\alpha Q = \Sigma_{impl}$. Thus, by HIDE-INVIS, there exists $A \in AllSet$ such that $\Sigma_{impl} - A = \Sigma_{impl} \cap Fvis = Fvis$. Hence, $Fvis \in AllSet$ by proposition A.2(1). □

A.2 Proofs from section 3.3

Proof of theorem 3.13

Proof. Let $Q \triangleq FST\langle t \rangle$ be a (component) implementation process. By proposition 2.12, $\tau Q = Pref(t)$ and $\beta(Q) = events(t)$. Hence, $\alpha Q =$

$[[events(t)]]$. We show that $\alpha Q \subseteq Fvis$ by considering each of two cases in turn.

Case 1: $t = \langle\rangle$. The proof is immediate in this case, since $\alpha Q = \varnothing$.

Case 2: $t \neq \langle\rangle$. In this case, $\alpha Q = [[events(t)]] = \bigcup_{i \in I} A_i$ where I is a non-empty indexing set into $MinSet$. Moreover, $events(t) \cap A_i \neq \varnothing$ and so $A_i \cap Fvis \neq \varnothing$ for every $i \in I$. It follows by propositions 3.12 and 3.4 that $A_i \subseteq Fvis$ for every $i \in I$ and so $\alpha Q \subseteq Fvis$.

Hence, by TI1, $\lambda(\tau Q)$ is defined and so, by definition 3.3, $\lambda(t)$ is defined. Also by TI1, $\lambda(\tau Q) = \tau Q$ and so $max_{\leq}(\lambda(\tau Q)) = max_{\leq}(\tau Q)$. Hence, by definition 3.3 and TR-MONO, $\lambda(t) = t$. □

Proof of theorem 3.15

Proof. We assume that $\lambda(t)$ is defined. Let $Q \triangleq FST\langle t\rangle$ be a (component) implementation process. By proposition 2.12(1), $\tau Q = Pref(t)$. By PREF-CLOS and definition 3.3, $\lambda(\tau Q)$ is defined. By TI2, $\lambda(\tau(Q \setminus A))$ is defined and so $\lambda(t \setminus A)$ is defined by definition 3.3. Also by TI2, $\lambda(\tau(Q \setminus A)) = \lambda(\tau Q) \setminus B$. Hence, by definition 3.3, TR-MONO and the monotonicity of the hiding operator over traces (proposition 2.6):

$$\lambda(t \setminus A) = max_{\leq}(\lambda(\tau(Q \setminus A))) = max_{\leq}(\lambda(\tau Q) \setminus B) = \lambda(t) \setminus B.$$

□

Proof of theorem 3.16

Proof. Let B be such that $\lambda(\setminus A) = \setminus B$.

($\subseteq$) We first show that $B \subseteq \lambda(A)$ by assuming there exists $a \in B - \lambda(A)$. By definition 3.11(1), $B \subseteq \Sigma_{spec}$ and so, by definition 3.4(2), there exists $s \in BTrace$ such that $a \in events(\lambda(s))$. By definition 3.8, $\lambda(A) = \bigcup\{events(\lambda(t)) \mid t \in BTrace \wedge events(t) \subseteq A)\}$ and so $events(s) \not\subseteq A$. Hence, by definitions 3.5(1) and 3.6, $events(s) \cap A = \varnothing$. Since $s \in BTrace$, then $\lambda(s)$ is defined. It follows by RAH2-T that $\lambda(s \setminus A) = \lambda(s) \setminus B$ and so, since $events(s) \cap A = \varnothing$, $\lambda(s) = \lambda(s) \setminus B$. Hence, $events(\lambda(s)) \cap B = \varnothing$. This, however, gives a contradiction, since $a \in events(\lambda(s)) \cap B$.

($\supseteq$) We now show that $\lambda(A) \subseteq B$. In the event that there does not exist $t \in BTrace$ such that $events(t) \subseteq A$, the proof is immediate by definition 3.8. We therefore assume that there exists $t \in BTrace$ such that $events(t) \subseteq A$. By definition 3.8, it suffices to show that $events(\lambda(t)) \subseteq B$. Since $t \in BTrace$, then $\lambda(t)$ is defined. Hence, by RAH2-T, $\lambda(\langle\rangle) = \lambda(t \setminus A) = \lambda(t) \setminus B$. By theorem 3.14, $\lambda(t) \setminus B = \langle\rangle$ and so $events(\lambda(t)) \subseteq B$. □

Proof of theorem 3.17

Proof. We assume the following:

- $T_Y = \{v \in \mathit{BTrace} \mid \mathit{events}(v) \subseteq Y\} \cup \{\langle\rangle\}$.
- $Q = \mathit{FST}\langle T_Y\rangle \parallel_Y \mathit{STOP}$ is a (component) implementation process.
- $P_w = \mathit{FST}\langle w\rangle \parallel_\varnothing Q$ is a (component) implementation process for $w \in \mathit{BTrace}$.

Before proceeding with the proof proper we prove (A.1) and (A.2), which relate to Q and P_w respectively.

$$\tau Q = \{\langle\rangle\},\ \lambda(\tau Q) \text{ is defined and } \alpha Q = Y \qquad \text{(A.1)}$$

By proposition 2.12(1), $\tau \mathit{FST}\langle T_Y\rangle = \mathit{Pref}(T_Y)$. Since $\mathit{events}(t) \subseteq Y$ for every $t \in \tau \mathit{FST}\langle T_Y\rangle$ and $\tau \mathit{STOP} = \{\langle\rangle\}$, $\tau Q = \{\langle\rangle\}$. Thus, by definition 3.3 and theorem 3.14, $\lambda(\tau Q)$ is defined. By proposition 2.12(2), $\beta(\mathit{FST}\langle T_Y\rangle) = \bigcup_{t\in T_Y} \mathit{events}(t)$. Hence, by proposition 3.5, $\beta(\mathit{FST}\langle T_Y\rangle) = Y$. Since $\beta(\mathit{STOP}) = \varnothing$, $\beta(Q) = Y$ and, since $Y \in \mathit{AllSet}$, $\alpha Q = [[Y]] = Y$. Hence, we have shown (A.1).

$$\tau P_w = \mathit{Pref}(w),\ \lambda(\tau P_w) \text{ is defined and } \alpha P_w = [[\mathit{events}(w)]] \cup Y \qquad \text{(A.2)}$$

By proposition 2.12(1), $\tau \mathit{FST}\langle w\rangle = \mathit{Pref}(w)$ and, by (A.1), $\tau P_w = \mathit{Pref}(w)$. Since $w \in \mathit{BTrace}$, $\lambda(\tau P_w)$ is defined by definition 3.3 and PREF-CLOS. By proposition 2.12(2), $\beta(\mathit{FST}\langle w\rangle) = \mathit{events}(w)$. By the proof of (A.1), $\beta(Q) = Y$ and so $\beta(P_w) = \mathit{events}(w) \cup Y$. Hence, since $Y \in \mathit{AllSet}$, $\alpha P_w = [[\mathit{events}(w)]] \cup Y$. Thus, we have shown (A.2).

We now proceed with the proof proper, where Z is such that $\lambda(\parallel_Y) = \parallel_Z$.

($\subseteq$) We show that $Z \subseteq \lambda(Y)$ by assuming there exists $a \in Z - \lambda(Y)$. By definition 3.11(2), we know that $Z \subseteq \Sigma_{spec}$ and so, by definition 3.4(2), there exists $s \in \mathit{BTrace}$ such that $a \in \mathit{events}(\lambda(s))$. By definition 3.8, $\lambda(Y) = \bigcup\{\mathit{events}(\lambda(t)) \mid t \in \mathit{BTrace} \wedge \mathit{events}(t) \subseteq Y)\}$ and so $\mathit{events}(s) \not\subseteq Y$. Hence, by definitions 3.5(1) and 3.6 we have that $\mathit{events}(s) \cap Y = \varnothing$. Let P_s be a (component) implementation process. By (A.2), $\tau P_s = \mathit{Pref}(s)$ and $\lambda(\tau P_s)$ is defined. By (A.1), $\tau Q = \{\langle\rangle\}$ and $\lambda(\tau Q)$ is defined. By (A.1) and (A.2), $\alpha P_s \cap \alpha Q = Y$. Hence, $\lambda(\tau(P_s \parallel_Y Q)) = \lambda(\tau P_s) \parallel_Z \lambda(\tau Q)$ by TI3. Since $\mathit{events}(s) \cap Y = \varnothing$ and by theorem 3.14 and definition 3.3, $\lambda(\mathit{Pref}(s)) = \lambda(\mathit{Pref}(s)) \parallel_Z \{\langle\rangle\}$. Hence, by definition 3.3, there exists $v \leq s$ such that $\lambda(s) \in \lambda(v) \parallel_Z \langle\rangle$. It follows by TRP (i.e. $\lambda(s){\lceil}Z = \langle\rangle{\lceil}Z$) that $\mathit{events}(\lambda(s)) \cap Z = \varnothing$ and so we have a contradiction since $a \in \mathit{events}(\lambda(s)) \cap Z$.

($\supseteq$) We now show $\lambda(Y) \subseteq Z$. In the event that there does not exist $t \in BTrace$ such that $events(t) \subseteq Y$, the proof is immediate by definition 3.8. Thus, we assume there exists $t \in BTrace$ such that $events(t) \subseteq Y$. By definition 3.8, it suffices to show that $events(\lambda(t)) \subseteq Z$. Since $events(t) \subseteq Y$ and $Y \in AllSet$, $[[events(t)]] \subseteq Y$ by definition 3.7. Let P_t be a (component) implementation process. By (A.2), $\tau P_t = Pref(t)$, $\lambda(\tau P_t)$ is defined and $\alpha P_t = [[events(t)]] \cup Y = Y$. Hence, $\lambda(\tau(P_t \parallel_Y P_t)) = \lambda(\tau P_t) \parallel_Z \lambda(\tau P_t)$ by TI3. Thus, since $events(t) \subseteq Y$, $\lambda(Pref(t)) = \lambda(Pref(t)) \parallel_Z \lambda(Pref(t))$. Then, by definition 3.3 and TR-MONO,

$$\{\lambda(t)\} = max_{\leq}(\lambda(Pref(t))) = max_{\leq}(\lambda(Pref(t)) \parallel_Z \lambda(Pref(t))).$$

It holds trivially that $\lambda(t)\lceil Z = \lambda(t)\lceil Z$ and so $\lambda(t) \parallel_Z \lambda(t) \neq \varnothing$. Hence, by proposition 2.7, $\{\lambda(t)\} = \lambda(t) \parallel_Z \lambda(t)$ and so $events(\lambda(t)) \subseteq Z$. □

A.3 Proofs from section 3.4

Proposition A.3. *Let $A \in AllSet$, $B = \lambda(A)$ and let t be a trace such that $\lambda(t)$ is defined. Then $\lambda(t\lceil A)$ is defined and $\lambda(t\lceil A) = \lambda(t)\lceil B$.*

Proof. Let $\overline{A} = \Sigma_{impl} - A$ and $\overline{B} = \Sigma_{spec} - B$. By proposition A.2, $\overline{A} \in AllSet$ and $\overline{B} = \lambda(\overline{A})$. By RAH2-T, $\lambda(t \setminus \overline{A})$ is defined and $\lambda(t \setminus \overline{A}) = \lambda(t) \setminus \overline{B}$. By proposition 3.9(3), $A \subseteq \Sigma_{impl}$ and $B \subseteq \Sigma_{spec}$. Hence, $\Sigma_{impl} = A \cup \overline{A}$ and $\Sigma_{spec} = B \cup \overline{B}$. Since $events(t) \subseteq \Sigma_{impl}$ by definition 3.4(1), then $t \setminus \overline{A} = t\lceil A$; hence, $\lambda(t\lceil A)$ is defined. Since $events(\lambda(t)) \subseteq \Sigma_{spec}$ by definition 3.4(2), $\lambda(t) \setminus \overline{B} = \lambda(t)\lceil B$. Hence, $\lambda(t\lceil A) = \lambda(t)\lceil B$. □

Proof of theorem 3.19

Proof. 1. ($\Longrightarrow$) We assume that $\lambda(t)$ and $\lambda((t \circ \langle a \rangle)\lceil A)$ are defined. Let $\overline{A} = \Sigma_{impl} - A$ and so $\overline{A} \in AllSet$ by proposition A.2(1). By proposition 3.9(3), $A \subseteq \Sigma_{impl}$ and so $\Sigma_{impl} = A \cup \overline{A}$. We define the (component) implementation processes P and Q as follows: $P = FST\langle t\lceil \overline{A}\rangle$ and $Q = FST\langle (t \circ \langle a \rangle)\lceil A\rangle$. By proposition 2.12, $\tau P = Pref(t\lceil \overline{A})$ and $\beta(P) = events(t\lceil \overline{A})$. Hence, $\alpha P \subseteq \overline{A}$ since $\overline{A} \in AllSet$. Moreover, $\lambda(\tau P)$ is defined by proposition A.3, PREF-CLOS and definition 3.3. Similarly, $\tau Q = Pref((t \circ \langle a \rangle)\lceil A)$, $\alpha Q \subseteq A$ and $\lambda(\tau Q)$ is defined. As a result, $\alpha P \cap \alpha Q = \varnothing$. Hence, by TI3, $\lambda(\tau(P \parallel_\varnothing Q))$ is defined. By definition 3.4(1) and since $\lambda(t)$ and $\lambda((t \circ \langle a \rangle)\lceil A)$ are defined,

$$events(t \circ \langle a \rangle) = events(t) \cup events((t \circ \langle a \rangle)\lceil A) \subseteq \Sigma_{impl}$$

and so, since $\Sigma_{impl} = A \cup \overline{A}$,

$$t \circ \langle a \rangle \in (Pref(t\lceil \overline{A}) \parallel_\varnothing Pref((t \circ \langle a \rangle)\lceil A)) = \tau(P \parallel_\varnothing Q).$$

Thus, $\lambda(t \circ \langle a \rangle)$ is defined by definition 3.3.

($\Longleftarrow$) We assume that $\lambda(t \circ \langle a \rangle)$ is defined. Hence, $\lambda(t)$ is defined by PREF-CLOS and $\lambda((t \circ \langle a \rangle)\lceil A)$ is defined by proposition A.3.

2. Let $B = \lambda(A)$. We assume that $\lambda(t \circ \langle a \rangle)$ is defined. By PREF-CLOS, $\lambda(t)$ is defined and, by TR-MONO,

$$\lambda(t \circ \langle a \rangle) = \lambda(t) \circ x$$

for some trace x. Also by TR-MONO,

$$\lambda(t \circ \langle a \rangle \lceil A) = \lambda(t \lceil A) \circ r$$

for some trace r. By proposition A.3, $\lambda(t \circ \langle a \rangle \lceil A) = \lambda(t \circ \langle a \rangle)\lceil B$. Thus,

$$\lambda(t\lceil A) \circ r \;=\; (\lambda(t) \circ x)\lceil B \;=\; \lambda(t)\lceil B \circ x\lceil B.$$

Hence, by proposition A.3, $r = x\lceil B$. By RAH2-T,

$$\lambda(t) \setminus B = \lambda(t \setminus A) = \lambda(t \circ \langle a \rangle \setminus A) = \lambda(t \circ \langle a \rangle) \setminus B = (\lambda(t) \circ x) \setminus B.$$

Thus, $events(x) \subseteq B$ and so $x\lceil B = x = r$. □

Results used in the proof of theorem 3.20

In all of the proofs in the remainder of this section, we assume that conditions S1-7 and TS1-4 from figure 3.4 all hold. (Recall also that some necessary supporting results which have already appeared are restated and reproved here using only S1-7 and TS1-4.)

Proposition A.4. *Let $A \in MinSet$ and $B \in AllSet$ be such that $A \cap B \neq \varnothing$. Then $A \subseteq B$.*

Proof. Since $A \cap B \neq \varnothing$, it follows that $B \neq \varnothing$. Hence, $B = \bigcup_{i \in I} A_i$ where I is a non-empty indexing set into *MinSet* and so $A \cap A_i \neq \varnothing$ for some $i \in I$. By S1(c) (definition 3.5), $A = A_i$ and so $A \subseteq B$. □

Proposition A.5. *Let $A, B \in AllSet$ and $\oplus \in \{-, \cup, \cap\}$. Then:*

1. *$A \oplus B \in AllSet$.*

2. *$\lambda(A \oplus B) = \lambda(A) \oplus \lambda(B)$.*

Proof. Let I, J be indexing sets into *MinSet* such that $A = \bigcup_{i \in I} A_i$ and $B = \bigcup_{j \in J} A_j$.

1. It is immediate by S1(c) (definition 3.5) that $A \oplus B = \bigcup_{k \in I \oplus J} A_k$ and so $A \oplus B \in AllSet$ by S1(c) (definition 3.6).

$$
\begin{aligned}
2.\quad \lambda(A \oplus B) &= \lambda(\textstyle\bigcup_{i\in I} A_i \oplus \bigcup_{j\in J} A_j) \\
&= \lambda(\textstyle\bigcup_{k\in I\oplus J} A_k) \\
&= \textstyle\bigcup_{k\in I\oplus J} \lambda(A_k) && \text{(by S5)} \\
&= \textstyle\bigcup_{i\in I} \lambda(A_i) \oplus \bigcup_{j\in J} \lambda(A_j) && \text{(by S1(c) (def. 3.5) and S4)} \\
&= \lambda(A) \oplus \lambda(B) && \text{(by S5)}
\end{aligned}
$$

□

Proposition A.6. *Let $A, A' \in AllSet$ be such that $A \cap A' = \varnothing$. Then $\lambda(A) \cap \lambda(A') = \varnothing$.*

Proof. Let $A = \bigcup_{i\in I} A_i$ and $A' = \bigcup_{j\in J} A_j$, where I, J are indexing sets into *MinSet* and $I \cap J = \varnothing$ since $A \cap A' = \varnothing$ (note that $\varnothing \notin$ *MinSet*). We consider each of two cases in turn.

Case 1: Either $A = \varnothing \neq A'$, $A' = \varnothing \neq A$ or $A = A' = \varnothing$. Wlog, we assume that $A = \varnothing$ and so $I = \varnothing$. The proof follows in this case by S5.

Case 2: $A \neq \varnothing \neq A'$ and so $I \neq \varnothing \neq J$. By S5, $\lambda(A) = \bigcup_{i\in I} \lambda(A_i)$ and $\lambda(A') = \bigcup_{j\in J} \lambda(A_j)$. Since $I \cap J = \varnothing$, $A_i \neq A_j$ for all $i \in I$, $j \in J$. By S1(c) (definition 3.5(2)) and S4, $\lambda(A_i) \cap \lambda(A_j) = \varnothing$ for all $i \in I$, $j \in J$ and so the proof in this case follows. □

Proposition A.7. *Let $t \circ \langle a \rangle$ be a trace such that $\lambda(t \circ \langle a \rangle)$ is defined. Then there exists $A \in MinSet$ such that $a \in A$.*

Proof. Since $\lambda(t \circ \langle a \rangle)$ is defined, then $events(t \circ \langle a \rangle) \subseteq \Sigma_{impl}$ by S1(b) (definition 3.4(1)). Hence, by S1(c) (definition 3.5), there exists $A \in MinSet$ such that $a \in A$. □

Proposition A.8. *Let $t \circ \langle a \rangle$ be a trace such that $\lambda(t \circ \langle a \rangle)$ is defined and $\lambda(t \circ \langle a \rangle) = \lambda(t) \circ r$.*

1. *If $a \in A \in AllSet$, then $events(r) \subseteq \lambda(A)$.*

2. *If $a \notin A \in AllSet$, then $events(r) \cap \lambda(A) = \varnothing$.*

Proof. By proposition A.7, there exists $A' \in MinSet$ such that $a \in A'$ and so $events(r) \subseteq \lambda(A')$ by Ts4.

1. We assume $a \in A \in AllSet$. By proposition A.4, $A' \subseteq A$ and so $\lambda(A') \subseteq \lambda(A)$ by S1(c) (definition 3.5) and S5.

2. We assume $a \notin A \in AllSet$. By proposition A.4, $A' \cap A = \varnothing$. Hence, by proposition A.6, $\lambda(A') \cap \lambda(A) = \varnothing$. □

Proposition A.9. *$\lambda(\langle\rangle)$ is defined and $\lambda(\langle\rangle) = \langle\rangle$.*

Proof. By S3 and S1(c) (definition 3.6), there exists $A \in \mathit{MinSet}$ such that $A \subseteq \mathit{Fvis}$. Since $\mathit{events}(\langle\rangle) = \varnothing$, we have that $\mathit{events}(\langle\rangle) \subseteq A$. The proof follows by Ts2. □

Proposition A.10. *Let t be a trace such that $\mathit{events}(t) \subseteq \mathit{Fvis}$. Then $\lambda(t)$ is defined and $\lambda(t) = t$.*

Proof. We proceed by induction on the length of t. In the base case, when $t = \langle\rangle$, the proof is immediate by proposition A.9. Let $t = u \circ \langle a \rangle$. By S3 and S1(c) (definition 3.6), there exists $A \in \mathit{MinSet}$ such that $a \in A$ and $A \subseteq \mathit{Fvis}$. Hence, $\lambda(t \lceil A)$ is defined by Ts2. By the inductive hypothesis, $\lambda(u)$ is defined and so $\lambda(t)$ is defined by Ts3. By Ts4, $\lambda(t) = \lambda(u) \circ r$, where $\lambda(t \lceil A) = \lambda(u \lceil A) \circ r$, and so $r = \langle a \rangle$ by Ts2. By the inductive hypothesis, $\lambda(u) = u$ and so $\lambda(t) = u \circ \langle a \rangle = t$. □

Proposition A.11. *Let t be a trace such that $\lambda(t)$ is defined and let $A \in \mathit{AllSet}$, where $B = \lambda(A)$. Then $\lambda(t \setminus A)$ is defined and $\lambda(t \setminus A) = \lambda(t) \setminus B$.*

Proof. We proceed by induction on the length of t. In the base case, when $t = \langle\rangle$, the proof is immediate by proposition A.9. Let $t = u \circ \langle a \rangle$. Hence, by proposition A.7, there exists $A' \in \mathit{MinSet}$ such that $a \in A'$. We consider each of two cases in turn.

Case 1: $a \in A$. In this case, $\lambda(t \setminus A) = \lambda(u \setminus A)$ and so $\lambda(t \setminus A)$ is defined by the inductive hypothesis. By Ts4, $\lambda(t) = \lambda(u) \circ r$ such that, by proposition A.8, $\mathit{events}(r) \subseteq B$. Hence, by the inductive hypothesis,

$$\lambda(t \setminus A) = \lambda(u \setminus A) = \lambda(u) \setminus B = (\lambda(u) \circ r) \setminus B = \lambda(t) \setminus B.$$

Case 2: $a \notin A$. Since $A' \not\subseteq A$, $A' \cap A = \varnothing$ by proposition A.4. Hence, $(t \setminus A) \lceil A' = t \lceil A'$ and so $\lambda((t \setminus A) \lceil A')$ is defined by Ts3. Since $\lambda(u \setminus A)$ is defined by the inductive hypothesis, then $\lambda(t \setminus A)$ is defined also by Ts3. By Ts4, $\lambda(t) = \lambda(u) \circ r$ such that $\lambda(t \lceil A') = \lambda(u \lceil A') \circ r$. Also by Ts4, $\lambda(t \setminus A) = \lambda(u \setminus A) \circ x$, such that $\lambda((t \setminus A) \lceil A') = \lambda((u \setminus A) \lceil A') \circ x$. Thus, since $A' \cap A = \varnothing$, $\lambda(t \setminus A) = \lambda(u \setminus A) \circ r$. By proposition A.8, $\mathit{events}(r) \cap B = \varnothing$. Hence, by the inductive hypothesis,

$$\lambda(t \setminus A) = \lambda(u \setminus A) \circ r = (\lambda(u) \setminus B) \circ r = (\lambda(u) \circ r) \setminus B = \lambda(t) \setminus B.$$

□

Proposition A.12. *Let s, u be traces such that $\lambda(s)$ and $\lambda(u)$ are defined. Let $Y \in \mathit{AllSet}$ be such that $[[\mathit{events}(s)]] \cap [[\mathit{events}(u)]] \subseteq Y$, where $\lambda(Y) = Z$. If $t \in s \parallel_Y u$, then $\lambda(t)$ is defined and $\lambda(t) \in \lambda(s) \parallel_Z \lambda(u)$.*

Proof. Let $t \in (s \parallel_Y u)$. Note by TRP that $t\lceil Y = s\lceil Y = u\lceil Y$. We proceed by induction on the length of t. In the base case, when $t = s = u = \langle\rangle$, the proof is immediate by proposition A.9. Let $t = p \circ \langle a\rangle$. We consider each of two cases in turn.

Case 1: $a \in Y$. In this case, $s = v \circ \langle a\rangle$ and $u = w \circ \langle a\rangle$ for some v, w. Hence, by proposition A.7, there exists $A \in \mathit{MinSet}$ such that $a \in A$. Since $A \cap Y \neq \varnothing$, $A \subseteq Y$ by proposition A.4. Thus, $t\lceil A = s\lceil A = u\lceil A$. Hence, since $\lambda(s)$ is defined, $\lambda(t\lceil A)$ is defined by TS3. By the inductive hypothesis, $\lambda(p)$ is defined and so $\lambda(t)$ is defined by TS3. By TS4 and since $t\lceil A = s\lceil A = u\lceil A$, $\lambda(t) = \lambda(p) \circ r$, $\lambda(s) = \lambda(v) \circ r$ and $\lambda(u) = \lambda(w) \circ r$ such that, by proposition A.8, $\mathit{events}(r) \subseteq Z$. By the inductive hypothesis, $\lambda(p) \in \lambda(v) \parallel_Z \lambda(w)$ and so $\lambda(p) \circ r \in (\lambda(v) \circ r \parallel_Z \lambda(w) \circ r)$.

Case 2: $a \notin Y$. Wlog, we assume that $s = v \circ \langle a\rangle$. Hence, by proposition A.7, there exists $A \in \mathit{MinSet}$ such that $a \in A$. Since $A \nsubseteq Y$, then $A \cap Y = \varnothing$ by proposition A.4. Since $A \cap \mathit{events}(s) \neq \varnothing$, $A \subseteq [[\mathit{events}(s)]]$ by proposition A.4. Since $[[\mathit{events}(s)]] \cap [[\mathit{events}(u)]] \subseteq Y$, then $A \nsubseteq [[\mathit{events}(u)]]$. Hence, by proposition A.4, $A \cap [[\mathit{events}(u)]] = \varnothing$ and so $t\lceil A = s\lceil A$. Since $\lambda(s)$ is defined, $\lambda(t\lceil A)$ is defined by TS3. Thus, by TS3 and the inductive hypothesis, $\lambda(t)$ is defined. By TS4 and since $t\lceil A = s\lceil A$, $\lambda(t) = \lambda(p) \circ r$ and $\lambda(s) = \lambda(v) \circ r$ such that, by proposition A.8, $\mathit{events}(r) \cap Z = \varnothing$. By the inductive hypothesis, $\lambda(p) \in \lambda(v) \parallel_Z \lambda(u)$. It therefore follows that $\lambda(p) \circ r \in (\lambda(v) \circ r \parallel_Z \lambda(u))$. □

Proposition A.13. *Let Q be an implementation process. If $\alpha Q \subseteq \mathit{Fvis}$, then $\lambda(\tau Q)$ is defined and $\lambda(\tau Q) = \tau Q$.*

Proof. We assume that $\alpha Q \subseteq \mathit{Fvis}$ and so, by PA1, $\mathit{events}(t) \subseteq \mathit{Fvis}$ for every $t \in \tau Q$. The proof follows by TS1 and proposition A.10. □

Proposition A.14. *Let Q be an implementation process. If $\lambda(\tau Q)$ is defined, $A \in \mathit{AllSet}$ and $\lambda(\backslash A) = \backslash B$, then:*

1. *$\lambda(\tau(Q \setminus A))$ is defined.*
2. *$\lambda(\tau(Q \setminus A)) = \lambda(\tau Q) \setminus B$.*

Proof. We assume that $\lambda(\tau Q)$ is defined, $A \in \mathit{AllSet}$ and $\lambda(\backslash A) = \backslash B$. By S6, $B = \lambda(A)$. Let $t \in \tau Q$. Then $\lambda(t)$ is defined by TS1. Also by TS1, it suffices to show that $\lambda(t \setminus A)$ is defined and $\lambda(t \setminus A) = \lambda(t) \setminus B$, which follows by proposition A.11. □

Proposition A.15. *Let P, Q be implementation processes. If $\lambda(\tau P)$, $\lambda(\tau Q)$ are defined and $Y = \alpha P \cap \alpha Q$, where $\lambda(\parallel_Y) = \parallel_Z$, then:*

1. *$\lambda(\tau(P \parallel_Y Q))$ is defined.*

2. $\lambda(\tau(P \parallel_Y Q)) \subseteq \lambda(\tau P) \parallel_Z \lambda(\tau Q)$.

Proof. We assume that $\lambda(\tau P)$, $\lambda(\tau Q)$ are defined and $Y = \alpha P \cap \alpha Q$, where $\lambda(\parallel_Y) = \parallel_Z$. That $Y \in \mathit{AllSet}$ follows by S2 and proposition A.5(1). By S7, $Z = \lambda(Y)$. Let $t \in \tau(P \parallel_Y Q)$ be such that $t \in (s \parallel_Y u)$ for $s \in \tau P$ and $u \in \tau Q$. By TS1, $\lambda(s)$ and $\lambda(u)$ are defined. Also by TS1, it suffices to show that $\lambda(t)$ is defined and $\lambda(t) \in \lambda(s) \parallel_Z \lambda(u)$. By PA1, $\mathit{events}(s) \subseteq \alpha P$ and so $[[\mathit{events}(s)]] \subseteq \alpha P$ since $\alpha P \in \mathit{AllSet}$. Similarly, $[[\mathit{events}(u)]] \subseteq \alpha Q$. Hence, $[[\mathit{events}(s)]] \cap [[\mathit{events}(u)]] \subseteq \alpha P \cap \alpha Q = Y$. The proof follows by proposition A.12. □

Proof of theorem 3.20

Proof. The proof is similar to that of theorem 3.1, using propositions A.13, A.14 and A.15 in place of conditions RAH1-3. □

A.4 Proofs from section 3.5

Proposition A.16. *Let Q be an implementation process. $\lambda([\![Q]\!]_T)$ is defined if and only if $\lambda([\![Q]\!]_{SF})$ is defined.*

Proof. ($\Longrightarrow$) We assume that $\lambda([\![Q]\!]_T)$ is defined. Let $(t, R) \in \phi Q$. By definition 3.12 (parts 1 and 3a), it suffices to show that $\lambda(R \cap \alpha Q, t)$ is defined. By SF2, $t \in \tau Q$ and so $\lambda(t)$ is defined by definition 3.3. Since $\alpha Q \in \mathit{AllSet}$, $R \cap \alpha Q \subseteq \Sigma_{impl}$ by proposition 3.9(3). Hence, by definition 3.12(3), $\lambda(R \cap \alpha Q, t)$ is defined.

($\Longleftarrow$) We assume that $\lambda([\![Q]\!]_{SF})$ is defined. The proof follows immediately by definition 3.12(1). □

Proof of theorem 3.21

Proof. We consider TI1, TI2 and TI3 in turn.

1. Let Q be an implementation process such that $\alpha Q \subseteq \mathit{Fvis}$. By TI1, we have to show that $\lambda(\tau Q)$ is defined and $\lambda(\tau Q) = \tau Q$. By SFI1, $\lambda((\tau Q, \phi Q))$ is defined and $\lambda((\tau Q, \phi Q)) = (\tau Q, \phi Q)$. Hence, by definition 3.12(1), $\lambda(\tau Q)$ is defined. Moreover, by definition 3.12(2), $(\lambda(\tau Q), \lambda(\phi Q)) = (\tau Q, \phi Q)$.

2. Let Q be an implementation process such that $\lambda(\tau Q)$ is defined. Moreover, let $A \in \mathit{AllSet}$ and $\lambda(\setminus A) = \setminus B$. By TI2, we have to show that $\lambda(\tau(Q \setminus A))$ is defined and $\lambda(\tau(Q \setminus A)) = \lambda(\tau Q) \setminus B$. By proposition A.16, $\lambda([\![Q]\!]_{SF})$ is defined. Hence, by SFI2, $\lambda([\![Q \setminus A]\!]_{SF})$ is defined and so, by definition 3.12(1), $\lambda(\tau(Q \setminus A))$ is defined. Also by SFI2, $\lambda([\![Q \setminus A]\!]_{SF}) = \lambda([\![Q]\!]_{SF}) \setminus B$. Thus, by definition 3.12(2),

$$(\lambda(\tau(Q \setminus A)), \lambda(\phi(Q \setminus A))) = (\lambda(\tau Q) \setminus B, \lambda(\phi Q) \setminus B).$$

3. Let P, Q be implementation processes such that $\lambda(\tau P)$ and $\lambda(\tau Q)$ are defined. Moreover, let $Y = \alpha P \cap \alpha Q$ and $\lambda(\|_Y) = \|_Z$. By Ti3, we have to show that $\lambda(\tau(P \|_Y Q))$ is defined and $\lambda(\tau(P \|_Y Q)) = \lambda(\tau P) \|_Z \lambda(\tau Q)$. By proposition A.16, $\lambda([\![P]\!]_{SF})$ and $\lambda([\![Q]\!]_{SF})$ are defined. Hence, by Sfi3, $\lambda([\![P \|_Y Q]\!]_{SF})$ is defined and so, by definition 3.12(1), $\lambda(\tau(P \|_Y Q))$ is defined. Also by Sfi3, $\lambda([\![P \|_Y Q]\!]_{SF}) = \lambda([\![P]\!]_{SF}) \|_Z \lambda([\![Q]\!]_{SF})$. Thus, by definition 3.12(2),

$$(\lambda(\tau(P \|_Y Q)), \lambda(\phi(P \|_Y Q))) = (\lambda(\tau P) \|_Z \lambda(\tau Q), \lambda(\phi P) \|_Z \lambda(\phi Q)).$$

□

Proof of theorem 3.22

Proof. By definition 3.12(3b), both $\lambda(\phi P)$ and $\lambda(\phi Q)$ are subset-closed. Also by definition 3.12(3b),

$$(t, R) \in \lambda(\phi P) \Longrightarrow (t, R \cup (\Sigma - \lambda(\alpha P))) \in \lambda(\phi P).$$

Hence, $\lambda(\alpha P) \in \mathcal{R}(\lambda(\phi P))$ by definition 2.5 and, similarly, $\lambda(\alpha Q) \in \mathcal{R}(\lambda(\phi Q))$. Since $Y = \alpha P \cap \alpha Q$ and $\alpha P, \alpha Q \in \mathit{AllSet}$, then $Z = \lambda(\alpha P) \cap \lambda(\alpha Q)$ by proposition 3.8(2). The proof follows by proposition 2.19. □

Proof of proposition 3.23

Proof. Let Q be an implementation process such that $\alpha Q \subseteq \mathit{Fvis}$. By Sfi1, $\lambda((\tau Q, \phi Q))$ is defined and $\lambda((\tau Q, \phi Q)) = (\tau Q, \phi Q)$. Hence, by definition 3.12(1), $\lambda(\phi Q)$ is defined. Moreover, by definition 3.12(2), $(\lambda(\tau Q), \lambda(\phi Q)) = (\tau Q, \phi Q)$. □

Proof of proposition 3.24

Proof. Let Q be an implementation process such that $\lambda([\![Q]\!]_{SF})$ is defined. Moreover, let $A \in \mathit{AllSet}$ and $\lambda(A) = B$. By theorem 3.16, $\lambda(\backslash A) = \backslash B$. By Sfi2, $\lambda([\![Q \setminus A]\!]_{SF})$ is defined and so, by definition 3.12(1), $\lambda(\phi(Q \setminus A))$ is defined. Also by Sfi2, $\lambda([\![Q \setminus A]\!]_{SF}) = \lambda([\![Q]\!]_{SF}) \setminus B$. Thus, by definition 3.12(2),

$$(\lambda(\tau(Q \setminus A)), \lambda(\phi(Q \setminus A))) = (\lambda(\tau Q) \setminus B, \lambda(\phi Q) \setminus B).$$

□

Proof of proposition 3.25

Proof. Let P, Q be implementation processes such that $\lambda([P]_{SF})$ and $\lambda([Q]_{SF})$ are defined. Moreover, let $Y = \alpha P \cap \alpha Q$ and $\lambda(Y) = Z$. Since $\alpha P, \alpha Q \in$ *AllSet*, then $Y \in$ *AllSet* by proposition 3.8(1). By theorem 3.17, $\lambda(\|_Y) = \|_Z$. By SFI3, $\lambda([\![P \|_Y Q]\!]_{SF})$ is defined and so, by definition 3.12(1), $\lambda(\phi(P \|_Y Q))$ is defined. Also by SFI3,

$$\lambda([\![P \|_Y Q]\!]_{SF}) = \lambda([\![P]\!]_{SF}) \|_Z \lambda([\![Q]\!]_{SF}).$$

Thus, by definition 3.12(2),

$$(\lambda(\tau(P \|_Y Q)), \lambda(\phi(P \|_Y Q))) = (\lambda(\tau P) \|_Z \lambda(\tau Q), \lambda(\phi P) \|_Z \lambda(\phi Q)).$$

□

Proof of theorem 3.26

Proof. Before proceeding with the proof proper, we show the following result.

$$\lambda(Fvis) = Fvis. \tag{A.3}$$

By proposition 3.12, $Fvis \in AllSet$ and so, by definition 3.8,

$$\lambda(Fvis) = \bigcup\{events(\lambda(t)) \mid t \in BTrace \wedge events(t) \subseteq Fvis\}.$$

Hence, by RAH1-T and proposition 3.5, $\lambda(Fvis) = Fvis$.

We now proceed with the proof proper. Let $T_{Fvis} = \{\langle a \rangle \mid a \in Fvis\}$ (recall that $Fvis \neq \varnothing$ and so $T_{Fvis} \neq \varnothing$). Let $Q = SFT\langle(t, R \cup (\Sigma - Fvis)), T_{Fvis}\rangle$ be a (component) implementation process. By proposition 2.13(3),

$$\beta(Q) = events(t) \cup (\Sigma - (R \cup (\Sigma - Fvis))) \cup \bigcup\{events(u) \mid u \in T_{Fvis}\}.$$

We then observe that:

- $events(t) \subseteq Fvis$.
- $\Sigma - (R \cup (\Sigma - Fvis)) \subseteq Fvis$.
- $\bigcup\{events(u) \mid u \in T_{Fvis}\} = Fvis$.

Hence, $\beta(Q) = Fvis$ and so $\alpha Q = Fvis$ by proposition 3.12. Thus, by proposition 3.23, $\lambda(\phi Q)$ is defined and $\lambda(\phi Q) = \phi Q$. By proposition 2.13(2), $\phi Q = \{(t, X) \mid X \subseteq R \cup (\Sigma - Fvis)\}$. Thus, by definition 3.12(3a) and since $R \subseteq Fvis = \alpha Q$, $\lambda(R, t)$ is defined. By definition 3.12(3b) and REF-MONO, $max(\lambda(\phi Q)) = (\lambda(t), \lambda(R, t) \cup (\Sigma - \lambda(\alpha Q)))$. We also observe that $max(\phi Q) = (t, R \cup (\Sigma - Fvis))$. Hence, since $\lambda(\phi Q) = \phi Q$ and since $\lambda(\alpha Q) = \lambda(Fvis) = Fvis$ by (A.3),

$$R \cup (\Sigma - Fvis) = \lambda(R,t) \cup (\Sigma - Fvis).$$

It is immediate that $R \cap (\Sigma - Fvis) = \varnothing$. By REF-BOUND, we observe that $\lambda(R,t) \subseteq \lambda(Fvis) = Fvis$ and so $\lambda(R,t) \cap (\Sigma - Fvis) = \varnothing$. Hence, $R = \lambda(R,t)$. □

Proof of theorem 3.27

Proof. Let

- $M = MFP\langle(s, S \cup (\Sigma - \alpha P)), P\rangle$ and
- $N = MFP\langle(u, U \cup (\Sigma - \alpha Q)), Q\rangle$

be (component) implementation processes. By proposition 2.15,

- $\beta(M) = \beta(P)$ and so $\alpha M = \alpha P$.
- $\tau M = \tau P$.
- $\phi M = \{(s,X) \mid X \subseteq S \cup (\Sigma - \alpha P)\} = \{(s,X) \mid X \subseteq S \cup (\Sigma - \alpha M)\}$.

Since $\lambda([\![P]\!]_{SF})$ is defined, then $\lambda(\tau P)$ and $\lambda(\phi P)$ are defined by definition 3.12(1). Hence, $\lambda(\tau M)$ is defined and so $\lambda([\![M]\!]_{SF})$ is defined by proposition A.16 ($\lambda(\phi M)$ is also defined by definition 3.12(1)). Similarly,

- $\alpha N = \alpha Q$.
- $\phi N = \{(u,X) \mid X \subseteq U \cup (\Sigma - \alpha Q)\} = \{(u,X) \mid X \subseteq U \cup (\Sigma - \alpha N)\}$.
- $\lambda([\![N]\!]_{SF})$ is defined (and so $\lambda(\phi N)$ is defined).

We observe that $Y = \alpha P \cap \alpha Q = \alpha M \cap \alpha N$ and so $Y \in AllSet$ by proposition 3.8(1). Hence, $\lambda(\phi(M \parallel_Y N))$ is defined by proposition 3.25(1). Since $\{t\} = (s \parallel_Y u)$, $S \subseteq \alpha P = \alpha M$ and $U \subseteq \alpha Q = \alpha N$, $(t, S \cup U) \in \phi(M \parallel_Y N)$ by theorem 2.20. Moreover, $\alpha(M \parallel_Y N) = \alpha M \cup \alpha N$ by proposition 3.3(2). Thus, $S \cup U \subseteq \alpha(M \parallel_Y N)$. Hence, since $\lambda(\phi(M \parallel_Y N))$ is defined and by definition 3.12(3a), $\lambda(S \cup U, t)$ is defined.

We now show that $\lambda(S \cup U, t) = \lambda(S,s) \cup \lambda(U,u)$. For $(w,X) \in \phi M$, $X \cap \alpha M \subseteq S$. Similarly, for $(w,X) \in \phi N$, $X \cap \alpha N \subseteq U$. Moreover, $S \subseteq \alpha M$, $U \subseteq \alpha N$ and $Y = \alpha M \cap \alpha N$. Hence, by theorem 2.20 and since $\{t\} = (s \parallel_Y u)$,

$$\phi(M \parallel_Y N) = \{(t,X) \mid X \subseteq S \cup U \cup (\Sigma - (\alpha M \cup \alpha N))\}.$$

Recall that $\alpha(M \parallel_Y N) = \alpha M \cup \alpha N$ and $S \cup U \subseteq \alpha M \cup \alpha N$. Thus, by definition 3.12(3b) and REF-MONO,

$$max(\lambda(\phi(M \parallel_Y N))) = \{(\lambda(t), \lambda(S \cup U, t) \cup (\Sigma - \lambda(\alpha M \cup \alpha N)))\} \quad \text{(A.4)}$$

We then observe that, by definition 3.12(3a,3b), REF-MONO and since $S \subseteq \alpha M$ and $U \subseteq \alpha N$:

- $\lambda(\phi M) = \{(\lambda(s), X) \mid X \subseteq \lambda(S, s) \cup (\Sigma - \lambda(\alpha M))\}$.
- $\lambda(\phi N) = \{(\lambda(u), X) \mid X \subseteq \lambda(U, u) \cup (\Sigma - \lambda(\alpha N))\}$.

Since $\lambda(\phi M)$ is defined, $\lambda(S, s)$ is defined by definition 3.12(3a). Similarly, $\lambda(U, u)$ is defined. By SF2 and PA1, *events*$(s) \subseteq \alpha M$ and *events*$(u) \subseteq \alpha N$. Hence, by REF-BOUND, $\lambda(S, s) \subseteq \lambda(\alpha M)$ and $\lambda(U, u) \subseteq \lambda(\alpha N)$. Let $Z = \lambda(Y)$. Thus, by theorem 3.22,

$$\begin{aligned}\lambda(\phi M) \parallel_Z \lambda(\phi N) \quad = \quad & \{(w, X) \mid w \in \lambda(s) \parallel_Z \lambda(u) \wedge \\ & X \subseteq \lambda(S, s) \cup \lambda(U, u) \cup (\Sigma - (\lambda(\alpha M) \cup \lambda(\alpha N)))\}.\end{aligned}$$

It is then immediate that the following holds:

$$\begin{aligned}max(\lambda(\phi M) \parallel_Z \lambda(\phi N)) = \{(w, \lambda(S, s) \cup \lambda(U, u) \cup \Sigma - (\lambda(\alpha M) \cup \lambda(\alpha N))) \mid \\ w \in \lambda(s) \parallel_Z \lambda(u)\}\end{aligned} \quad \text{(A.5)}$$

By proposition 3.25(2), $\lambda(\phi(M \parallel_Y N)) = \lambda(\phi M) \parallel_Z \lambda(\phi N)$. Moreover, by proposition 3.8(2), $\lambda(\alpha M \cup \alpha N) = \lambda(\alpha M) \cup \lambda(\alpha N)$. Hence, by (A.4) and (A.5),

$$\lambda(S, s) \cup \lambda(U, u) \cup (\Sigma - \lambda(\alpha M \cup \alpha N)) = \lambda(S \cup U, t) \cup (\Sigma - \lambda(\alpha M \cup \alpha N)).$$

By SF2 and PA1, *events*$(t) \subseteq \alpha(M \parallel_Y N)$. Since $\alpha(M \parallel_Y N) = \alpha M \cup \alpha N$, *events*$(t) \cup S \cup U \subseteq \alpha M \cup \alpha N$. Hence, since $\lambda(S \cup U, t)$ is defined, $\lambda(S \cup U, t) \subseteq \lambda(\alpha M \cup \alpha N)$ by REF-BOUND. Since $\lambda(S, s) \subseteq \lambda(\alpha M)$ and $\lambda(U, u) \subseteq \lambda(\alpha N)$,

$$\lambda(S, s) \cup \lambda(U, u) \subseteq \lambda(\alpha M) \cup \lambda(\alpha N) = \lambda(\alpha M \cup \alpha N).$$

Hence, $\lambda(S, s) \cup \lambda(U, u) = \lambda(S \cup U, t)$. □

Lemma A.17. *Let P be an implementation process such that $\lambda([\![P]\!]_{SF})$ is defined. Let $(t, R \cup (\Sigma - \alpha P)) \in \phi P$ be a refusal-maximal failure such that $R \subseteq \alpha P$. Let $A \in AllSet$ and $\lambda(A) = B$. If $A \subseteq R$ then:*

1. $B \subseteq \lambda(R, t)$.

2. $\lambda(R - A, t \setminus A)$ is defined and $\lambda(R - A, t \setminus A) = \lambda(R, t) - B$.

Proof. We assume that $A \subseteq R$. Let $Q = \mathit{MFP}\langle(t, R \cup (\Sigma - \alpha P)), P\rangle$ be a (component) implementation process. By proposition 2.15,

- $\beta(Q) = \beta(P)$ and so $\alpha Q = \alpha P$.
- $\tau Q = \tau P$.
- $\phi Q = \{(t, X) \mid X \subseteq R \cup (\Sigma - \alpha P)\} = \{(t, X) \mid X \subseteq R \cup (\Sigma - \alpha Q)\}$.

Since $\lambda([\![P]\!]_{SF})$ is defined, then $\lambda(\tau P)$ and $\lambda(\phi P)$ are defined by definition 3.12(1). Hence, $\lambda(\tau Q)$ is defined and so $\lambda([\![Q]\!]_{SF})$ is defined by proposition A.16. By definition 3.12(1), this also means that $\lambda(\phi Q)$ is defined. We now proceed with the proof proper.

1. By proposition 3.24(1), $\lambda(\phi(Q \setminus A))$ is defined. Since $A \subseteq R$, $\phi(Q \setminus A) \neq \varnothing$ and so, by definition 3.12(3b), $\lambda(\phi(Q \setminus A)) \neq \varnothing$. Hence, by proposition 3.24(2), $\lambda(\phi Q) \setminus B \neq \varnothing$. By definition 3.12(3a,3b), REF-MONO and since $R \subseteq \alpha Q$,

$$\lambda(\phi Q) = \{(\lambda(t), X) \mid X \subseteq \lambda(R, t) \cup (\Sigma - \lambda(\alpha Q))\}.$$

Thus, $B \subseteq \lambda(R, t) \cup (\Sigma - \lambda(\alpha Q))$. Let I, J be indexing sets into *MinSet* such that $A = \bigcup_{i \in I} A_i$ and $\alpha Q = \bigcup_{j \in J} A_j$. Since $A \subseteq R \subseteq \alpha Q$ and by definition 3.5, $I \subseteq J$. Hence, by proposition 3.6, $B \subseteq \lambda(\alpha Q)$ and so $B \subseteq \lambda(R, t)$.

2. Since $A \subseteq R \subseteq \alpha Q$ and $\alpha(Q \setminus A) = \alpha Q - A$ by proposition 3.3(1),

$$\begin{aligned} \phi(Q \setminus A) &= \{(t \setminus A, X) \mid X \subseteq R \cup (\Sigma - \alpha Q)\} \\ &= \{(t \setminus A, X) \mid X \subseteq (R - A) \cup (\Sigma - \alpha(Q \setminus A))\}. \end{aligned}$$

Since $R \subseteq \alpha Q$ and $\alpha(Q \setminus A) = \alpha Q - A$, then $R - A \subseteq \alpha(Q \setminus A)$. By proposition 3.24(1), $\lambda(\phi(Q \setminus A))$ is defined. Hence, by definition 3.12(3a), $\lambda(R - A, t \setminus A)$ is defined. Also, by definition 3.12(3b) and REF-MONO,

$$\lambda(\phi(Q \setminus A)) = \{(\lambda(t \setminus A), X) \mid X \subseteq \lambda(R - A, t \setminus A) \cup (\Sigma - \lambda(\alpha(Q \setminus A)))\}.$$

By proposition 3.8(2), $\lambda(\alpha(Q \setminus A)) = \lambda(\alpha Q - A) = \lambda(\alpha Q) - B$. Moreover, by the proof of part 1 of the lemma, $B \subseteq \lambda(\alpha Q)$. Hence,

$$\Sigma - \lambda(\alpha(Q \setminus A)) = (\Sigma - \lambda(\alpha Q)) \cup (B \cap \lambda(\alpha Q)) = (\Sigma - \lambda(\alpha Q)) \cup B.$$

Thus, we have that

$$\lambda(\phi(Q \setminus A)) = \{(\lambda(t \setminus A), X) \mid X \subseteq \lambda(R - A, t \setminus A) \cup B \cup (\Sigma - \lambda(\alpha Q))\}.$$

and so it is immediate that

$$max(\lambda(\phi(Q \setminus A))) = (\lambda(t \setminus A), \lambda(R - A, t \setminus A) \cup B \cup (\Sigma - \lambda(\alpha Q))). \quad \text{(A.6)}$$

Since $\lambda(\phi Q) = \{(\lambda(t), X) \mid X \subseteq \lambda(R,t) \cup (\Sigma - \lambda(\alpha Q))\}$ and $B \subseteq \lambda(R,t)$ by (the proof of) part 1 of the lemma,

$$\lambda(\phi Q) \setminus B = \{(\lambda(t) \setminus B, X) \mid X \subseteq \lambda(R,t) \cup (\Sigma - \lambda(\alpha Q))\}$$

and so

$$max(\lambda(\phi Q) \setminus B) = (\lambda(t) \setminus B, \lambda(R,t) \cup (\Sigma - \lambda(\alpha Q))). \quad \text{(A.7)}$$

By proposition 3.24(2), $\lambda(\phi(Q \setminus A)) = \lambda(\phi Q) \setminus B$ and so, by (A.6) and (A.7),

$$\lambda(R - A, t \setminus A) \cup B \cup (\Sigma - \lambda(\alpha Q)) = \lambda(R,t) \cup (\Sigma - \lambda(\alpha Q)).$$

Since $\lambda(\phi Q)$ is defined and $R \subseteq \alpha Q$, then $\lambda(R,t)$ is defined by definition 3.12(3a); moreover, $events(t) \subseteq \alpha Q$ by SF2 and PA1. Hence, by REF-BOUND, $\lambda(R,t) \subseteq \lambda(\alpha Q)$. Recall that we have already shown that $B \subseteq \lambda(\alpha Q)$. Since $events(t \setminus A) \cup (R - A) \subseteq \alpha(Q \setminus A)$ and $\lambda(R - A, t \setminus A)$ is defined, then $\lambda(R - A, t \setminus A) \subseteq \lambda(\alpha(Q \setminus A)) = \lambda(\alpha Q) - B$ also by REF-BOUND. Hence,

$$\lambda(R - A, t \setminus A) = \lambda(R,t) - B.$$

□

Proposition A.18. *We assume the following, where $A \in AllSet$.*

- $T_A = \{u \in BTrace \mid events(t) \subseteq A\} \cup \{\langle\rangle\}$.
- $(t, R \cup (\Sigma - A))$ *is a failure such that* $events(t) \cup R \subseteq A$, $\lambda(t)$ *is defined and* $\lambda(t \circ \langle a \rangle)$ *is defined for every* $a \in A - R$.
- $Q = SFT\langle (t, R \cup (\Sigma - A)), T_A \rangle$ *is a (component) implementation process.*

Then:

1. $\lambda([\![Q]\!]_{SF})$ *is defined.*
2. $(t, R \cup (\Sigma - \alpha Q)) \in \phi Q$ *is refusal-maximal.*
3. $\alpha Q = A$ *and so* $R \subseteq \alpha Q$.

Proof. By proposition 2.13(3),

$$\beta(Q) = events(t) \cup (\Sigma - (R \cup (\Sigma - A))) \cup \bigcup\{events(u) \mid u \in T_A\}.$$

We then observe the following:

- $events(t) \subseteq A$.
- $\Sigma - (R \cup (\Sigma - A)) \subseteq A$.
- $\bigcup\{events(u) \mid u \in T_A\} = A$ by proposition 3.5.

Hence, $\beta(Q) = A$ and so $\alpha Q = A$, which means that $R \subseteq \alpha Q$. By proposition 2.13(1,2),

- $\phi Q = \{(t, X) \mid X \subseteq R \cup (\Sigma - A)\} = \{(t, X) \mid X \subseteq R \cup (\Sigma - \alpha Q)\}$.
- $\tau Q = Pref(T_A) \cup Pref(t) \cup \{t \circ \langle a \rangle \mid a \in \Sigma - (R \cup (\Sigma - A))\}$.

Hence, it is immediate that $(t, R \cup (\Sigma - \alpha Q)) \in \phi Q$ is refusal-maximal. By PREF-CLOS and theorem 3.14, $\lambda(u)$ is defined for every $u \in Pref(T_A) \cup Pref(t)$. We know that $\lambda(t \circ \langle a \rangle)$ is defined for every $a \in A - R$. Hence, by definition 3.3 and since $\Sigma - (R \cup (\Sigma - A)) = A - R$, $\lambda(\tau Q)$ is defined. Thus, by proposition A.16, $\lambda([\![Q]\!]_{SF})$ is defined. □

Proof of theorem 3.28

Proof. The proof of part 1 of the theorem is immediate by lemma A.17(1) and we therefore consider part 2. We assume the following:

- $T_A = \{u \in BTrace \mid events(u) \subseteq A\} \cup \{\langle\rangle\}$.
- $Q = MFTP\langle(t, R \cup A \cup (\Sigma - (\alpha P \cup A))), T_A, P\rangle$ is a (component) implementation process.

By proposition 2.16,

- $\tau Q = \tau P \cup Pref(T_A)$.
- $\phi Q = \phi P \cup \{(t, X) \mid X \subseteq R \cup A \cup (\Sigma - (\alpha P \cup A))\}$.
- $\beta(Q) = \beta(P) \cup \bigcup\{events(u) \mid u \in T_A\}$.

By proposition 3.5, $\beta(Q) = \beta(P) \cup A$ and so $\alpha Q = \alpha P \cup A$. Hence,

$$\phi Q = \phi P \cup \{(t, X) \mid X \subseteq R \cup A \cup (\Sigma - \alpha Q)\}.$$

Since $\lambda([\![P]\!]_{SF})$ is defined, $\lambda(\tau P)$ is defined by definition 3.12(1). By theorem 3.14 and PREF-CLOS, $\lambda(u)$ is defined for every $u \in Pref(T_A)$ and so, by definition 3.3, $\lambda(\tau Q)$ is defined. Hence, $\lambda([\![Q]\!]_{SF})$ is defined by proposition A.16.

Since $(t, R \cup (\Sigma - \alpha P)) \in \phi P$ is refusal-maximal, it is also the case that $(t, (R \cup A) \cup (\Sigma - \alpha Q)) \in \phi Q$ is refusal-maximal. Moreover, since $R \subseteq \alpha P$, then $R \cup A \subseteq \alpha Q$. Hence, by lemma A.17(2), $\lambda((R \cup A) - A, t \setminus A) = \lambda(R - A, t \setminus A)$ is defined and

$$\lambda(R - A, t \setminus A) = \lambda((R \cup A) - A), t \setminus A) = \lambda(R \cup A, t) - B. \qquad \text{(A.8)}$$

We also assume that $K = SFT\langle(t\lceil A, A \cup (\Sigma - A)), T_A\rangle$ is a (component) implementation process. Since $(t, \varnothing) \in \phi P$, then $t \in \tau P$ by SF2. Thus, $\lambda(t)$ is defined by definition 3.3 and so, by proposition A.3, $\lambda(t\lceil A)$ is defined. Thus, by proposition A.18,

- $\lambda([\![K]\!]_{SF})$ is defined.
- $(t\lceil A, A \cup (\Sigma - \alpha K)) \in \phi K$ is refusal-maximal.
- $\alpha K = A$.

Let $Y = \alpha P \cap \alpha K$. By SF2 and PA1, $events(t) \subseteq \alpha P$ and $events(t\lceil A) \subseteq \alpha K$ and so $events(t\lceil A) \subseteq Y$. Moreover, since $\alpha K = A$, we observe that $Y \subseteq A$ and so $events(t \setminus A) \cap Y = \varnothing$. Hence, $\{t\} = t \parallel_Y t\lceil A$. Recall that $\lambda([\![P]\!]_{SF})$ is defined, $(t, R \cup (\Sigma - \alpha P)) \in \phi P$ is refusal-maximal and $R \subseteq \alpha P$. As a result, by RAH3-SF:

$$\lambda(R \cup A, t) = \lambda(R, t) \cup \lambda(A, t\lceil A).$$

Thus, by (A.8),

$$\lambda(R - A, t \setminus A) = (\lambda(R, t) - B) \cup (\lambda(A, t\lceil A) - B).$$

Since $\lambda(t\lceil A)$ is defined, $\lambda(A, t\lceil A)$ is defined by proposition 3.9(3) and definition 3.12(3). Hence, by REF-BOUND, $\lambda(A, t\lceil A) \subseteq B$ and so

$$\lambda(R - A, t \setminus A) = \lambda(R, t) - B.$$

□

Proof of theorem 3.30

Proof. We assume that $\lambda(A, t)$ is defined and so $\lambda(t)$ is defined by definition 3.12(3). By REF-BOUND, $\lambda(A, t) \subseteq \lambda(A)$. Hence, it suffices to show that $\lambda(A) \subseteq \lambda(A, t)$. We assume the following:

- $T_A = \{u \in BTrace \mid events(u) \subseteq A\} \cup \{\langle\rangle\}$.
- $Q = SFT\langle(t, A \cup (\Sigma - A)), T_A\rangle$ is a (component) implementation process.

Thus, by proposition A.18,

- $\lambda([\![Q]\!]_{SF})$ is defined.
- $(t, A \cup (\Sigma - \alpha Q)) \in \phi Q$ is refusal-maximal.
- $\alpha Q = A$ and so $A \subseteq \alpha Q$.

Hence, by RAH2-SF(1), $\lambda(A) \subseteq \lambda(A, t)$. □

Proof of theorem 3.31

Proof. We first observe that, by definition 3.12(3) and since $\lambda(R, t)$ is defined, $\lambda(t)$ is defined. We assume the following:

- $T_A = \{u \in BTrace \mid events(u) \subseteq A\} \cup \{\langle\rangle\}$.
- $P = SFT\langle(t, R \cup (\Sigma - A)), T_A\rangle$ is a (component) implementation process.
- $Q = SFT\langle(t, S \cup (\Sigma - A)), T_A\rangle$ is a (component) implementation process.

Thus, by proposition A.18,

- $\lambda([\![P]\!]_{SF})$ is defined.
- $(t, R \cup (\Sigma - \alpha P)) \in \phi P$ is refusal-maximal.
- $\alpha P = A$ and so $R \subseteq \alpha P$.

Similarly,

- $\lambda([\![Q]\!]_{SF})$ is defined.
- $(t, S \cup (\Sigma - \alpha Q)) \in \phi Q$ is refusal-maximal.
- $\alpha Q = A$ and so $S \subseteq \alpha Q$.

We observe that $\alpha P \cap \alpha Q = A$ and $\{t\} = t \parallel_A t$. The proof then follows by RAH3-SF. □

Lemma A.19. *Let t be a trace and $R \subseteq \Sigma$ such that $\lambda(R, t)$ is defined and $\lambda(t \circ \langle a\rangle)$ is defined for every $a \in [[events(t) \cup R]] - R$. Let $A \in AllSet$, where $\lambda(A) = B$. Then:*

1. *$\lambda(R \cap A, t\lceil A)$ is defined.*
2. *$\lambda(R \cap A, t\lceil A) = \lambda(R, t) \cap B$.*

Proof. We first observe that, by definition 3.12(3) and since $\lambda(R,t)$ is defined, $\lambda(t)$ is defined. We assume the following:

- $T = \{u \in \mathit{BTrace} \mid \mathit{events}(u) \subseteq [[\mathit{events}(t) \cup R]]\} \cup \{\langle\rangle\}$.
- $Q = \mathit{SFT}\langle(t, R \cup (\Sigma - [[\mathit{events}(t) \cup R]])), T\rangle$ is a (component) implementation process.

Thus, by proposition A.18,

- $\lambda([\![Q]\!]_{SF})$ is defined.
- $(t, R \cup (\Sigma - \alpha Q)) \in \phi Q$ is refusal-maximal.
- $\alpha Q = [[\mathit{events}(t) \cup R]]$ and so $R \subseteq \alpha Q$.

Let $\overline{A} = \Sigma_{impl} - A$ and $\overline{B} = \Sigma_{spec} - B$. By proposition A.2, $\overline{A} \in \mathit{AllSet}$ and $\overline{B} = \lambda(\overline{A})$. Thus, by RAH2-SF(2), $\lambda(R - \overline{A}, t \setminus \overline{A})$ is defined and

$$\lambda(R - \overline{A}, t \setminus \overline{A}) = \lambda(R,t) - \overline{B}.$$

By proposition 3.9(3), $A \subseteq \Sigma_{impl}$ and $B \subseteq \Sigma_{spec}$. Hence, $\Sigma_{impl} = A \cup \overline{A}$ and $\Sigma_{spec} = B \cup \overline{B}$. By definition 3.12(3), $R \subseteq \Sigma_{impl}$ and so $R - \overline{A} = R \cap A$. Since $\mathit{events}(t) \subseteq \Sigma_{impl}$ by definition 3.4(1), then $t \setminus \overline{A} = t \lceil A$. Thus, $\lambda(R \cap A, t\lceil A)$ is defined. By proposition 3.9(1,3), $\Sigma_{impl} \in \mathit{AllSet}$ and $\lambda(\Sigma_{impl}) \subseteq \Sigma_{spec}$. Thus, by REF-BOUND, $\lambda(R,t) \subseteq \Sigma_{spec}$ and so $\lambda(R,t) - \overline{B} = \lambda(R,t) \cap B$. Hence, $\lambda(R \cap A, t\lceil A) = \lambda(R,t) \cap B$. □

Proof of theorem 3.32

Proof. 1. The proof is immediate by lemma A.19(1).

2. We observe that, by definition 3.12(3) and since $\lambda(R,t)$ is defined, $R \subseteq \Sigma_{impl}$ and $\lambda(t)$ is defined. Thus, $\mathit{events}(t) \subseteq \Sigma_{impl}$ by definition 3.4(1). By proposition 3.9(1,3), $\Sigma_{impl} \in \mathit{AllSet}$ and $\lambda(\Sigma_{impl}) \subseteq \Sigma_{spec}$. Thus, by REF-BOUND, $\lambda(R,t) \subseteq \Sigma_{spec}$. Moreover, by proposition 3.9(2), $\Sigma_{spec} = \bigcup_{A \in \mathit{MinSet}} \lambda(A)$. We therefore observe that:

$$\begin{aligned} \lambda(R,t) &= \lambda(R,t) \cap \textstyle\bigcup_{A \in \mathit{MinSet}} \lambda(A) \\ &= \textstyle\bigcup_{A \in \mathit{MinSet}} \lambda(R,t) \cap \lambda(A) \\ &= \textstyle\bigcup_{A \in \mathit{MinSet}} \lambda(R \cap A, t\lceil A) \qquad \text{(by lemma A.19(2)).} \end{aligned}$$

□

Results used in the proof of theorem 3.33

In all of the proofs in the remainder of this section, we assume that the conditions from figures 3.4 and 3.6 all hold.

Proposition A.20. *$\lambda(\varnothing, \langle\rangle)$ is defined and $\lambda(\varnothing, \langle\rangle) = \varnothing$.*

Proof. By S3, $Fvis \in AllSet$ and $Fvis \neq \varnothing$. Hence, by S1(c) (definition 3.6), there exists $A \in MinSet$ such that $A \subseteq Fvis$. It is immediate that $events(\langle\rangle) \cup \varnothing \subseteq A$ and so the proof follows by SFS2. □

Lemma A.21. *Let t be a trace and $R \subseteq \Sigma$. If $events(t) \cup R \subseteq Fvis$, then $\lambda(R,t)$ is defined and $\lambda(R,t) = R$.*

Proof. We assume that $events(t) \cup R \subseteq Fvis$. By S3, $Fvis \in AllSet$ and so $Fvis \subseteq \Sigma_{impl}$ by S1(c) (definitions 3.5 and 3.6). Thus, $R \subseteq \Sigma_{impl}$. By proposition A.10, $\lambda(t)$ is defined and so $\lambda(R,t)$ is defined by SFS1 (definition 3.12(3)). Since $Fvis \in AllSet$, we observe that $[[events(t) \cup R]] \subseteq Fvis$ and so, by proposition A.10, $\lambda(t \circ \langle a \rangle)$ is defined for every $a \in [[events(t) \cup R]] - R$. Hence, by SFS7, $\lambda(R,t) = \bigcup_{A \in MinSet} \lambda(R \cap A, t\lceil A)$. Let $A \in MinSet$. By proposition A.4, either $A \cap Fvis = \varnothing$ or $A \subseteq Fvis$. We therefore show that $\lambda(R \cap A, t\lceil A) = R \cap A$ by considering the following two cases in turn.

Case 1: $A \cap Fvis = \varnothing$. In this case, by proposition A.20,

$$\lambda(R \cap A, t\lceil A) = \lambda(\varnothing, \langle\rangle) = \varnothing = R \cap A.$$

Case 2: $A \subseteq Fvis$. In this case, $\lambda(R \cap A, t\lceil A) = R \cap A$ by SFS2.
Hence,

$$\begin{aligned} \lambda(R,t) &= \textstyle\bigcup_{A \in MinSet} R \cap A \\ &= R \cap \textstyle\bigcup_{A \in MinSet} A \\ &= R \cap \Sigma_{impl} && \text{(by S1(c) (definition 3.5))} \\ &= R. \end{aligned}$$

□

Proposition A.22. *Let P be an implementation process such that $\lambda([\![P]\!]_{SF})$ is defined. Let $(t, R \cup (\Sigma - \alpha P)) \in \phi P$ be a refusal-maximal failure such that $R \subseteq \alpha P$. Then:*

1. *$\lambda(t)$ is defined.*

2. *$\lambda(R,t)$ is defined.*

3. *$\lambda(t \circ \langle a \rangle)$ is defined for every $a \in [[events(t) \cup R]] - R$.*

Proof. By SF2, $t \in \tau P$. By SFS1 (definition 3.12(1)), $\lambda(\tau P)$ is defined and so, by TS1, $\lambda(t)$ is defined. Since $R \subseteq \alpha P$ and $\alpha P \in \mathit{AllSet}$, then $R \subseteq \Sigma_{impl}$ by S1(c) (definitions 3.5 and 3.6). Thus, by SFS1 (definition 3.12(3)), $\lambda(R, t)$ is defined. $t \circ \langle a \rangle \in \tau P$ for every $a \in \Sigma - (R \cup (\Sigma - \alpha P)) = \alpha P - R$ by proposition 2.5. Hence, since $\lambda(\tau P)$ is defined and by TS1, $\lambda(t \circ \langle a \rangle)$ is defined for every $a \in \alpha P - R$. By PA1, $\mathit{events}(t) \subseteq \alpha P$ and so, since $R \subseteq \alpha P$, $[[\mathit{events}(t) \cup R]] \subseteq \alpha P$. Hence, $\lambda(t \circ \langle a \rangle)$ is defined for every $a \in [[\mathit{events}(t) \cup R]] - R$. □

Lemma A.23. *Let P be an implementation process such that $\lambda([\![P]\!]_{SF})$ is defined. Let $(t, R \cup (\Sigma - \alpha P)) \in \phi P$ be a refusal-maximal failure such that $R \subseteq \alpha P$. Let $A \in \mathit{AllSet}$ and $\lambda(A) = B$. Then:*

1. *If $A \subseteq R$ then $B \subseteq \lambda(R, t)$.*
2. *$\lambda(R - A, t \setminus A) = \lambda(R, t) - B$.*

Proof. By proposition A.22(2,3), $\lambda(R, t)$ is defined and $\lambda(t \circ \langle a \rangle)$ is defined for every $a \in [[\mathit{events}(t) \cup R]] - R$. By S1(c) (definition 3.6), $A = \bigcup_{i \in I} A_i$, where I is an indexing set into *MinSet*. In the event that $A = \varnothing$ and so $I = \varnothing$, $B = \varnothing$ by S5 and so the proof is immediate for both parts of the lemma. We therefore consider the case that $A \neq \varnothing$.

1. We assume that $A \subseteq R$. By SFS7, $\lambda(R \cap A', t \lceil A')$ is defined for every $A' \in \mathit{MinSet}$ and $\lambda(R, t) = \bigcup_{A' \in \mathit{MinSet}} \lambda(R \cap A', t \lceil A')$. Let $i \in I$. Since $A \subseteq R$ and by SFS4,

$$\lambda(R \cap A_i, t \lceil A_i) = \lambda(A_i, t \lceil A_i) = \lambda(A_i).$$

Hence, and by S5, $B = \bigcup_{i \in I} \lambda(A_i) \subseteq \lambda(R, t)$.

2. By proposition A.22(1), $\lambda(t)$ is defined and so $\lambda(t \setminus A)$ is also defined by proposition A.11. Since $\lambda(R, t)$ is defined then $R \subseteq \Sigma_{impl}$ by SFS1 (definition 3.12(3)) and so $(R - A) \subseteq \Sigma_{impl}$. Thus, $\lambda(R - A, t \setminus A)$ is defined by SFS1 (definition 3.12(3)).

Let J be an indexing set into *MinSet* such that $A' \in \mathit{MinSet}$ only if there exists $j \in J$ such that $A' = A_j$. Then, by SFS7 and S5,

$$\lambda(R, t) - B = \bigcup_{j \in J} \lambda(R \cap A_j, t \lceil A_j) - \bigcup_{i \in I} \lambda(A_i).$$

By SFS7, $\lambda(R \cap A_j, t \lceil A_j)$ is defined and so, by SFS3, $\lambda(R \cap A_j, t \lceil A_j) \subseteq \lambda(A_j)$ for any $j \in J$. Hence, for $i \in I$, $\lambda(R \cap A_i, t \lceil A_i) - \lambda(A_i) = \varnothing$. Moreover, for $j \notin I$ and $i \in I$, $A_j \cap A_i = \varnothing$ by S1(c) (definition 3.5). Thus, by proposition A.6, $\lambda(A_j) \cap \lambda(A_i) = \varnothing$ and so $\lambda(R \cap A_j, t \lceil A_j) \cap \lambda(A_i) = \varnothing$ for $j \notin I$ and $i \in I$. Hence,

$$\lambda(R, t) - B = \bigcup_{j \in J - I} \lambda(R \cap A_j, t \lceil A_j). \tag{A.9}$$

It is immediate that $[[events(t \setminus A) \cup (R - A)]] \subseteq [[events(t) \cup R]]$; moreover, $[[events(t \setminus A) \cup (R - A)]] \cap A = \varnothing$. Hence,

$$[[events(t \setminus A) \cup (R - A)]] - (R - A) \subseteq [[events(t) \cup R]] - R$$

and so $\lambda(t \circ \langle a \rangle)$ is defined for every $a \in [[events(t \setminus A) \cup (R - A)]] - (R - A)$. Since $[[events(t \setminus A) \cup (R - A)]] \cap A = \varnothing$ and by proposition A.11, $\lambda(t \setminus A \circ \langle a \rangle)$ is defined for every $a \in [[events(t \setminus A) \cup (R - A)]] - (R - A)$. We have already shown that $\lambda(R - A, t \setminus A)$ is defined. Hence, by SFS7,

$$\lambda(R - A, t \setminus A) = \bigcup_{j \in J} \lambda((R - A) \cap A_j, (t \setminus A) \lceil A_j).$$

Let $j \in J$. We consider each of two cases in turn.

Case 1: $j \notin I$. In this case, and by S1(c) (definition 3.5), $A_j \cap A = \varnothing$. Thus,

$$\lambda((R - A) \cap A_j, (t \setminus A) \lceil A_j) = \lambda(R \cap A_j, t \lceil A_j).$$

Case 2: $j \in I$. In this case, $A_j \subseteq A$ and so, by proposition A.20,

$$\lambda((R - A) \cap A_j, (t \setminus A) \lceil A_j) = \lambda(\varnothing, \langle \rangle) = \varnothing.$$

Hence, $\lambda(R - A, t \setminus A) = \bigcup_{j \in J - I} \lambda(R \cap A_j, t \lceil A_j)$ and so the proof follows by (A.9). □

Lemma A.24. *Let P and Q be implementation processes such that $\lambda([\![P]\!]_{SF})$ and $\lambda([\![Q]\!]_{SF})$ are defined. Let $(s, S \cup (\Sigma - \alpha P)) \in \phi P$ be a refusal-maximal failure such that $S \subseteq \alpha P$. Let $(u, U \cup (\Sigma - \alpha Q)) \in \phi Q$ be a refusal-maximal failure such that $U \subseteq \alpha Q$. Moreover, let $t \in s \parallel_Y u$, where $Y = \alpha P \cap \alpha Q$. Then $\lambda(S \cup U, t) = \lambda(S, s) \cup \lambda(U, u)$.*

Proof. We first show the following:

$$\lambda(t) \text{ is defined.} \qquad \text{(A.10)}$$

By proposition A.22(1), $\lambda(s)$ and $\lambda(u)$ are defined. By SF2 and PA1, $events(s) \subseteq \alpha P$ and so $[[events(s)]] \subseteq \alpha P$. Similarly, $[[events(u)]] \subseteq \alpha Q$ and so

$$[[events(s)]] \cap [[events(u)]] \subseteq \alpha P \cap \alpha Q = Y.$$

Since $\alpha P, \alpha Q \in AllSet$ by S2, then $Y \in AllSet$ by proposition A.5(1). It follows by proposition A.12 that $\lambda(t)$ is defined and so we have shown (A.10).

We now show the following:

$$\lambda(t \circ \langle a \rangle) \text{ is defined for every } a \in [[events(t) \cup S \cup U]] - (S \cup U) \qquad \text{(A.11)}$$

Let $a \in [[events(t) \cup S \cup U]] - (S \cup U)$. Since $[[events(t) \cup S \cup U]] \in AllSet$, then, by S1(c) (definition 3.6), there exists $A \in MinSet$ such that $a \in A$. Since $t \in s \parallel_Y u$, then $events(t) = events(s) \cup events(u)$. Thus,

$$[[events(s) \cup S]] \cup [[events(u) \cup U]] = [[events(t) \cup S \cup U]]$$

and so we consider each of two cases in turn.

Case 1: $a \in [[events(s) \cup S]]$. We consider each of two sub-cases in turn in order to show that $t \circ \langle a \rangle \lceil A = s \circ \langle a \rangle \lceil A$.

Case 1a: $a \in Y$. By TRP, $t \lceil Y = s \lceil Y$. Since $A \cap Y \neq \varnothing$, then $A \subseteq Y$ by proposition A.4. Thus, $t \lceil A = s \lceil A$ and so $t \circ \langle a \rangle \lceil A = s \circ \langle a \rangle \lceil A$.

Case 1b: $a \notin Y$. In this case, $A \nsubseteq Y$ and so $A \cap Y = \varnothing$ by proposition A.4. Since $a \in [[events(s) \cup S]]$, then $A \subseteq [[events(s) \cup S]]$ by proposition A.4. By SF2 and PA1, $events(s) \subseteq \alpha P$ and we already know that $S \subseteq \alpha P$. Hence, $A \subseteq [[events(s) \cup S]] \subseteq \alpha P$. Also by SF2 and PA1, $events(u) \subseteq \alpha Q$. Recall that $\alpha P \cap \alpha Q = Y$. Thus, since $A \cap Y = \varnothing$, $A \cap \alpha Q = \varnothing$ and so $A \cap events(u) = \varnothing$. This means that $t \lceil A = s \lceil A$ and so $t \circ \langle a \rangle \lceil A = s \circ \langle a \rangle \lceil A$.

Since $a \in [[events(s) \cup S]]$ and $a \notin (S \cup U)$, then $a \in [[events(s) \cup S]] - S$ and so $\lambda(s \circ \langle a \rangle)$ is defined by proposition A.22(3). Hence, $\lambda(s \circ \langle a \rangle \lceil A)$ is defined by TS3. Thus, $\lambda(t \circ \langle a \rangle \lceil A)$ is defined and, since $\lambda(t)$ is defined by (A.10), $\lambda(t \circ \langle a \rangle)$ is defined by TS3.

Case 2: $a \in [[events(u) \cup U]]$. The proof in this case is similar to that of Case 1.

Hence, we have shown (A.11) and so now proceed with the proof proper. By (A.10), $\lambda(t)$ is defined. By proposition A.22(2), $\lambda(S, s)$ and $\lambda(U, u)$ are defined; thus, by SFS1 (definition 3.12(3)), $S, U \subseteq \Sigma_{impl}$. Hence, $S \cup U \subseteq \Sigma_{impl}$ and so $\lambda(S \cup U, t)$ is defined by SFS1 (definition 3.12(3)).

Let J be an indexing set into $MinSet$ such that $A' \in MinSet$ only if there exists $j \in J$ such that $A' = A_j$. By proposition A.22(2,3), $\lambda(S, s)$ and $\lambda(U, u)$ are defined, $\lambda(s \circ \langle a \rangle)$ is defined for every $a \in [[events(s) \cup S]] - S$ and $\lambda(u \circ \langle a \rangle)$ is defined for every $a \in [[events(u) \cup U]] - U$. Thus, by SFS7,

- $\lambda(S, s) = \bigcup_{j \in J} \lambda(S \cap A_j, s \lceil A_j)$ and $\lambda(S \cap A_j, s \lceil A_j)$ is defined for every $j \in J$.
- $\lambda(U, u) = \bigcup_{j \in J} \lambda(U \cap A_j, u \lceil A_j)$ and $\lambda(U \cap A_j, u \lceil A_j)$ is defined for every $j \in J$.

Moreover, since $\lambda(S \cup U, t)$ is defined and by (A.11) and SFS7,

$$\lambda(S \cup U, t) = \bigcup_{j \in J} \lambda((S \cup U) \cap A_j, t \lceil A_j).$$

Thus, for $j \in J$, it suffices to show that

$$\lambda((S \cup U) \cap A_j, t \lceil A_j) = \lambda(S \cap A_j, s \lceil A_j) \cup \lambda(U \cap A_j, u \lceil A_j).$$

In order to do this, and by proposition A.4, we consider the following two cases in turn.

Case 1: $A_j \subseteq Y$. In this case, by TRP, $t\lceil A_j = s\lceil A_j = u\lceil A_j$. We consider each of two sub-cases in turn.

Case 1a: $S \cap A_j \neq \varnothing$ and $U \cap A_j \neq \varnothing$. By SFS6 and due to the fact that $t\lceil A_j = s\lceil A_j = u\lceil A_j$, it suffices to show that $\lambda(t\lceil A_j \circ \langle a\rangle)$ is defined for every $a \in (A_j - (S \cap A_j)) \cup (A_j - (U \cap A_j))$. Wlog, let $a \in A_j - (S \cap A_j)$. Since $S \cap A_j \neq \varnothing$ and by proposition A.4, $A_j \subseteq [[events(s) \cup S]]$. Hence, $A_j - (S \cap A_j) \subseteq [[events(s) \cup S]] - S$ and so $\lambda(s \circ \langle a\rangle)$ is defined. Since $a \in A_j$ and $t\lceil A_j = s\lceil A_j$, $\lambda(s \circ \langle a\rangle\lceil A_j) = \lambda(t\lceil A_j \circ \langle a\rangle)$ is defined by TS3.

Case 1b: Either $S \cap A_j = \varnothing$ or $U \cap A_j = \varnothing$. Wlog, we assume that $S \cap A_j = \varnothing$. Hence, and since $t\lceil A_j = u\lceil A_j$,

$$\lambda((S \cup U) \cap A_j, t\lceil A_j) = \lambda(U \cap A_j, u\lceil A_j).$$

Moreover, by SFS5 and since $t\lceil A_j = s\lceil A_j = u\lceil A_j$,

$$\lambda(S \cap A_j, s\lceil A_j) = \lambda(\varnothing, t\lceil A_j) \subseteq \lambda(U \cap A_j, t\lceil A_j) = \lambda(U \cap A_j, u\lceil A_j)$$

and so the proof follows in this case.

Case 2: $A_j \cap Y = \varnothing$. Since $\alpha P \cap \alpha Q = Y$, either $A_j \cap \alpha P = \varnothing$ or $A_j \cap \alpha Q = \varnothing$. Wlog, we assume that $A_j \cap \alpha P = \varnothing$ and so $A_j \cap (events(s) \cup S) = \varnothing$. Thus, by proposition A.20, $\lambda(S \cap A_j, s\lceil A_j) = \lambda(\varnothing, \langle\rangle) = \varnothing$. Since $events(s) \cap A_j = \varnothing$, then $t\lceil A_j = u\lceil A_j$. As a result, $\lambda((S \cup U) \cap A_j, t\lceil A_j) = \lambda(U \cap A_j, u\lceil A_j)$ and so the proof follows in this case. $\square$

Proposition A.25. *Let Q be an implementation process. If $\lambda(\tau Q)$ is defined then $\lambda([\![Q]\!]_{SF})$ is defined.*

Proof. We assume that $\lambda(\tau Q)$ is defined. Let $(t, R) \in \phi Q$. By SFS1 (definition 3.12 (parts 1 and 3a)), it suffices to show that $\lambda(R \cap \alpha Q, t)$ is defined. By SF2, $t \in \tau Q$ and so $\lambda(t)$ is defined by TS1. Since $\alpha Q \in AllSet$, $R \cap \alpha Q \subseteq \Sigma_{impl}$ by S1(c) (definitions 3.5 and 3.6). Hence, by SFS1 (definition 3.12(3)), $\lambda(R \cap \alpha Q, t)$ is defined. $\square$

Proposition A.26. *Let Q be an implementation process. If $\alpha Q \subseteq Fvis$, then $\lambda([\![Q]\!]_{SF})$ is defined and $\lambda([\![Q]\!]_{SF}) = [\![Q]\!]_{SF}$.*

Proof. We assume that $\alpha Q \subseteq Fvis$. Before proceeding with the proof proper, we first show the following result.

$$\lambda(\alpha Q) = \alpha Q. \tag{A.12}$$

By S2, $\alpha Q \in AllSet$ and so, by S1(c) (definition 3.6), $\alpha Q = \bigcup_{i \in I} A_i$ where I is an indexing set into $MinSet$. By S5, $\lambda(\alpha Q) = \bigcup_{i \in I} \lambda(A_i)$. In the event that $\alpha Q = \varnothing$ and so $I = \varnothing$ the proof is immediate and so we assume that $\alpha Q \neq \varnothing$. Let $i \in I$ and so $A_i \subseteq \alpha Q \subseteq \mathit{Fvis}$. By S1(c) (definition 3.5), $A_i \subseteq \Sigma_{impl}$ and so, by proposition A.9 and definition 3.12(3), $\lambda(A_i, \langle\rangle)$ is defined. Moreover, by SFS2 and SFS4, $A_i = \lambda(A_i, \langle\rangle) = \lambda(A_i)$. Hence, $\lambda(\alpha Q) = \bigcup_{i \in I} A_i = \alpha Q$.

We now proceed with the proof proper. By proposition A.13, $\lambda(\tau Q)$ is defined and so, by proposition A.25, $\lambda([\![Q]\!]_{SF})$ is defined. Thus, by SFS1 (definition 3.12(1)), $\lambda(\phi Q)$ is defined. Also by proposition A.13, $\lambda(\tau Q) = \tau Q$ and so, by SFS1 (definition 3.12(2)), we have to show that $\lambda(\phi Q) = \phi Q$. By SFS1 (definition 3.12(3b)),

$$\begin{aligned}\lambda(\phi Q) = \{(\lambda(t), X) \mid\ & (\exists (t, R) \in \phi Q)\ R \subseteq \alpha Q \wedge \\ & X \subseteq \lambda(R, t) \cup (\Sigma - \lambda(\alpha Q))\}.\end{aligned}$$

Let $(t, R) \in \phi Q$ be such that $R \subseteq \alpha Q \subseteq \mathit{Fvis}$. By SF2 and PA1, $\mathit{events}(t) \subseteq \alpha Q \subseteq \mathit{Fvis}$. Thus, by proposition A.10, $\lambda(t) = t$. Moreover, by lemma A.21, $\lambda(R, t) = R$. Hence, since $\lambda(\alpha Q) = \alpha Q$ by (A.12),

$$\lambda(\phi Q) = \{(t, X) \mid (\exists (t, R) \in \phi Q)\ R \subseteq \alpha Q \wedge X \subseteq R \cup (\Sigma - \alpha Q)\}.$$

Thus, $\lambda(\phi Q) = \phi Q$ by PA2 and SF3. □

Proposition A.27. *Let P, Q be implementation processes and $A, Y \in \mathit{AllSet}$.*

1. $\alpha(P \setminus A) = (\alpha P) - A$.
2. $\alpha(P \parallel_Y Q) = \alpha P \cup \alpha Q$.

Proof. 1.

$$\begin{aligned}\alpha(P \setminus A) &= [[\beta(P \setminus A)]] && \text{(by S2)}\\ &= [[\beta(P) - A]] && \text{(by figure 2.5)}\\ &= [[\beta(P)]] - A && \text{(since } A \in \mathit{AllSet})\\ &= (\alpha P) - A && \text{(by S2)}\end{aligned}$$

2.

$$\begin{aligned}\alpha(P \parallel_Y Q) &= [[\beta(P \parallel_Y Q)]] && \text{(by S2)}\\ &= [[\beta(P) \cup \beta(Q)]] && \text{(by figure 2.5)}\\ &= [[\beta(P)]] \cup [[\beta(Q)]] && \\ &= \alpha P \cup \alpha Q && \text{(by S2)}\end{aligned}$$

□

Proposition A.28. *Let Q be an implementation process. If $\lambda([\![Q]\!]_{SF})$ is defined, $A \in \mathit{AllSet}$ and $\lambda(\setminus A) = \setminus B$, then:*

1. $\lambda([\![Q \setminus A]\!]_{SF})$ *is defined.*

2. $\lambda([\![Q \setminus A]\!]_{SF}) \subseteq \lambda([\![Q]\!]_{SF}) \setminus B$.

Proof. We assume that $\lambda([\![Q]\!]_{SF})$ is defined, $A \in \mathit{AllSet}$ and $\lambda(\setminus A) = \setminus B$. By S6, $B = \lambda(A)$. By SFS1 (definition 3.12(1)), $\lambda(\tau Q)$ is defined and so, by proposition A.14(1), $\lambda(\tau(Q \setminus A))$ is defined. Hence, $\lambda([\![Q \setminus A]\!]_{SF})$ is defined by proposition A.25. By proposition A.14(2), $\lambda(\tau(Q \setminus A)) = \lambda(\tau Q) \setminus B$ and so, by SFS1 (definition 3.12(2)), it suffices to show that $\lambda(\phi(Q \setminus A)) \subseteq \lambda(\phi Q) \setminus B$.

Let $(t, X) \in \phi(Q \setminus A)$ be refusal-maximal. By SF3, $(t, X \cap \alpha(Q \setminus A))) \in \phi(Q \setminus A)$. Since $\lambda([\![Q \setminus A]\!]_{SF})$ is defined, then $\lambda(\phi(Q \setminus A))$ is defined by SFS1 (definition 3.12(1)) and so, by SFS1 (definition 3.12(3a)), $\lambda(X \cap \alpha(Q \setminus A), t)$ is defined. By SFS1 (definition 3.12(1)), $\lambda(\phi Q)$ is defined and so, by SFS1 (definition 3.12(3b)), $\lambda(\phi Q)$ and $\lambda(\phi Q) \setminus B$ are subset-closed sets of failures. Hence, by SFS1 (definition 3.12(3b)) and SFS5, it suffices to show that

$$(\lambda(t), \lambda(X \cap \alpha(Q \setminus A), t) \cup (\Sigma - \lambda(\alpha(Q \setminus A)))) \in \lambda(\phi Q) \setminus B.$$

By proposition A.27(1), $\alpha(Q \setminus A) = \alpha Q - A$ and so $\alpha(Q \setminus A) \subseteq \alpha Q$. Hence, $(\Sigma - \alpha Q) \subseteq (\Sigma - \alpha(Q \setminus A))$ and so, by PA2, $(\Sigma - \alpha Q) \subseteq X$. Thus, let R be such that $R \subseteq \alpha Q$ and $X = R \cup (\Sigma - \alpha Q)$. Then,

$$X \cap \alpha(Q \setminus A) = (X \cap \alpha Q) \cap \alpha(Q \setminus A) = R \cap \alpha(Q \setminus A) = R - A.$$

By proposition A.5(2), $\lambda(\alpha(Q \setminus A)) = \lambda(\alpha Q - A) = \lambda(\alpha Q) - B$. Hence, and since $B - \lambda(\alpha Q) \subseteq (\Sigma - \lambda(\alpha Q))$,

$$\Sigma - \lambda(\alpha(Q \setminus A)) = (\Sigma - \lambda(\alpha Q)) \cup (B \cap \lambda(\alpha Q)) = (\Sigma - \lambda(\alpha Q)) \cup B.$$

As a result, we have to show that

$$(\lambda(t), \lambda(R - A, t) \cup B \cup (\Sigma - \lambda(\alpha Q))) \in \lambda(\phi Q) \setminus B. \qquad (**)$$

Since $(t, R \cup (\Sigma - \alpha Q)) \in \phi(Q \setminus A)$ is refusal-maximal,

- $A \subseteq R \cup (\Sigma - \alpha Q)$.
- $(w, R \cup (\Sigma - \alpha Q)) \in \phi Q$ is refusal-maximal, where $t = w \setminus A$. Moreover, by SF3, $(w, R) \in \phi Q$.

Since $\lambda([\![Q]\!]_{SF})$ is defined, $\lambda(\tau Q)$ and $\lambda(\phi Q)$ are defined by SFS1 (definition 3.12(1)). By SF2, $w \in \tau Q$ and so, by TS1, $\lambda(w)$ is defined. Moreover, since $\lambda(\phi Q)$ is defined and $R \subseteq \alpha Q$, and by SFS1 (definition 3.12(3b)),

$$(\lambda(w), \lambda(R, w) \cup (\Sigma - \lambda(\alpha Q))) \in \lambda(\phi Q).$$

Since $A \subseteq R \cup (\Sigma - \alpha Q)$, then $A \cap \alpha Q \subseteq R$. By proposition A.5, $A \cap \alpha Q \in$ *AllSet* and $\lambda(A \cap \alpha Q) = B \cap \lambda(\alpha Q)$. Thus, by lemma A.23(1), $B \cap \lambda(\alpha Q) \subseteq \lambda(R, w)$. Thus, since $B - \lambda(\alpha Q) \subseteq (\Sigma - \lambda(\alpha Q))$, $B \subseteq \lambda(R, w) \cup (\Sigma - \lambda(\alpha Q))$. From this and proposition A.11,

$$(\lambda(t), \lambda(R, w) \cup (\Sigma - \lambda(\alpha Q))) \in \lambda(\phi Q) \setminus B$$

and so

$$(\lambda(t), (\lambda(R, w) - B) \cup B \cup (\Sigma - \lambda(\alpha Q))) \in \lambda(\phi Q) \setminus B.$$

Thus, by lemma A.23(2),

$$(\lambda(t), \lambda(R - A, t) \cup B \cup (\Sigma - \lambda(\alpha Q))) \in \lambda(\phi Q) \setminus B.$$

and the proof follows by $(**)$. □

Lemma A.29. *Let t be a trace and $R \subseteq \Sigma$ such that $\lambda(R, t)$ is defined and $events(t) \cup R \subseteq A \in AllSet$. If, for every $a \in [[events(t) \cup R]] - R$, $\lambda(t \circ \langle a \rangle)$ is defined then $\lambda(R, t) \subseteq \lambda(A)$.*

Proof. By S1(c) (definition 3.6), $A = \bigcup_{i \in I} A_i$, where I is an indexing set into *MinSet*. In the event that $A = \varnothing$, $\lambda(R, t) = \lambda(\varnothing, \langle\rangle) = \varnothing$ by proposition A.20 and so we consider the case that $A \neq \varnothing$. We assume that $\lambda(t \circ \langle a \rangle)$ is defined for every $a \in [[events(t) \cup R]] - R$. Let J be an indexing set into *MinSet* such that $A' \in$ *MinSet* only if there exists $j \in J$ such that $A' = A_j$. By SFS7, $\lambda(R \cap A_j, t \lceil A_j)$ is defined for every $j \in J$ and

$$\lambda(R, t) = \bigcup_{j \in J} \lambda(R \cap A_j, t \lceil A_j).$$

Let $j \in J$. It suffices to show that $\lambda(R \cap A_j, t \lceil A_j) \subseteq \lambda(A)$ and we consider each of two cases in turn.

Case 1: $j \notin I$. By S1(c) (definition 3.5), $A_j \cap A = \varnothing$. As a result, $\lambda(R \cap A_j, t \lceil A_j) = \lambda(\varnothing, \langle\rangle) = \varnothing$ by proposition A.20.

Case 2: $j \in I$. By SFS3 and S5,

$$\lambda(R \cap A_j, t \lceil A_j) \subseteq \lambda(A_j) \subseteq \lambda(A).$$

□

Proposition A.30. *Let P, Q be implementation processes. If $\lambda([\![P]\!]_{SF})$, $\lambda([\![Q]\!]_{SF})$ are defined and $Y = \alpha P \cap \alpha Q$, where $\lambda(\|_Y) = \|_Z$, then:*

1. *$\lambda([\![P \|_Y Q]\!]_{SF})$ is defined.*

2. $\lambda([\![P \parallel_Y Q]\!]_{SF}) \subseteq \lambda([\![P]\!]_{SF}) \parallel_Z \lambda([\![Q]\!]_{SF})$.

Proof. We assume that $\lambda([\![P]\!]_{SF})$, $\lambda([\![Q]\!]_{SF})$ are defined and $Y = \alpha P \cap \alpha Q$, where $\lambda(\parallel_Y) = \parallel_Z$. That $Y \in$ *AllSet* follows by S2 and proposition A.5(1). Thus, by S7, $Z = \lambda(Y)$. By SFS1 (definition 3.12(1)), $\lambda(\tau P)$ and $\lambda(\tau Q)$ are defined and so, by proposition A.15(1), $\lambda(\tau(P \parallel_Y Q))$ is defined. Hence, $\lambda([\![P \parallel_Y Q]\!]_{SF})$ is defined by proposition A.25.

By proposition A.15(2), $\lambda(\tau(P \parallel_Y Q)) \subseteq \lambda(\tau P) \parallel_Z \lambda(\tau Q)$. Thus, by SFS1 (definition 3.12(2)), we show $\lambda(\phi(P \parallel_Y Q)) \subseteq \lambda(\phi P) \parallel_Z \lambda(\phi Q)$. Let $(t, X) \in \phi(P \parallel_Y Q)$ be refusal-maximal. By proposition A.27(2), we have $\alpha(P \parallel_Y Q) = \alpha P \cup \alpha Q$ and so $(\Sigma - (\alpha P \cup \alpha Q)) \subseteq X$ by PA2. Thus, let R be such that $R \subseteq \alpha P \cup \alpha Q$ and $X = R \cup (\Sigma - (\alpha P \cup \alpha Q))$. By SF3, $(t, R) \in \phi(P \parallel_Y Q)$. Since $\lambda([\![P \parallel_Y Q]\!]_{SF})$ is defined, then $\lambda(\phi(P \parallel_Y Q))$ is defined by SFS1 (definition 3.12(1)) and so, by SFS1 (definition 3.12(3a)), $\lambda(R, t)$ is defined. By SFS1 (definition 3.12(1)), $\lambda(\phi P)$ and $\lambda(\phi Q)$ are defined. Thus, by SFS1 (definition 3.12(3b)), $\lambda(\phi P)$ and $\lambda(\phi Q)$ are subset-closed sets of failures and so $\lambda(\phi P) \parallel_Z \lambda(\phi Q)$ is subset-closed by proposition 2.18. Hence, by SFS1 (definition 3.12(3b)) and SFS5, it suffices to show that

$$(\lambda(t), \lambda(R, t) \cup (\Sigma - \lambda(\alpha P \cup \alpha Q))) \in \lambda(\phi P) \parallel_Z \lambda(\phi Q). \qquad (**)$$

By theorem 2.20, there exist $(s, S) \in \phi P$ and $(u, U) \in \phi Q$ such that:

- $S \subseteq \alpha P$.
- $U \subseteq \alpha Q$.
- $R \cup (\Sigma - (\alpha P \cup \alpha Q)) = S \cup U \cup (\Sigma - (\alpha P \cup \alpha Q))$ and so $R = S \cup U$, since $R, (S \cup U) \subseteq \alpha P \cup \alpha Q$.
- $t \in s \parallel_Y u$.

We assume that S and U are as large as possible such that the above conditions hold. By PA2, $(s, S \cup (\Sigma - \alpha P)) \in \phi P$ and $(u, U \cup (\Sigma - \alpha Q)) \in \phi Q$. Moreover, $(s, S \cup (\Sigma - \alpha P)) \in \phi P$ and $(u, U \cup (\Sigma - \alpha Q)) \in \phi Q$ are refusal-maximal, since otherwise $(t, R \cup (\Sigma - (\alpha P \cup \alpha Q))) \in \phi(P \parallel_Y Q)$ would not be refusal-maximal. Since $\lambda([\![P]\!]_{SF})$ and $\lambda([\![Q]\!]_{SF})$ are defined, $\lambda(\phi P)$ and $\lambda(\phi Q)$ are defined by SFS1 (definition 3.12(1)). Thus, by SFS1 (definition 3.12(3b)),

- $(\lambda(s), \lambda(S, s) \cup (\Sigma - \lambda(\alpha P)) \in \lambda(\phi P)$.
- $(\lambda(u), \lambda(U, u) \cup (\Sigma - \lambda(\alpha Q)) \in \lambda(\phi Q)$.

By proposition A.22(2,3), $\lambda(S, s)$ is defined and $\lambda(s \circ \langle a \rangle)$ is defined for every $a \in [[events(s) \cup S]] - S$. We know that $S \subseteq \alpha P$ and, by SF2 and PA1, $events(s) \subseteq \alpha P$. Thus, by lemma A.29, $\lambda(S, s) \subseteq \lambda(\alpha P)$. Similarly, $\lambda(U, u) \subseteq \lambda(\alpha Q)$. By proposition A.22(1), $\lambda(s)$ and $\lambda(u)$ are defined. Since $events(s) \subseteq \alpha P$ and, by SF2 and PA1, $events(u) \subseteq \alpha Q$, we observe that $[[events(s)]] \cap [[events(u)]] \subseteq \alpha P \cap \alpha Q = Y$. Thus, by proposition A.12, $\lambda(t) \in \lambda(s) \parallel_Z \lambda(u)$. Hence, by theorem 3.22,

$$(\lambda(t), \lambda(S, s) \cup \lambda(U, u) \cup (\Sigma - (\lambda(\alpha P) \cup \lambda(\alpha Q)))) \in \lambda(\phi P) \parallel_Z \lambda(\phi Q).$$

We observe that, by proposition A.5(2), $\lambda(\alpha P \cup \alpha Q) = \lambda(\alpha P) \cup \lambda(\alpha Q)$. The proof then follows by (**), the fact that $R = S \cup U$ and lemma A.24. □

Proof of theorem 3.33

Proof. The proof is similar to that of theorem 3.1, using proposition A.26 in place of RAH1, proposition A.28 in place of RAH2 and proposition A.30 in place of RAH3. □

Proposition A.31. *The following hold:*

1. $\Sigma_{impl} \in AllSet$.
2. $\Sigma_{spec} = \bigcup_{A \in MinSet} \lambda(A)$.
3. *For every* $A \in AllSet$, $A \subseteq \Sigma_{impl}$ *and* $\lambda(A) \subseteq \Sigma_{spec}$.

Proof. 1. The proof is immediate by S1(c) (definitions 3.5 and 3.6).

2.

$$\begin{aligned}
& \textstyle\bigcup_{A \in MinSet} \lambda(A) \\
= \; & \textstyle\bigcup_{A \in MinSet} \bigcup \{events(\lambda(t)) \mid t \in BTrace \ \wedge \ events(t) \subseteq A\} && \text{(by S4)} \\
= \; & \textstyle\bigcup \{events(\lambda(t)) \mid t \in BTrace \ \wedge \ ((\exists A \in MinSet) \ events(t) \subseteq A)\} \\
= \; & \textstyle\bigcup \{events(\lambda(t)) \mid t \in BTrace \ \wedge \ events(t) \subseteq \bigcup_{A \in MinSet} A\} \\
& && \text{(by (S1)(c) (definition 3.5(1)))} \\
= \; & \textstyle\bigcup \{events(\lambda(t)) \mid t \in BTrace \ \wedge \ events(t) \subseteq \Sigma_{impl}\} \\
& && \text{(by (S1)(c) (definition 3.5))} \\
= \; & \textstyle\bigcup \{events(\lambda(t)) \mid t \in BTrace\} && \text{(by S1(b) (definition 3.4(1)))} \\
= \; & \Sigma_{spec} && \text{(by S1(b) (definition 3.4(2)))}
\end{aligned}$$

3. Let $A \in AllSet$. By S1(c) (definition 3.5), $\Sigma_{impl} = \bigcup_{A' \in MinSet} A'$ and so $A \subseteq \Sigma_{impl}$ by S1(c) (definition 3.6). Moreover, $\lambda(A) \subseteq \lambda(\Sigma_{impl})$ by S5. By part 2 of the proposition and S5,

$$\Sigma_{spec} = \bigcup_{A' \in MinSet} \lambda(A') = \lambda(\bigcup_{A' \in MinSet} A') = \lambda(\Sigma_{impl})$$

and so $\lambda(A) \subseteq \Sigma_{spec}$. □

Proof of theorem 3.34

Proof. By proposition A.25, $\lambda([\![P]\!]_{SF})$ and $\lambda([\![Q]\!]_{SF})$ are defined. Moreover, by SFS1 (definition 3.12(1)), $\lambda(\phi P)$ and $\lambda(\phi Q)$ are defined. Let $(t, R) \in \phi P = \phi Q$ be refusal-maximal and let $R_P = R \cap \alpha P$ and $R_Q = R \cap \alpha Q$. By SF3, $(t, R_P) \in \phi P$ and $(t, R_Q) \in \phi Q$. By PA2, $\Sigma - \alpha P \subseteq R$ and $\Sigma - \alpha Q \subseteq R$. Thus,

$$R = R_P \cup (\Sigma - \alpha P) = R_Q \cup (\Sigma - \alpha Q).$$

Note also that, by SFS1 (definition 3.12(3a)), $\lambda(R_P, t)$ and $\lambda(R_Q, t)$ are defined. Thus, according to SFS1 (definition 3.12(3b)) and SFS5, it suffices to show that

$$\lambda(R_P, t) \cup (\Sigma - \lambda(\alpha P)) = \lambda(R_Q, t) \cup (\Sigma - \lambda(\alpha Q)).$$

By S1(b) (definition 3.4), $\Sigma_{spec} \subseteq \Sigma$. Moreover, by proposition A.31(3), $\lambda(\alpha P), \lambda(\alpha Q) \subseteq \Sigma_{spec}$. Thus, $\Sigma - \lambda(\alpha Q) = (\Sigma - \Sigma_{spec}) \cup (\Sigma_{spec} - \lambda(\alpha Q))$ and $\Sigma - \lambda(\alpha P) = (\Sigma - \Sigma_{spec}) \cup (\Sigma_{spec} - \lambda(\alpha P))$. Thus, it it suffices to show that

$$\lambda(R_P, t) \cup (\Sigma_{spec} - \lambda(\alpha P)) = \lambda(R_Q, t) \cup (\Sigma_{spec} - \lambda(\alpha Q)).$$

In fact, we show that $\lambda(R_P, t) \cup (\Sigma_{spec} - \lambda(\alpha P)) \subseteq \lambda(R_Q, t) \cup (\Sigma_{spec} - \lambda(\alpha Q))$ and the proof in the other direction is similar.

Let I, K be indexing sets into *MinSet* such that $\alpha P = \bigcup_{i \in I} A_i$ and $\alpha Q = \bigcup_{k \in K} A_k$. Let J be an indexing set into *MinSet* such that $A \in MinSet$ only if there exists $j \in J$ such that $A = A_j$. By proposition A.22(3), $\lambda(t \circ \langle a \rangle)$ is defined for every $a \in [[events(t) \cup R_Q]] - R_Q$. Thus, by SFS7,

$$\lambda(R_Q, t) = \bigcup_{j \in J} \lambda(R_Q \cap A_j, t \lceil A_j) \tag{A.13}$$

and $\lambda(R_Q \cap A_j, t \lceil A_j)$ is defined for every $j \in J$.

We first show that $\lambda(R_P, t) \subseteq \lambda(R_Q, t) \cup (\Sigma_{spec} - \lambda(\alpha Q))$. By proposition A.22(3), $\lambda(t \circ \langle a \rangle)$ is defined for every $a \in [[events(t) \cup R_P]] - R_P$. Thus, by SFS7,

$$\lambda(R_P, t) = \bigcup_{j \in J} \lambda(R_P \cap A_j, t \lceil A_j)$$

and $\lambda(R_P \cap A_j, t \lceil A_j)$ is defined for every $j \in J$. It therefore suffices to show that

$$\lambda(R_P \cap A_j, t \lceil A_j) \subseteq \lambda(R_Q, t) \cup (\Sigma_{spec} - \lambda(\alpha Q))$$

for $j \in J$. Let $j \in J$. We consider each of three cases in turn.

Case 1: $j \in I \cap K$. In this case, $A_j \subseteq \alpha P$ and $A_j \subseteq \alpha Q$. Thus, $R_P \cap A_j = R_Q \cap A_j$ and so $\lambda(R_P \cap A_j, t\lceil A_j) \subseteq \lambda(R_Q, t)$ by (A.13).

Case 2: $j \notin K$. In this case, by S1(c) (definition 3.5), $A_j \cap \alpha Q = \varnothing$. Thus, $\lambda(A_j) \cap \lambda(\alpha Q) = \varnothing$ by proposition A.6. By proposition A.31(3), $\lambda(A_j) \subseteq \Sigma_{spec}$ and so $\lambda(A_j) \subseteq (\Sigma_{spec} - \lambda(\alpha Q))$. Since $\lambda(R_P \cap A_j, t\lceil A_j)$ is defined and by SFS1 (definition 3.12(3)), $\lambda(t\lceil A_j)$ is defined. By S1(c) (definition 3.5), $A_j \subseteq \Sigma_{impl}$ and so $\lambda(A_j, t\lceil A_j)$ is defined also by SFS1 (definition 3.12(3)). Thus, by SFS4 and SFS5,

$$\lambda(R_P \cap A_j, t\lceil A_j) \subseteq \lambda(A_j, t\lceil A_j) = \lambda(A_j)$$

and so $\lambda(R_P \cap A_j, t\lceil A_j) \subseteq (\Sigma_{spec} - \lambda(\alpha Q))$.

Case 3: $j \notin I$. In this case, by S1(c) (definition 3.5), $A_j \cap \alpha P = \varnothing$. By SF2 and PA1, *events*$(t) \subseteq \alpha P$. Thus, $\lambda(R_P \cap A_j, t\lceil A_j) = \lambda(\varnothing, \langle\rangle) = \varnothing$ by proposition A.20.

We now show that $(\Sigma_{spec} - \lambda(\alpha P)) \subseteq \lambda(R_Q, t) \cup (\Sigma_{spec} - \lambda(\alpha Q))$. By proposition A.31(2), $\Sigma_{spec} = \bigcup_{A \in MinSet} \lambda(A)$ and so, by S1(c) (definition 3.5) and S5, $\Sigma_{spec} = \lambda(\Sigma_{impl})$. Thus, by proposition A.5(2) and S5,

$$\Sigma_{spec} - \lambda(\alpha P) = \lambda(\Sigma_{impl} - \alpha P) = \lambda(\bigcup_{j \in J-I} A_i) = \bigcup_{j \in J-I} \lambda(A_i).$$

It therefore suffices to show that $\lambda(A_j) \subseteq \lambda(R_Q, t) \cup (\Sigma_{spec} - \lambda(\alpha Q))$ for $j \in J - I$. Let $j \in J - I$. We consider each of two cases in turn.

Case 1: $j \notin K$. In this case, by S1(c) (definition 3.5), $A_j \cap \alpha Q = \varnothing$ and so $\lambda(A_j) \cap \lambda(\alpha Q) = \varnothing$ by proposition A.6. Thus, since $\lambda(A_j) \subseteq \Sigma_{spec}$ by proposition A.31(3), $\lambda(A_j) \subseteq (\Sigma_{spec} - \lambda(\alpha Q))$.

Case 2: $j \in K$. In this case, $A_j \subseteq \alpha Q$. Moreover, since $j \notin I$, $A_j \cap \alpha P = \varnothing$. Since $\Sigma - \alpha P \subseteq R$, then $A_j \subseteq R$ and so $A_j \subseteq R_Q$. Thus, $\lambda(R_Q \cap A_j, t\lceil A_j) = \lambda(A_j, t\lceil A_j)$ and the proof in this case follows by SFS4 and (A.13). □

A.5 Proofs from section 3.6

Proof of theorem 3.35

Proof. By SEQ, there exists a deterministic implementation process Q such that $\tau Q = Pref(\{\ldots, t_i, \ldots\})$. Since Q is deterministic, then $\delta Q = \varnothing$. Thus, by definitions 3.3 and 3.13(3), $\lambda(\delta Q)$ is defined. By PREF-CLOS and definition 3.3, $\lambda(\tau Q)$ is defined and so $\lambda([\![Q]\!]_{SF})$ is defined by proposition A.16. Hence, by definition 3.12(1), $\lambda(\phi Q)$ is defined and so $\lambda(\phi_\perp Q)$ is defined by definition 3.13(4). Thus, $\lambda([\![Q]\!]_{FD})$ is defined by definition 3.13(1).

We now show the following:

$$\{\lambda(w) \mid w \in \mathit{Pref}(\{\ldots, t_i, \ldots\})\} = \{t \mid (t, \varnothing) \in \lambda(\phi_\perp Q)\}. \tag{A.14}$$

Since $\lambda(\phi Q)$ is defined and by definition 3.12(3b),

$$\{t \mid (t, \varnothing) \in \lambda(\phi Q)\} = \{\lambda(w) \mid (w, \varnothing) \in \phi Q\}.$$

Since Q is deterministic and by definition 2.4(2), $\tau Q = \{w \mid (w, \varnothing) \in \phi Q\}$. Thus, by definition 3.3, $\lambda(\tau Q) = \{t \mid (t, \varnothing) \in \lambda(\phi Q)\}$. Moreover, also by definition 3.3, we have that $\lambda(\tau Q) = \{\lambda(w) \mid w \in \mathit{Pref}(\{\ldots, t_i, \ldots\})\}$. The proof of (A.14) follows by definition 3.13(4) and the fact that $\lambda(\delta Q) = \varnothing$ by definition 3.13(3) and since $\delta Q = \mathit{min}\delta Q = \varnothing$.

By FDI, $\lambda([\![Q \backslash A]\!]_{FD})$ is defined and so $\lambda(\delta(Q \backslash A))$ is defined by definition 3.13(1). We recall that $\delta Q = \varnothing$ and so $\lambda(\delta Q) = \varnothing$ by definition 3.13(3). Moreover, $\delta(Q \setminus A) \neq \varnothing$ and so $\lambda(\delta(Q \setminus A)) \neq \varnothing$ also by definition 3.13(3). Let $\mathcal{F}$ be a set of failures and $\mathcal{D}$ a set of divergences such that $\lambda([\![Q]\!]_{FD}) \backslash B = (\mathcal{F}, \mathcal{D})$, where $\lambda(\backslash A) = \backslash B$. Thus, by definition 3.13(2) and FDI,

$$(\lambda(\phi_\perp(Q \setminus A)), \lambda(\delta(Q \setminus A))) = (\lambda(\phi_\perp Q), \lambda(\delta Q)) \setminus B = (\mathcal{F}, \mathcal{D}).$$

Hence, $\lambda(\delta(Q \setminus A)) = \mathcal{D}$ and so $\mathcal{D} \neq \varnothing$. Since $\lambda(\delta Q) = \varnothing$, then, by (A.14), TR-MONO and the detail in chapter 2.4.3, $\ldots, \lambda(t_i), \ldots$ must be an ω-sequence. □

Results used in the proof of theorem 3.36

In all of the proofs in the remainder of this section, we assume that the conditions from figures 3.4, 3.6 and 3.8 all hold.

Lemma A.32. *Let t, u be traces. If $\lambda(t)$ is defined, then $\lambda(u)$ is defined for all $u \leq t$.*

Proof. The proof proceeds by induction on the length of t using proposition A.7 and TS3. □

Proposition A.33. *Let v, u be traces. If $u \leq v$ and $\lambda(u)$ and $\lambda(v)$ are defined, then $\lambda(u) \leq \lambda(v)$.*

Proof. The proof proceeds by induction on $n = |v| - |u|$. In the base case, the proof is immediate. Let $v = u \circ w \circ \langle a \rangle$. Since $\lambda(v)$ is defined, by lemma A.32 we have that $\lambda(u \circ w)$ is defined. Thus, by the inductive hypothesis, $\lambda(u) \leq \lambda(u \circ w)$. Moreover, by proposition A.7, there exists $A \in \mathit{MinSet}$ such that $a \in A$. Thus, by TS4, $\lambda(u \circ w \circ \langle a \rangle) = \lambda(u \circ w) \circ r$ (for some trace r) and so the proof follows. □

Proposition A.34. *Let Q be an implementation process. $\lambda(\tau Q)$ is defined if and only if $\lambda(\llbracket Q \rrbracket_{FD})$ is defined.*

Proof. ($\Longrightarrow$) We assume that $\lambda(\tau Q)$ is defined. By proposition A.25, $\lambda(\llbracket Q \rrbracket_{SF})$ is defined and so $\lambda(\phi Q)$ is defined by SFS1 (definition 3.12(1)). Moreover, $\lambda(\tau Q \cap \delta Q)$ is defined by TS1. Thus, by FDS1 (definition 3.13(3)), $\lambda(\delta Q)$ is defined. Hence, by FDS1 (definition 3.13(1,4)), $\lambda(\phi_{\perp} Q)$ and then $\lambda(\llbracket Q \rrbracket_{FD})$ are defined.

($\Longleftarrow$) We assume that $\lambda(\llbracket Q \rrbracket_{FD})$ is defined. By FDS1 (definition 3.13), $\lambda(\phi_{\perp} Q)$ is defined and so $\lambda(\phi Q)$ is defined; moreover, $\lambda(\delta Q)$ is defined. By DR3, we have that $\tau Q = \{t \mid (t, \varnothing) \in \phi Q\} \cup (\tau Q \cap \delta Q)$. By SFS1 (definition 3.12(3)), $\lambda(w)$ is defined for every $w \in \{t \mid (t, \varnothing) \in \phi Q\}$ and, by FDS1 (definition 3.13(3)), $\lambda(\tau Q \cap \delta Q)$ is defined. Thus, by TS1, $\lambda(\tau Q)$ is defined. □

Lemma A.35. *Let Q be an implementation process. If $\alpha Q \subseteq Fvis$, then $\lambda(\llbracket Q \rrbracket_{FD})$ is defined and $\lambda(\llbracket Q \rrbracket_{FD}) = \llbracket Q \rrbracket_{FD}$.*

Proof. By proposition A.13, $\lambda(\tau Q)$ is defined and so $\lambda(\llbracket Q \rrbracket_{FD})$ is defined by proposition A.34. By FDS1 (definition 3.13(1,2)), $\lambda(\phi_{\perp} Q)$ and $\lambda(\delta Q)$ are defined and it suffices to show that $\lambda(\phi_{\perp} Q) = \phi_{\perp} Q$ and $\lambda(\delta Q) = \delta Q$. Since $min\delta Q \subseteq \tau Q$ by MD, $events(t) \subseteq \alpha Q \subseteq Fvis$ for every $t \in min\delta Q$ by PA1. Thus, by FDS1 (definition 3.13(3)), proposition A.10 and the extension closure of sets of divergent traces by FD4, $\lambda(\delta Q) = \delta Q$. By proposition A.26 and SFS1 (definition 3.12(2)), $\lambda(\phi Q) = \phi Q$. Hence, by FDS1 (definition 3.13(4)) and since $\lambda(\delta Q) = \delta Q$, $\lambda(\phi_{\perp} Q) = \phi Q \cup \{(t, R) \mid t \in \delta Q \ \wedge \ R \subseteq \Sigma\}$. Thus, by DR2, $\lambda(\phi_{\perp} Q) = \phi_{\perp} Q$. □

Lemma A.36. *Let $\ldots, t_i, \ldots$ be an ω-sequence such that $\lambda(t_i)$ is defined for each t_i. Then $\ldots, \lambda(t_i), \ldots$ is also an ω-sequence.*

Proof. Let $w \in \Sigma^{\omega}$ be the least upper bound of the sequence $\ldots, t_i, \ldots$. Since $\lambda(t_i)$ is defined for each t_i and by S1(b) (definition 3.4(1)), $events(t_i) \subseteq \Sigma_{impl}$ for each t_i. Thus, $events(w) \subseteq \Sigma_{impl}$. Hence, by S1(c) (definition 3.5), there exists $A \in MinSet$ such that $w \lceil A \in \Sigma^{\omega}$. It follows that $\ldots, t_i \lceil A, \ldots$ is an ω-sequence and so $\ldots, \lambda(t_i \lceil A), \ldots$ is also an ω-sequence by FDS2. By induction on the length of traces using proposition A.9 and TS4, $\lambda(t_i) \in |||_{A' \in MinSet} \lambda(t_i \lceil A')$ for each t_i. Thus, the length of the $\lambda(t_i)$ increases unboundedly. It follows by proposition A.33 that $\ldots, \lambda(t_i), \ldots$ is an ω-sequence. □

Lemma A.37. *Let Q be an implementation process and P a process. Let $A \in AllSet$, where $\lambda(\backslash A) = \backslash B$. If $\lambda(\llbracket Q \rrbracket_{FD})$ is defined and $\lambda(\llbracket Q \rrbracket_{FD}) \subseteq \llbracket P \rrbracket_{FD}$, then:*

1. $\lambda([\![Q \setminus A]\!]_{FD})$ *is defined.*

2. $\lambda([\![Q \setminus A]\!]_{FD}) \subseteq [\![P \setminus B]\!]_{FD}$.

Proof. We assume that $\lambda([\![Q]\!]_{FD})$ is defined and P is a process such that $\lambda([\![Q]\!]_{FD}) \subseteq [\![P]\!]_{FD}$. By proposition A.34, $\lambda(\tau Q)$ is defined and so, by proposition A.14, $\lambda(\tau(Q \setminus A))$ is defined. Hence, $\lambda([\![Q \setminus A]\!]_{FD})$ is defined also by proposition A.34. By FDs1 (definition 3.13(2)), $\lambda(\phi_\perp Q) \subseteq \phi_\perp P$ and $\lambda(\delta Q) \subseteq \delta P$. Also by FDs1 (definition 3.13), $\lambda(\phi_\perp Q)$ is defined and so $\lambda(\phi Q)$ is defined; moreover, $\lambda(\delta Q)$ is defined.

By FDs1 (definition 3.13(2)), we show that $\lambda(\delta(Q \setminus A)) \subseteq \delta(P \setminus B)$ and $\lambda(\phi_\perp(Q \setminus A)) \subseteq \phi_\perp(P \setminus B)$. We first show that $\lambda(\delta(Q \setminus A)) \subseteq \delta(P \setminus B)$. Let $t \in min\delta(Q \setminus A)$. By FDs1 (definition 3.13(3)) and FD4, it suffices to show that $\lambda(t) \in \delta(P \setminus B)$. We note that $B = \lambda(A)$ by S6 and then consider each of two cases in turn according to the semantics of the hiding operator in the failures divergences model.

Case 1: There exists $s \in \delta Q$ such that $s \in min\delta Q$, where $t = s \setminus A$. (If $s \notin min\delta Q$ then we have $u < s$ such that $u \in min\delta Q$ and either $u \setminus A = s \setminus A$ and we take $t = u \setminus A$ or $u \setminus A < s \setminus A$ and so $t \notin min\delta(Q \setminus A)$.) By FDs1 (definition 3.13(3)) and since $\lambda(\delta Q) \subseteq \delta P$, $\lambda(s) \in \delta P$. Hence, $\lambda(s) \setminus B \in \delta(P \setminus B)$. Since $\lambda(\delta Q)$ is defined, then $\lambda(s)$ is defined by FDs1 (definition 3.13(3)), MD and Ts1. Thus, by proposition A.11, $\lambda(s \setminus A) = \lambda(s) \setminus B$ and so $\lambda(t) \in \delta(P \setminus B)$.

Case 2: There exists $w \in \Sigma^\omega$ such that $t = w \setminus A$ and, for every $u < w$,

$$u \in \tau_\perp Q = \{z \mid (z, \varnothing) \in \phi_\perp Q\}.$$

In the event that there exists $u < w$ such that $u \in \delta Q$, then there exists $u' \leq u$ such that $u' \in min\delta Q$, where $t = u' \setminus A$, and the proof proceeds as for Case 1 (we know that $t = u' \setminus A$ since, otherwise, $t \notin min\delta(Q \setminus A)$). We therefore assume that, for every $u < w$, $u \notin \delta Q$ and so, by DR2, that $(u, \varnothing) \in \phi Q$. Recall that $\lambda(\phi_\perp Q) \subseteq \phi_\perp P$. Thus, by FDs1 (definition 3.13(4)) and SFs1 (definition 3.12(3b)), $(\lambda(u), \varnothing) \in \phi_\perp P$ and so $\lambda(u) \in \tau_\perp P$ for every $u < w$. Moreover, since $\lambda(\phi Q)$ is defined and by SFs1 (definition 3.12(3)), $\lambda(u)$ is defined for every $u < w$. By lemma A.36, there exists $x \in \Sigma^\omega$ such that x is the least upper bound of the sequence of $\lambda(u)$ where $u < w$. By the prefix-closure of $\tau_\perp P$ (FD1), we have that $v \in \tau_\perp P$ for every $v < x$. Since $w \in \Sigma^\omega$ and $w \setminus A$ is finite, we have that $w = r \circ s$, where $r \in \Sigma^*$, $s \in A^\omega$ and $r \setminus A = w \setminus A$. By proposition A.11, we know that for any trace k such that $r \leq k < w$,

$$\lambda(t) = \lambda(w \setminus A) = \lambda(k \setminus A) = \lambda(k) \setminus B.$$

Hence, $x \setminus B = \lambda(t)$ and so $\lambda(t) \in \delta(P \setminus B)$.

We now show that $\lambda(\phi_\perp(Q \setminus A) \subseteq \phi_\perp(P \setminus B)$. By FDs1 (definition 3.13(1)), $\lambda(\phi_\perp(Q \setminus A))$ is defined and so, by FDs1 (definition 3.13(4)),

$$\lambda(\phi_\perp(Q \setminus A)) = \lambda(\phi(Q \setminus A)) \cup \{(t, R) \mid t \in \lambda(\delta(Q \setminus A)) \ \wedge \ R \subseteq \Sigma\}.$$

Thus, since $\lambda(\delta(Q \setminus A)) \subseteq \delta(P \setminus B)$ and by FD5, it is sufficient to show that $\lambda(\phi(Q \setminus A)) \subseteq \lambda(\phi_\perp P) \setminus B$. Since $\lambda(\phi_\perp Q)$ is defined and $\lambda(\phi_\perp Q) \subseteq \phi_\perp P$, we know that $\lambda(\phi Q) \subseteq \phi_\perp P$ by FDs1 (definition 3.13(4)). By definition,

$$\{(t \setminus B, R) \mid (t, R \cup B) \in \phi_\perp P\} \subseteq \phi_\perp(P \setminus B).$$

Since $\lambda(\phi(Q)) \setminus B$ is given by $\{(t \setminus B, R) \mid (t, R \cup B) \in \lambda(\phi Q)\}$, it follows that $\lambda(\phi Q) \setminus B \subseteq \phi_\perp(P \setminus B)$. Since $\lambda(\tau Q)$ is defined, then $\lambda([\![Q]\!]_{SF})$ is defined by proposition A.25. Hence, by proposition A.28 and SFs1 (definition 3.12(2)), $\lambda(\phi(Q \setminus A)) \subseteq \phi_\perp(P \setminus B)$. □

Lemma A.38. *Let Q_1, Q_2 be implementation processes and $Y = \alpha Q_1 \cap \alpha Q_2$, where $\lambda(\|_Y) = \|_Z$. Let P_1, P_2 be processes. If $\lambda([\![Q_i]\!]_{FD})$ is defined and $\lambda([\![Q_i]\!]_{FD}) \subseteq [\![P_i]\!]_{FD}$ for $i = 1, 2$ then:*

1. *$\lambda([\![Q_1 \|_Y Q_2]\!]_{FD})$ is defined.*
2. *$\lambda([\![Q_1 \|_Y Q_2]\!]_{FD}) \subseteq [\![P_1 \|_Z P_2]\!]_{FD}$.*

Proof. We assume that $\lambda([\![Q_i]\!]_{FD})$ is defined and $\lambda([\![Q_i]\!]_{FD}) \subseteq [\![P_i]\!]_{FD}$ for $i = 1, 2$. By proposition A.34, $\lambda(\tau Q_i)$ is defined for $i = 1, 2$ and so, by proposition A.15, $\lambda(\tau(Q_1 \|_Y Q_2))$ is defined. Hence, $\lambda([\![Q_1 \|_Y Q_2]\!]_{FD})$ is defined also by proposition A.34. By FDs1 (definition 3.13(2)), $\lambda(\phi_\perp Q_i) \subseteq \phi_\perp P_i$ and $\lambda(\delta Q_i) \subseteq \delta P_i$ for $i = 1, 2$. Also by FDs1 (definition 3.13) and for $i = 1, 2$, $\lambda(\phi_\perp Q_i)$ is defined and so $\lambda(\phi Q_i)$ is defined; moreover, $\lambda(\delta Q_i)$ is defined.

By FDs1 (definition 3.13(2)), it suffices to show that:

- $\lambda(\delta(Q_1 \|_Y Q_2)) \subseteq \delta(P_1 \|_Z P_2)$.
- $\lambda(\phi_\perp(Q_1 \|_Y Q_2)) \subseteq \phi_\perp(P_1 \|_Z P_2)$.

We first show that $\lambda(\delta(Q_1 \|_Y Q_2)) \subseteq \delta(P_1 \|_Z P_2)$. Let $t \in min\delta(Q_1 \|_Y Q_2)$. By FDs1 (definition 3.13(3)) and FD4, it is sufficient to show that $\lambda(t) \in \delta(P_1 \|_Z P_2)$. Since $t \in min\delta(Q_1 \|_Y Q_2)$ and so $t \in \delta(Q_1 \|_Y Q_2)$, there exist $s \in \tau_\perp Q_1$, $u \in \tau_\perp Q_2$ such that $t \in (s \|_Y u)$ and $s \in \delta Q_1$ or $u \in \delta Q_2$. If $s \in \delta Q_1$ then $s \in min\delta Q_1$, since otherwise there exists $w < t$ such that $w \in \delta(Q_1 \|_Y Q_2)$ and so $t \notin min\delta(Q_1 \|_Y Q_2)$. Similarly, if $u \in \delta Q_2$ then $u \in min\delta Q_2$. For $i = 1, 2$, we observe that $min\delta Q_i \subseteq \tau Q_i$ by MD and

$\tau_\perp Q_i = \tau Q_i \cup \delta Q_i$ by DR1. Thus, $s \in \tau Q_1$, $u \in \tau Q_2$ and either $s \in min\delta Q_1$ or $u \in min\delta Q_2$.

Wlog, we assume that $s \in min\delta Q_1$. By FDS1 (definition 3.13(3)) and since $\lambda(\delta Q_1) \subseteq \delta P_1$, $\lambda(s) \in \delta P_1$ and so $\lambda(s) \in \tau_\perp P_1$ by DR1. We show that $\lambda(u) \in \tau_\perp P_2$ by considering each of two cases in turn.

Case 1: $u \notin \delta Q_2$ and so $(u, \varnothing) \in \phi Q_2$ by DR3. Recall that $\lambda(\phi_\perp Q_2) \subseteq \phi_\perp P_2$. Thus, by FDS1 (definition 3.13(4)) and SFS1 (definition 3.12(3b)), $(\lambda(u), \varnothing) \in \phi_\perp P_2$ and so $\lambda(u) \in \tau_\perp P_2$.

Case 2: $u \in \delta Q_2$. We have seen above that, in this case, $u \in min\delta Q_2$. Thus, $\lambda(u) \in \delta P_2$ by FDS1 (definition 3.13(3)) and since $\lambda(\delta Q_2) \subseteq \delta P_2$. The proof of this case then follows by DR1.

Thus, $\lambda(s) \parallel_Z \lambda(u) \subseteq \delta(P_1 \parallel_Z P_2)$. Since $s \in \tau Q_1$, $u \in \tau Q_2$ and $\lambda(\tau Q_i)$ is defined for $i = 1, 2$, $\lambda(s)$ and $\lambda(u)$ are defined by TS1. Also since $s \in \tau Q_1$ and $u \in \tau Q_2$, $events(s) \subseteq \alpha Q_1$ and $events(u) \subseteq \alpha Q_2$ by PA1. Hence, $[[events(s)]] \subseteq \alpha Q_1$, $[[events(u)]] \subseteq \alpha Q_2$ and so

$$[[events(s)]] \cap [[events(u)]] \subseteq \alpha Q_1 \cap \alpha Q_2 = Y.$$

Since $\alpha P, \alpha Q \in AllSet$ by S2, then $Y \in AllSet$ by proposition A.5(1). Hence, $Z = \lambda(Y)$ by S7. Thus, since $t \in s \parallel_Y u$, $\lambda(t) \in \lambda(s) \parallel_Z \lambda(u)$ by proposition A.12 and so $\lambda(t) \in \delta(P_1 \parallel_Z P_2)$.

We now show that $\lambda(\phi_\perp(Q_1 \parallel_Y Q_2)) \subseteq \phi_\perp(P_1 \parallel_Z P_2)$. By FDS1 (definition 3.13(1)), $\lambda(\phi_\perp(Q_1 \parallel_Y Q_2))$ is defined and so, by FDS1 (definition 3.13(4)),

$$\lambda(\phi_\perp(Q_1 \parallel_Y Q_2)) = \lambda(\phi(Q_1 \parallel_Y Q_2)) \cup \{(t, R) \mid t \in \lambda(\delta(Q_1 \parallel_Y Q_2)) \wedge R \subseteq \Sigma\}.$$

Thus, since $\lambda(\delta(Q_1 \parallel_Y Q_2)) \subseteq \delta(P_1 \parallel_Z P_2)$ and by FD5, it is sufficient to show that $\lambda(\phi(Q_1 \parallel_Y Q_2)) \subseteq \phi_\perp(P_1 \parallel_Z P_2)$. Since $\lambda(\phi_\perp Q_i)$ is defined and $\lambda(\phi_\perp Q_i) \subseteq \phi_\perp P_i$, we know that $\lambda(\phi Q_i) \subseteq \phi_\perp P_i$ for $i = 1, 2$ by FDS1 (definition 3.13(4)). By definition, $\phi_\perp(P_1 \parallel_Z P_2)$ contains the following:

$$\{(w, S \cup U) \mid (\exists (s, S) \in \phi_\perp P_1, (u, U) \in \phi_\perp P_2)\, w \in (s \parallel_Z u) \wedge S - Z = U - Z\}.$$

Also by definition, $\lambda(\phi Q_1) \parallel_Z \lambda(\phi Q_2)$ is given by:

$$\{(w, S \cup U) \mid (\exists (s, S) \in \lambda(\phi Q_1), (u, U) \in \lambda(\phi Q_2))\, w \in (s \parallel_Z u) \wedge S - Z = U - Z\}.$$

Thus,

$$\lambda(\phi Q_1) \parallel_Z \lambda(\phi Q_2) \subseteq \phi_\perp(P_1 \parallel_Z P_2).$$

Since $\lambda(\tau Q_1)$, $\lambda(\tau Q_2)$ are defined, then $\lambda([\![Q_1]\!]_{SF})$, $\lambda([\![Q_2]\!]_{SF})$ are defined by proposition A.25. Thus, by proposition A.30 and SFS1 (definition 3.12(2)),

$$\lambda(\phi(Q_1 \parallel_Y Q_2)) \subseteq \lambda(\phi Q_1) \parallel_Z \lambda(\phi Q_2)$$

and so $\lambda(\phi(Q_1 \parallel_Y Q_2)) \subseteq \phi_\perp(P_1 \parallel_Z P_2)$. □

Proof of theorem 3.36

Proof. The proof is similar to that of theorem 3.1, using lemmas A.35, A.37 and A.38 in place of conditions RAH1-3. □

Appendix B

Proofs from chapter 4

B.1 Proofs from section 4.1

Proof of proposition 4.3

Proof. We observe the following:

$$\begin{aligned} Y &= \bigcup\{A_i \mid ep_i \in EP(P)\} \cap \bigcup\{A_j \mid ep_j \in EP(Q)\} && \text{(by def. 4.5)} \\ &= \bigcup\{A_i \mid ep_i \in EP(P) \cap EP(Q)\} && \text{(by E}\textsc{p-Uni1)} \end{aligned}$$

□

Proof of proposition 4.5

Proof. $EP(P \otimes_Y Q)$ is given by the following:

$$\begin{aligned} & \{ep_i \in EP \mid A_i \cap \beta(P \otimes_Y Q) \neq \varnothing\} && \text{(by def. 4.4)} \\ = \; & \{ep_i \in EP \mid A_i \cap ((\beta(P) \cup \beta(Q)) - Y) \neq \varnothing\} && \text{(by figure 2.5)} \\ = \; & \{ep_i \in EP - (EP(P) \cap EP(Q)) \mid A_i \cap (\beta(P) \cup \beta(Q)) \neq \varnothing\} \\ & && \text{(by prop. 4.3 and E}\textsc{p-Uni1)} \\ = \; & \{ep_i \in EP \mid A_i \cap (\beta(P) \cup \beta(Q)) \neq \varnothing\} - (EP(P) \cap EP(Q)) \\ = \; & (EP(P) \cup EP(Q)) - (EP(P) \cap EP(Q)) && \text{(by def. 4.4)} \end{aligned}$$

□

Proof of proposition 4.6

Proof. $\alpha(P \otimes_Y Q)$ is given by the following:

$$
\begin{aligned}
& \bigcup\{A_i \mid ep_i \in EP(P \otimes_Y Q)\} && \text{(by def. 4.5)} \\
= {} & \bigcup\{A_i \mid ep_i \in (EP(P) \cup EP(Q)) - (EP(P) \cap EP(Q))\} \\
& && \text{(by prop. 4.5)} \\
= {} & \bigcup\{A_i \mid ep_i \in (EP(P) \cup EP(Q))\} - \\
& \bigcup\{A_j \mid ep_j \in (EP(P) \cap EP(Q))\} && \text{(by EP-UNI1)} \\
= {} & (\alpha P \cup \alpha Q) - (\alpha P \cap \alpha Q) && \text{(by def. 4.5 and prop. 4.3)}
\end{aligned}
$$

□

B.2 Example processes used in proofs

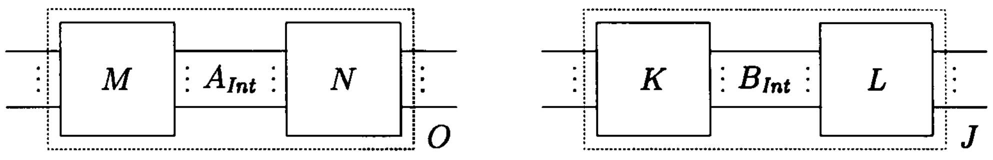

Figure B.1: Processes in proofs from chapter 4

The processes from figure B.1 are used in the statement and proof of most of the results which follow in this chapter. Processes M and N are implementation processes; K and L are the corresponding specification processes. We also denote:

- $I = M \parallel_{A_{Int}} N$.
- $H = K \parallel_{B_{Int}} L$.
- $O = M \otimes_{A_{Int}} N$.
- $J = K \otimes_{B_{Int}} L$.

The following are assumed to hold:

- $Comm(A_i, M) \neq Comm(A_i, N)$ for $ep_i \in EP(M) \cap EP(N)$.
- $A_{Int} = \alpha M \cap \alpha N$.
- $B_{Int} = extr^{set}(A_{Int})$.

As a result, the composition used to define O meets the restrictions REP1 and REP2. In the proofs in the remainder of this chapter, M, N, K, L, H, I, J, O are as described here.

B.3 Proofs from section 4.2

Results used in the proof of theorem 4.8

Proposition B.1. *The following hold:*

1. $A_{Int} = \bigcup\{A_i \mid ep_i \in EP(M) \cap EP(N)\}$.
2. $B_{Int} = \bigcup\{B_i \mid ep_i \in EP(M) \cap EP(N)\}$.

Proof. 1. The proof is immediate by proposition 4.3.

2. The proof is immediate by part 1 of the proposition and definition 4.2. □

Proposition B.2. *Let $ep_i \in EP$. If $A_i \cap A_{Int} \neq \varnothing$, then $A_i \subseteq A_{Int}$.*

Proof. We assume that $A_i \cap A_{Int} \neq \varnothing$. Hence, by proposition B.1(1), there exists $ep_j \in EP(M) \cap EP(N)$ such that $A_i \cap A_j \neq \varnothing$. Thus, by EP-UNI1, $ep_i = ep_j$ and so $A_i \subseteq A_{Int}$ by proposition B.1(1). □

Proposition B.3. *Let Q be an implementation process. Then the following hold:*

1. *$Dom_{EP(Q)}$ is non-empty and prefix-closed.*
2. *$extr_{EP(Q)}$ over traces is monotonic and strict.*

Proof. 1. The proof is immediate by EP3-T and TR-GLOBAL1.

2. That both properties hold follows from TR-GLOBAL2. □

Proposition B.4. *Let Q be an implementation process such that $\alpha Q \subseteq Fvis$. Moreover, let $t \in \tau Q$. Then:*

1. *$t \in Dom_{EP(Q)}$.*
2. *$extr_{EP(Q)}(t) = t$.*
3. *$extr_{EP(Q)}(\tau Q) = \tau Q$.*

Proof. We first show the following.

$$Dom_{EP(Q)} = (\alpha Q)^* \tag{B.1}$$

By definition 4.5, $\alpha Q = \bigcup\{A_i \mid ep_i \in EP(Q)\}$. Hence, since $\alpha Q \subseteq Fvis$, $A_i \subseteq Fvis$ for every $ep_i \in EP(Q)$. Thus, by EP3-FVI, $Dom_i = A_i^*$ for $ep_i \in EP(Q)$ and so $Dom_{EP(Q)} = (\alpha Q)^*$ by TR-GLOBAL1.

Hence, we have shown (B.1) and now proceed with the proof proper.

1. By PA1 and (B.1), $t \in (\alpha Q)^* = Dom_{EP(Q)}$.

2. By definition 4.5 and since $\alpha Q \subseteq Fvis$, $A_i \subseteq Fvis$ for every $ep_i \in EP(Q)$. By part 1 of the proposition, $t \in Dom_{EP(Q)}$. The proof then follows by a straightforward induction on the length of t using proposition B.3(1), TR-GLOBAL2 and EP4-FVI.

3. By PA1 and (B.1), $\tau Q \subseteq (\alpha Q)^* = Dom_{EP(Q)}$. Thus, $\tau_{Dom_{EP(Q)}} Q = \tau Q$ by definition 4.8. Hence, by TR-DEF1 and part 2 of the proposition, $extr_{EP(Q)}(\tau Q) = \tau Q$. □

Proof of theorem 4.7

Proof. ($\Longrightarrow$) We assume that $Q \sqsupseteq_T^{EP(Q)} P$. Hence, by TR-DEF2 and proposition B.4(3), $\tau Q \subseteq \tau P$.

($\Longleftarrow$) We assume that $Q \sqsupseteq_T P$. Thus, $extr_{EP(Q)}(\tau Q) \subseteq \tau P$ by proposition B.4(3). By proposition B.4(1), $t \in Dom_{EP(Q)}$ for all $t \in \tau Q$ and so Q meets Dom-T-check. Thus, by TR-DEF2, $Q \sqsupseteq_T^{EP(Q)} P$. □

Proposition B.5. *Let $t \circ \langle a \rangle \in Dom_{EP(I)}$ be such that, for some trace r, $extr_{EP(I)}(t \circ \langle a \rangle) = extr_{EP(I)}(t) \circ r$.*

1. *If $a \in A_{Int}$ then $events(r) \subseteq B_{Int}$.*

2. *If $a \notin A_{Int}$ then $events(r) \cap B_{Int} = \varnothing$.*

Proof. By TR-GLOBAL1, there exists $ep_i \in EP(I)$ such that $a \in A_i$. Thus, by TR-GLOBAL2 and EP4, $events(r) \subseteq B_i$.

1. We assume $a \in A_{Int}$ and so $A_i \cap A_{Int} \neq \varnothing$. Thus, by proposition B.1(1), there exists $ep_j \in EP(M) \cap EP(N)$ such that $A_i \cap A_j \neq \varnothing$. Hence, by EP-UNI1, $ep_i = ep_j$ and so $events(r) \subseteq B_i \subseteq B_{Int}$ by proposition B.1(2).

2. We assume $A \notin A_{Int}$ and so $A_i \not\subseteq A_{Int}$. Hence, by proposition B.1(1), there does not exist $ep_j \in EP(M) \cap EP(N)$ such that $A_i = A_j$ and so $ep_i \notin EP(M) \cap EP(N)$. Thus, by EP-UNI1 and proposition B.1(2), $B_i \cap B_{Int} = \varnothing$ and so $events(r) \cap B_{Int} = \varnothing$. □

Proposition B.6. *Let $s \in \tau M$, $u \in \tau N$ and $t \in (s \parallel_{A_{Int}} u)$.*

1. *If $a \in A_{Int}$, then there exists $ep_i \in EP(M) \cap EP(N)$ such that $a \in A_i$ and $t\lceil A_i = s\lceil A_i = u\lceil A_i$.*

2. *If $a \notin A_{Int}$ and $a \in events(s)$, then there exists $ep_i \in EP(M) - EP(N)$ such that $a \in A_i$ and $t\lceil A_i = s\lceil A_i$; moreover, $A_i \cap A_{Int} = \varnothing$.*

Proof. 1. We assume $a \in A_{Int}$. By proposition B.1(1), there exists $ep_i \in EP(M) \cap EP(N)$ such that $a \in A_i$ and $A_i \subseteq A_{Int}$. By TRP, $t\lceil A_{Int} = s\lceil A_{Int} = u\lceil A_{Int}$ and so $t\lceil A_i = s\lceil A_i = u\lceil A_i$.

2. We assume $a \notin A_{Int}$ and $a \in events(s)$. By PA1 and definition 4.5, there exists $ep_i \in EP(M)$ such that $a \in A_i$ and so $A_i \not\subseteq A_{Int}$. Thus, $ep_i \notin EP(N)$ by proposition B.1(1). Hence, by PA1, definition 4.5 and EP-UNI1, $A_i \cap events(u) = \varnothing$. Thus, $t\lceil A_i = s\lceil A_i$. Moreover, since $A_i \not\subseteq A_{Int}$, $A_i \cap A_{Int} = \varnothing$ by proposition B.2. □

Proposition B.7. *Let $s \in \tau_{Dom_{EP(M)}}M$, $u \in \tau_{Dom_{EP(N)}}N$ and $t \in (s \parallel_{A_{Int}} u)$. Then $t \in Dom_{EP(I)}$ and $extr_{EP(I)}(t) \in (extr_{EP(M)}(s) \parallel_{B_{Int}} extr_{EP(N)}(u))$.*

Proof. Note by definition 4.4 that $EP(I) = EP(M) \cup EP(N)$. We proceed by induction on the length of t. In the base case, when $t = s = u = \langle\rangle$, the proof is immediate by proposition B.3(1,2). Let $t = p \circ \langle a\rangle$. We consider each of two cases in turn.

Case 1: $a \in A_{Int}$. In this case, $s = v \circ \langle a\rangle$, $u = w \circ \langle a\rangle$ for some v, w. By proposition B.3(1), $v \in \tau_{Dom_{EP(M)}}M$ and $w \in \tau_{Dom_{EP(N)}}N$. Moreover, by proposition B.6(1), there exists $ep_i \in EP(M) \cap EP(N)$ such that $a \in A_i$ and $t\lceil A_i = s\lceil A_i = u\lceil A_i$. We first show that $t \in Dom_{EP(I)}$. By the inductive hypothesis, $p \in Dom_{EP(I)}$ and so, by TR-GLOBAL1 and EP-UNI1, $t\lceil A_j = p\lceil A_j \in Dom_j$ for $ep_j \in EP(I) - \{ep_i\}$. Thus, by TR-GLOBAL1, it suffices to show that $t\lceil A_i \in Dom_i$. This follows since $s \in Dom_{EP(M)}$ and $t\lceil A_i = s\lceil A_i$. We now proceed with the remainder of the proof in this case. By TR-GLOBAL2 and since $t\lceil A_i = s\lceil A_i = u\lceil A_i$,

- $extr_{EP(I)}(t) = extr_{EP(I)}(p) \circ r$
- $extr_{EP(M)}(s) = extr_{EP(M)}(v) \circ r$
- $extr_{EP(N)}(u) = extr_{EP(N)}(w) \circ r$

such that, by proposition B.5(1), $events(r) \subseteq B_{Int}$. By the inductive hypothesis, $extr_{EP(I)}(p) \in (extr_{EP(M)}(v) \parallel_{B_{Int}} extr_{EP(N)}(w))$ and so

$$extr_{EP(I)}(p) \circ r \in (extr_{EP(M)}(v) \circ r \parallel_{B_{Int}} extr_{EP(N)}(w) \circ r).$$

Case 2: $a \notin A_{Int}$. In this case, wlog, we assume $s = v \circ \langle a\rangle$. Thus, by proposition B.6(2), there exists $ep_i \in EP(M) - EP(N)$ such that $a \in A_i$ and $t\lceil A_i = s\lceil A_i$. Hence, by TR-GLOBAL1 and since $s \in Dom_{EP(M)}$, $t\lceil A_i \in Dom_i$. Thus, $t \in Dom_{EP(I)}$ by TR-GLOBAL1 and EP-UNI1 and since $p \in Dom_{EP(I)}$ by the inductive hypothesis. By TR-GLOBAL2 and since $t\lceil A_i = s\lceil A_i$,

- $extr_{EP(I)}(t) = extr_{EP(I)}(p) \circ r$
- $extr_{EP(M)}(s) = extr_{EP(M)}(v) \circ r$

such that, by proposition B.5(2), $events(r) \cap B_{Int} = \varnothing$. By the inductive hypothesis, $extr_{EP(I)}(p) \in (extr_{EP(M)}(v) \parallel_{B_{Int}} extr_{EP(N)}(u))$ and so

$$extr_{EP(I)}(p) \circ r \in (extr_{EP(M)}(v) \circ r \parallel_{B_{Int}} extr_{EP(N)}(u)).$$

□

Proposition B.8. *Let* $t \in Dom_{EP(I)}$. *Then* $t \setminus A_{Int} \in Dom_{EP(O)}$ *and* $extr_{EP(O)}(t \setminus A_{Int}) = extr_{EP(I)}(t) \setminus B_{Int}$.

Proof. We proceed by induction on the length of t. In the base case, when $t = \langle\rangle$, the proof is immediate by proposition B.3(1,2). Let $t = u \circ \langle a \rangle$. Hence, by proposition B.3(1), $u \in Dom_{EP(I)}$ and, by TR-GLOBAL1, there exists $ep_i \in EP(I)$ such that $a \in A_i$. We consider each of two cases in turn.

Case 1: $a \in A_{Int}$. In this case, $t \setminus A_{Int} = u \setminus A_{Int} \in Dom_{EP(O)}$ by the inductive hypothesis. By TR-GLOBAL2, $extr_{EP(I)}(t) = extr_{EP(I)}(u) \circ r$ such that, by proposition B.5(1), $events(r) \subseteq B_{Int}$. Hence, by the inductive hypothesis,

$$extr_{EP(O)}(t \setminus A_{Int}) = extr_{EP(O)}(u \setminus A_{Int}) = extr_{EP(I)}(u) \setminus B_{Int}$$

and so

$$extr_{EP(O)}(t \setminus A_{Int}) = extr_{EP(I)}(u) \circ r \setminus B_{Int} = extr_{EP(I)}(t) \setminus B_{Int}.$$

Case 2: $a \notin A_{Int}$. Since $a \notin A_{Int}$, $A_i \not\subseteq A_{Int}$ and so $A_i \cap A_{Int} = \varnothing$ by proposition B.2. We first show that $t \setminus A_{Int} \in Dom_{EP(O)}$. By TR-GLOBAL1, it suffices to show that $(t \setminus A_{Int}) \lceil A_j \in Dom_j$ for every $ep_j \in EP(O)$. Let $ep_j \in EP(O)$. We consider each of two sub-cases in turn.

Case 2a: $j = i$. Since $t \in Dom_{EP(I)}$, $(t \setminus A_{Int}) \lceil A_i = t \lceil A_i \in Dom_i$ by TR-GLOBAL1 and since $A_i \cap A_{Int} = \varnothing$.

Case 2b: $j \neq i$. By the inductive hypothesis, $u \setminus A_{Int} \in Dom_{EP(O)}$. Thus, by TR-GLOBAL1 and EP-UNI1, $(t \setminus A_{Int}) \lceil A_j = (u \setminus A_{Int}) \lceil A_j \in Dom_j$.

Hence, we have shown that $t \setminus A_{Int} \in Dom_{EP(O)}$. Thus, by TR-GLOBAL1 and EP-UNI1 and since $a \in events(t \setminus A_{Int})$, $ep_i \in EP(O)$. We now show that $extr_{EP(O)}(t \setminus A_{Int}) = extr_{EP(I)}(t) \setminus B_{Int}$. By TR-GLOBAL2,

- $extr_{EP(I)}(t) = extr_{EP(I)}(u) \circ r$, such that $extr_i(t \lceil A_i) = extr_i(u \lceil A_i) \circ r$.
- $extr_{EP(O)}(t \setminus A_{Int}) = extr_{EP(O)}(u \setminus A_{Int}) \circ x$, such that $extr_i((t \setminus A_{Int}) \lceil A_i) = extr_i((u \setminus A_{Int}) \lceil A_i) \circ x$.

Thus, since $A_i \cap A_{Int} = \varnothing$, $extr_{EP(O)}(t \setminus A_{Int}) = extr_{EP(O)}(u \setminus A_{Int}) \circ r$. Moreover, by proposition B.5(2), $events(r) \cap B_{Int} = \varnothing$. Hence, by the inductive hypothesis,

$$extr_{EP(O)}(t \setminus A_{Int}) = extr_{EP(O)}(u \setminus A_{Int}) \circ r = extr_{EP(I)}(u) \setminus B_{Int} \circ r$$

and so

$$extr_{EP(O)}(t \setminus A_{Int}) = extr_{EP(I)}(u) \circ r \setminus B_{Int} = extr_{EP(I)}(t) \setminus B_{Int}.$$

□

Proposition B.9. *Assume that both M and N meet condition Dom-T-check. Let $t \in \tau O$ be such that $t \in (s \parallel_{A_{Int}} u) \setminus A_{Int}$, where $s \in \tau M$ and $u \in \tau N$. If $t \in Dom_{EP(O)}$, then $s \in Dom_{EP(M)}$ and $u \in Dom_{EP(N)}$.*

Proof. We assume $t \in Dom_{EP(O)}$. Let y be such that $y \in (s \parallel_{A_{Int}} u)$ and $y \setminus A_{Int} = t$. We proceed by induction on the length of y. In the base case, when $y = s = u = \langle\rangle$, the proof is immediate by proposition B.3(1). Let $y = x \circ \langle a \rangle$. By proposition B.3(1), $x \setminus A_{Int} \in Dom_{EP(O)}$. We consider each of two cases in turn.

Case 1: $a \in A_{Int}$. In this case, $s = v \circ \langle a \rangle$ and $u = w \circ \langle a \rangle$ for some v, w. Hence, by proposition B.6(1), there exists $ep_i \in EP(M) \cap EP(N)$ such that $a \in A_i$ and $s\lceil A_i = u\lceil A_i$. By definition 4.7 and since $Comm(A_i, M) \neq Comm(A_i, N)$, either $a \notin Proj_{EP(M)}$ or $a \notin Proj_{EP(N)}$ (this disjunction need not be exclusive by definition 4.7(1)). Wlog, we assume that $a \notin Proj_{EP(M)}$. Thus, $v\lceil Proj_{EP(M)} = s\lceil Proj_{EP(M)}$. Moreover, by the inductive hypothesis, $v \in Dom_{EP(M)}$ and so

$$s\lceil Proj_{EP(M)} = v\lceil Proj_{EP(M)} \in Dom_{EP(M)}\lceil Proj_{EP(M)}.$$

Hence, since M meets Dom-T-check, $s \in Dom_{EP(M)}$. Thus, by TR-GLOBAL1, $u\lceil A_i = s\lceil A_i \in Dom_i$. As a result, $u \in Dom_{EP(N)}$ by TR-GLOBAL1 and EP-UNI1 and since $w \in Dom_{EP(N)}$ by the inductive hypothesis.

Case 2: $a \notin A_{Int}$. In this case, wlog, we assume $s = v \circ \langle a \rangle$ and $tail(u) \neq a$. Thus, by the inductive hypothesis, $u \in Dom_{EP(N)}$. By proposition B.6(2), there exists $ep_i \in EP(M) - EP(N)$ such that $a \in A_i$, $y\lceil A_i = s\lceil A_i$ and $A_i \cap A_{Int} = \varnothing$. Thus, $t\lceil A_i = s\lceil A_i$ and, by proposition 4.5, $ep_i \in EP(O)$. Hence, since $t \in Dom_{EP(O)}$ and by TR-GLOBAL1, $s\lceil A_i \in Dom_i$. By the inductive hypothesis, $v \in Dom_{EP(M)}$ and so, by TR-GLOBAL1 and EP-UNI1, $s \in Dom_{EP(M)}$. □

Lemma B.10. *Let Q be an implementation process and $t \in \tau Q$. Then $t\lceil Proj_{EP(Q)} \in Dom_{EP(Q)}\lceil Proj_{EP(Q)}$ if and only if $t\lceil Proj_i \in Dom_i\lceil Proj_i$ for every $ep_i \in EP(Q)$.*

Proof. ($\Longrightarrow$) We assume $t\lceil Proj_{EP(Q)} \in Dom_{EP(Q)}\lceil Proj_{EP(Q)}$. Let $ep_i \in EP(Q)$. By definition 4.7(3), $t\lceil Proj_i \in Dom_i\lceil Proj_i$.

($\Longleftarrow$) We assume $t\lceil Proj_i \in Dom_i\lceil Proj_i$ for every $ep_i \in EP(Q)$. We first observe the following.

- $Dom_{EP(Q)} = |||_{1 \le i \le m} Dom_i$ by TR-GLOBAL1.
- $Proj_{EP(Q)} = \bigcup_{1 \le i \le m} Proj_i$ by definition 4.7(3).
- Let $ep_i \in EP(Q)$. Then the following hold.
  - $events(t) \subseteq A_i$ for every $t \in Dom_i$ by EP3-T.
  - $Proj_i \subseteq A_i$ by definition 4.7 and EP2.
  - $A_i \cap A_j = \varnothing$ for $ep_j \in EP(Q)$ such that $i \neq j$ by EP-UNI1.

Hence,

$$Dom_{EP(Q)} \lceil Proj_{EP(Q)} = |||_{1 \le i \le m}(Dom_i \lceil Proj_i)$$

and $t \lceil Proj_{EP(Q)} \in |||_{1 \le i \le m} t \lceil Proj_i$. Thus, since $t \lceil Proj_i \in Dom_i \lceil Proj_i$ for every $ep_i \in EP(Q)$, $t \lceil Proj_{EP(Q)} \in Dom_{EP(Q)} \lceil Proj_{EP(Q)}$. □

Proposition B.11. *Assume that both M and N meet condition Dom-T-check. Then O meets Dom-T-check.*

Proof. Let $t \in \tau O$ be such that $t \in (s \parallel_{A_{Int}} u) \setminus A_{Int}$, where $s \in \tau M$, $u \in \tau N$ and s, u are the shortest such traces. By Dom-T-check, it suffices to show that

$$t \lceil Proj_{EP(O)} \in (Dom_{EP(O)} \lceil Proj_{EP(O)}) \Longrightarrow t \in Dom_{EP(O)}.$$

We assume that $t \lceil Proj_{EP(O)} \in (Dom_{EP(O)} \lceil Proj_{EP(O)})$ and proceed by induction on the length of t. In the base case, when $t = \langle\rangle$, the proof is immediate by proposition B.3(1). Let $t = x \circ \langle a \rangle$. Since $a \in \alpha O$ by PA1, by definition 4.5 there exists $ep_i \in EP(O)$ such that $a \in A_i$. Wlog, and by proposition 4.5, we assume $ep_i \in EP(M)$ and $ep_i \notin EP(N)$. Thus, by definition 4.5, EP-UNI1 and PA1, $a \notin events(u)$. Hence, since s and u are as short as possible, $s = v \circ \langle a \rangle$ for some trace v and so $x \in (v \parallel_{A_{Int}} u) \setminus A_{Int}$. Before proceeding, we show the following.

$$s \lceil A_i = t \lceil A_i. \tag{B.2}$$

Let y be such that $y \in (s \parallel_{A_{Int}} u)$ and $t = y \setminus A_{Int}$. Since $ep_i \notin EP(N)$ and by definition 4.5, EP-UNI1 and PA1, $A_i \cap events(u) = \varnothing$. Thus, $y \lceil A_i = s \lceil A_i$. Since $ep_i \in EP(M) - EP(N)$, and by proposition B.1(1) and EP-UNI1, $A_i \cap A_{Int} = \varnothing$. Thus, $t \lceil A_i = (y \setminus A_{Int}) \lceil A_i = y \lceil A_i = s \lceil A_i$. Hence, we have shown (B.2).

Since $t \lceil Proj_{EP(O)} \in Dom_{EP(O)} \lceil Proj_{EP(O)}$ and by proposition B.3(1), $x \lceil Proj_{EP(O)} \in Dom_{EP(O)} \lceil Proj_{EP(O)}$. Hence, by the inductive hypothesis, $x \in Dom_{EP(O)}$. Thus, by TR-GLOBAL1 and EP-UNI1, it suffices to show that $t \lceil A_i \in Dom_i$. Moreover, by proposition B.9, $v \in Dom_{EP(M)}$. We now consider each of two cases in turn.

Case 1: $a \in \mathit{Proj}_{EP(O)}$. Since $t\lceil \mathit{Proj}_{EP(O)} \in \mathit{Dom}_{EP(O)}\lceil \mathit{Proj}_{EP(O)}$, then $s\lceil \mathit{Proj}_i = t\lceil \mathit{Proj}_i \in \mathit{Dom}_i\lceil \mathit{Proj}_i$ by lemma B.10, (B.2) and since $\mathit{Proj}_i \subseteq A_i$. Since $v \in \mathit{Dom}_{EP(M)}$, then $v\lceil \mathit{Proj}_{EP(M)} \in \mathit{Dom}_{EP(M)}\lceil \mathit{Proj}_{EP(M)}$ and so, by lemma B.10, EP-UNI1 and since $\mathit{Proj}_j \subseteq A_j$,

$$s\lceil \mathit{Proj}_j = v\lceil \mathit{Proj}_j \in \mathit{Dom}_j\lceil \mathit{Proj}_j$$

for every $ep_j \in EP(M) - \{ep_i\}$. Thus, $s\lceil \mathit{Proj}_{EP(M)} \in \mathit{Dom}_{EP(M)}\lceil \mathit{Proj}_{EP(M)}$ by lemma B.10. Hence, since M meets Dom-T-check, $s \in \mathit{Dom}_{EP(M)}$ and so, by TR-GLOBAL1 and (B.2), $t\lceil A_i \in \mathit{Dom}_i$.

Case 2: $a \notin \mathit{Proj}_{EP(O)}$. By definition 4.7(3) and since $a \in A_i$, $a \notin \mathit{Proj}_i$ and so $a \notin \mathit{Proj}_{EP(M)}$ by EP-UNI1. Hence, and since $v \in \mathit{Dom}_{EP(M)}$,

$$s\lceil \mathit{Proj}_{EP(M)} = v\lceil \mathit{Proj}_{EP(M)} \in \mathit{Dom}_{EP(M)}\lceil \mathit{Proj}_{EP(M)}.$$

Since M meets Dom-T-check, $s \in \mathit{Dom}_{EP(M)}$ and so, by TR-GLOBAL1 and (B.2), $t\lceil A_i \in \mathit{Dom}_i$. □

Proposition B.12. *If $M \sqsupseteq_T^{EP(M)} K$ and $N \sqsupseteq_T^{EP(N)} L$, $\mathit{extr}_{EP(O)}(\tau O) \subseteq \tau J$.*

Proof. We assume $M \sqsupseteq_T^{EP(M)} K$ and $N \sqsupseteq_T^{EP(N)} L$. Let $t \in \tau_{\mathit{Dom}_{EP(O)}} O$ be such that $s \in \tau M$, $u \in \tau N$ and $t \in (s \parallel_{A_{Int}} u)\backslash A_{Int}$. By TR-DEF1, it suffices to show that $\mathit{extr}_{EP(O)}(t) \in \tau J$. By TR-DEF2, M and N meet Dom-T-check and so, by proposition B.9, $s \in \tau_{\mathit{Dom}_{EP(M)}} M$ and $u \in \tau_{\mathit{Dom}_{EP(N)}} N$. Thus, by TR-DEF1 and TR-DEF2, $\mathit{extr}_{EP(M)}(s) \in \tau K$ and $\mathit{extr}_{EP(N)}(u) \in \tau L$. Hence,

$$(\mathit{extr}_{EP(M)}(s) \parallel_{B_{Int}} \mathit{extr}_{EP(N)}(u)) \setminus B_{Int} \subseteq \tau J.$$

Thus, by propositions B.7 and B.8,

$$\mathit{extr}_{EP(O)}(t) \in (\mathit{extr}_{EP(M)}(s) \parallel_{B_{Int}} \mathit{extr}_{EP(N)}(u)) \setminus B_{Int} \subseteq \tau J.$$

□

Lemma B.13. *If $M \sqsupseteq_T^{EP(M)} K$ and $N \sqsupseteq_T^{EP(N)} L$ then $O \sqsupseteq_T^{EP(O)} J$.*

Proof. We assume that $M \sqsupseteq_T^{EP(M)} K$ and $N \sqsupseteq_T^{EP(N)} L$. By TR-DEF2 and proposition B.11, O meets Dom-T-check. Hence, by TR-DEF2, it suffices to show that $\mathit{extr}_{EP(O)}(\tau O) \subseteq \tau J$ and so the proof follows by proposition B.12. □

Proof of theorem 4.8

Proof. We assume that $\alpha F_{impl}(Q_1, Q_2, \ldots, Q_n) \subseteq \mathit{Fvis}$ and $Q_i \sqsupseteq_T^{EP(Q_i)} P_i$ for $1 \leq i \leq n$. Let $Q = F_{impl}(Q_1, Q_2, \ldots, Q_n)$ and $P = F_{spec}(P_1, P_2, \ldots, P_n)$. By induction on n using lemma B.13, $Q \sqsupseteq_T^{EP(Q)} P$. Hence, by theorem 4.7 and since $\alpha Q \subseteq \mathit{Fvis}$, $Q \sqsupseteq_T P$. □

B.4 Proofs from section 4.4

Results used in the proof of theorem 4.10

Proposition B.14. *Let Q be an implementation process. Then $Dom_{EP(Q)}$ is the prefix-closure of $dom_{EP(Q)}$.*

Proof. By EP3-SF, $Dom_i = Pref(dom_i)$ for $ep_i \in EP(Q)$. Thus, by SF-GLOBAL1 andTR-GLOBAL1, $dom_{EP(Q)} \subseteq Dom_{EP(Q)}$ and, for $t \in Dom_{EP(Q)}$, there exists $u \in dom_{EP(Q)}$ such that $t \leq u$. Finally, $Dom_{EP(Q)}$ is prefix-closed by proposition B.3(1). □

Proposition B.15. *Let Q be an implementation process such that $\alpha Q \subseteq Fvis$. Then $extr_{EP(Q)}(\phi Q) = \phi Q$.*

Proof. By definition 4.5, $A_i \subseteq Fvis$ for every $ep_i \in EP(Q)$. Thus, by definitions 4.2 and 4.5 and EP1-FVI,

$$extr^{set}(\alpha Q) = \alpha Q. \tag{B.3}$$

We now show the following.

Let $(t, R) \in \phi_{dom_{EP(Q)}}Q$ such that $R \subseteq \alpha Q$. Then $extr^{ref}_{EP(Q)}(R, t, Q) = R$. (B.4)

By definition 4.11(2) and proposition B.14, $t \in dom_{EP(Q)} \subseteq Dom_{EP(Q)}$ and so:

$$\begin{aligned} extr^{ref}_{EP(Q)}(R,t,Q) &= \textstyle\bigcup_{1\leq i\leq m} extr^{ref}_i(R \cap A_i, t\lceil A_i, Q) && \text{(SF-GLOBAL2)}\\ &= \textstyle\bigcup_{1\leq i\leq m} R \cap A_i && \text{(by EP5-FVI)}\\ &= R \cap \textstyle\bigcup_{1\leq i\leq m} A_i \\ &= R \cap \alpha Q && \text{(by definition 4.5)}\\ &= R. \end{aligned}$$

Hence, we have shown (B.4). Recall that $A_i \subseteq Fvis$ for every $ep_i \in EP(Q)$ by definition 4.5. Thus, by EP3A-FVI, TR-GLOBAL1 and SF-GLOBAL1, $Dom_{EP(Q)} = dom_{EP(Q)}$. Moreover, for every $(t, R) \in \phi Q$, $t \in \tau Q$ by SF2 and so $t \in Dom_{EP(Q)} = dom_{EP(Q)}$ by proposition B.4(1). Thus, by definition 4.11(2),

$$\phi_{dom_{EP(Q)}}Q = \phi Q. \tag{B.5}$$

Let $(t, R) \in \phi_{dom_{EP(Q)}}Q$. Then $t \in \tau Q$ by SF2 and so $extr_{EP(Q)}(t) = t$ by proposition B.4(2). Hence, and by (B.3), (B.4), (B.5) and SF-DEF2,

$$extr_{EP(Q)}(\phi Q) = \{(t, X) \mid (t, R) \in \phi Q \ \wedge \ R \subseteq \alpha Q \ \wedge \ X \subseteq R \cup (\Sigma - \alpha Q)\}.$$

Thus, $extr_{EP(Q)}(\phi Q) = \phi Q$ by PA2 and SF3. □

Proof of theorem 4.9

Proof. ($\Longrightarrow$) We assume that $Q \sqsupseteq_{SF}^{EP(Q)} P$. Hence, by SF-DEF3 and SF-DEF1, $extr_{EP(Q)}(\tau Q) \subseteq \tau P$ and $extr_{EP(Q)}(\phi Q) \subseteq \phi P$. Thus, by propositions B.4(3) and B.15, $\tau Q \subseteq \tau P$ and $\phi Q \subseteq \phi P$.

($\Longleftarrow$) We assume that $Q \sqsupseteq_{SF} P$. Thus, by propositions B.4(3) and B.15, $extr_{EP(Q)}(\tau Q) \subseteq \tau P$ and $extr_{EP(Q)}(\phi Q) \subseteq \phi P$. By proposition B.4(1), $t \in Dom_{EP(Q)}$ for all $t \in \tau Q$ and so Q meets Dom-T-check. Q meets Dom-SF-check since, by definition 4.5, $A_i \subseteq Fvis$ for every $ep_i \in EP(Q)$. Thus, by SF-DEF1 and SF-DEF3, $Q \sqsupseteq_{SF}^{EP(Q)} P$. □

Lemma B.16. *Let* $(s, S) \in \phi_{Dom_{EP(M)}} M$ *and* $(u, U) \in \phi_{Dom_{EP(N)}} N$ *be such that* $S \subseteq \alpha M$, $U \subseteq \alpha N$ *and* $(s \parallel_{A_{Int}} u) \neq \varnothing$. *If* $A_{Int} \subseteq S \cup U$, *then* $B_{Int} \subseteq extr^{ref}_{EP(M)}(S, s, M) \cup extr^{ref}_{EP(N)}(U, u, N)$.

Proof. We assume that $A_{Int} \subseteq S \cup U$. By SF-GLOBAL2, $extr^{ref}_{EP(M)}(S, s, M) \cup extr^{ref}_{EP(N)}(U, u, N)$ is given by

$$\bigcup_{ep_i \in EP(M)} extr^{ref}_i(S \cap A_i, s\lceil A_i, M) \cup \bigcup_{ep_j \in EP(N)} extr^{ref}_j(U \cap A_j, u\lceil A_j, N).$$

Thus, and by proposition B.1(2), it suffices to show that

$$B_i = extr^{ref}_i(S \cap A_i, s\lceil A_i, M) \cup extr^{ref}_i(U \cap A_i, u\lceil A_i, N)$$

for every $ep_i \in EP(M) \cap EP(N)$. Let $ep_i \in EP(M) \cap EP(N)$. By proposition B.1(1), $A_i \subseteq A_{Int}$ and so, since $A_{Int} \subseteq S \cup U$,

$$(S \cap A_i) \cup (U \cap A_i) = (S \cup U) \cap A_i = A_i.$$

By EP1, we consider each of two cases in turn.

Case 1: $A_i \subseteq Fvis$. In this case, by EP5-FVI,

$$extr^{ref}_i(S \cap A_i, s\lceil A_i, M) = S \cap A_i \text{ and } extr^{ref}_i(U \cap A_i, u\lceil A_i, N) = U \cap A_i.$$

Moreover, by EP1-FVI, $(S \cap A_i) \cup (U \cap A_i) = A_i = B_i$.

Case 2: $A_i \cap Fvis = \varnothing$. Since $(s \parallel_{A_{Int}} u) \neq \varnothing$, then $s\lceil A_{Int} = u\lceil A_{Int}$ by TRP and so $s\lceil A_i = u\lceil A_i$ since $A_i \subseteq A_{Int}$. Wlog, we assume that $Comm(A_i, M) = Left$ and $Comm(A_i, N) = Right$. Since $(S \cap A_i) \cup (U \cap A_i) = A_i$ and $s\lceil A_i = u\lceil A_i$, and by definition 4.9, either

$$S \cap A_i \notin ref_i(s\lceil A_i) \text{ or } U \cap A_i \notin \overline{ref}_i(u\lceil A_i).$$

Hence, either $extr^{ref}_i(S \cap A_i, s\lceil A_i, M) = B_i$ or $extr^{ref}_i(U \cap A_i, u\lceil A_i, N) = B_i$ by definition 4.10. □

Proposition B.17. *Let $(t, R) \in \phi O$. Then there exist $(s, S) \in \phi M$ and $(u, U) \in \phi N$ such that:*

- $t \in (s \parallel_{A_{Int}} u) \setminus A_{Int}$.
- $S \subseteq \alpha M$ *and* $U \subseteq \alpha N$.
- $R \cup A_{Int} = (S \cup U) \cup Z$, *where* $Z \subseteq (\Sigma - (\alpha M \cup \alpha N))$.

Proof. By definition of the hiding operator in the stable failures model, there exists $(w, R \cup A_{Int}) \in \phi I$ where $w \setminus A_{Int} = t$. Recall also that $I = M \parallel_{A_{Int}} N$ and $A_{Int} = \alpha M \cap \alpha N$. Hence, the proof follows by theorem 2.20. □

Proposition B.18. *Assume that M and N meet Dom-T-check and Dom-SF-check. Then O meets Dom-SF-check.*

Proof. Let $(t, R) \in \phi_{Dom_{EP(O)}} O$ be such that $R \subseteq \alpha O$ and let $ep_i \in EP(O)$ be such that $A_i \cap Fvis = \varnothing$. Moreover, assume that $extr_i^{ref}(R \cap A_i, t\lceil A_i, O) = B_i$. By definition of Dom-SF-check, it suffices to show that $t\lceil A_i \in dom_i$. By proposition B.17, there exist $(s, S) \in \phi M$ and $(u, U) \in \phi N$ such that:

- $t \in (s \parallel_{A_{Int}} u) \setminus A_{Int}$.
- $S \subseteq \alpha M$ and $U \subseteq \alpha N$.
- $R \cup A_{Int} = (S \cup U) \cup Z$, where $Z \subseteq (\Sigma - (\alpha M \cup \alpha N))$.

By SF2, $t \in \tau O$, $s \in \tau M$ and $u \in \tau N$. Thus, by proposition B.9 and definition 4.11(1), $(s, S) \in \phi_{Dom_{EP(M)}} M$ and $(u, U) \in \phi_{Dom_{EP(N)}} N$. Wlog, and by proposition 4.5, we assume that $ep_i \in EP(M) - EP(N)$. We then observe the following:

- $A_i \cap A_{Int} = \varnothing$ by proposition B.1(1) and EP-UNI1.
- By definition 4.5, $A_i \subseteq \alpha M$ and, also by EP-UNI1, $A_i \cap \alpha N = \varnothing$.
- $A_i \cap events(u) = \varnothing$ by SF2, PA1 and since $A_i \cap \alpha N = \varnothing$.

Since $A_i \cap events(u) = \varnothing$ and $A_i \cap A_{Int} = \varnothing$, $t\lceil A_i = s\lceil A_i$. Since $A_i \cap A_{Int} = \varnothing$, $A_i \subseteq \alpha M$ and $A_i \cap \alpha N = \varnothing$,

$$R \cap A_i = (R \cup A_{Int}) \cap A_i = (S \cup U \cup Z) \cap A_i = S \cap A_i.$$

By definition 4.6, $Comm(A_i, O) = Comm(A_i, M)$. Thus, by definition 4.10,

$$extr_i^{ref}(S \cap A_i, s\lceil A_i, M) = extr_i^{ref}(R \cap A_i, t\lceil A_i, O) = B_i.$$

Hence, $t\lceil A_i = s\lceil A_i \in dom_i$ since M meets Dom-SF-check. □

Lemma B.19. *Assume that M and N meet Dom-T-check and Dom-SF-check. Let $(t, R) \in \phi_{dom_{EP(O)}} O$ be such that $R \subseteq \alpha O$. Let $(s, S) \in \phi M$ and $(u, U) \in \phi N$ be such that:*

- $t \in (s \parallel_{A_{Int}} u) \setminus A_{Int}$.
- $S \subseteq \alpha M$ *and* $U \subseteq \alpha N$.
- $R \cup A_{Int} = (S \cup U) \cup Z$, *where* $Z \subseteq (\Sigma - (\alpha M \cup \alpha N))$.

Then $(s, S) \in \phi_{dom_{EP(M)}} M$ and $(u, U) \in \phi_{dom_{EP(N)}} N$.

Proof. We show that $(s, S) \in \phi_{dom_{EP(M)}} M$; that $(u, U) \in \phi_{dom_{EP(N)}} N$ may be proved in a similar way. By definition 4.11(2), it suffices to show that $s \in dom_{EP(M)}$. Let $ep_i \in EP(M)$. Then, by SF-GLOBAL1, it suffices to show that $s \lceil A_i \in dom_i$. By SF2, $t \in \tau O$, $s \in \tau M$ and $u \in \tau N$. By proposition B.14, $t \in dom_{EP(O)} \subseteq Dom_{EP(O)}$. Thus, since M and N both meet Dom-T-check and by proposition B.9, $s \in Dom_{EP(M)}$ and $u \in Dom_{EP(N)}$. We now consider each of three cases in turn.

Case 1: $A_i \subseteq Fvis$. In this case, $s \lceil A_i \in dom_i$ by EP3-FVI and EP3A-FVI.

Case 2: $A_i \cap Fvis = \varnothing$ and $ep_i \in EP(N)$. By proposition B.1(1), $A_i \subseteq A_{Int}$ and so, by TRP, $s \lceil A_i = u \lceil A_i$. Recall that $A_{Int} = \alpha M \cap \alpha N$. Thus, since $R \cup A_{Int} = (S \cup U) \cup Z$, then $A_i \subseteq A_{Int} \subseteq S \cup U$ and so

$$A_i = (S \cap A_i) \cup (U \cap A_i).$$

Wlog, we assume that $Comm(A_i, M) = Left$ and so $Comm(A_i, N) = Right$. By definition 4.9, either $S \cap A_i \notin ref_i(s \lceil A_i)$ or $U \cap A_i \notin \overline{ref}_i(u \lceil A_i)$ and so, by definition 4.10, either

$$extr_i^{ref}(S \cap A_i, s \lceil A_i, M) = B_i \text{ or } extr_i^{ref}(U \cap A_i, u \lceil A_i, N) = B_i.$$

Hence, since M and N both meet Dom-SF-check, $s \lceil A_i = u \lceil A_i \in dom_i$.

Case 3: $A_i \cap Fvis = \varnothing$ and $ep_i \notin EP(N)$. In this case, by proposition 4.5, $ep_i \in EP(O)$. By definition 4.5 and EP-UNI1, $A_i \cap \alpha N = \varnothing$. Thus, $A_i \cap events(u) = \varnothing$ by SF2 and PA1. By proposition B.1(1) and EP-UNI1, $A_i \cap A_{Int} = \varnothing$. Hence, $s \lceil A_i = t \lceil A_i$ and so, since $t \in dom_{EP(O)}$, $s \lceil A_i = t \lceil A_i \in dom_i$ by SF-GLOBAL1. □

Lemma B.20. *Assume that $extr_{EP(M)}(\phi M) \subseteq \phi K$ and $extr_{EP(N)}(\phi N) \subseteq \phi L$. Let $(s, S) \in \phi_{dom_{EP(M)}} M$ and $(u, U) \in \phi_{dom_{EP(N)}} N$ be such that $S \subseteq \alpha M$ and $U \subseteq \alpha N$. Then $(w, Y) \in \phi H$ such that*

- $w \in (extr_{EP(M)}(s) \parallel_{B_{Int}} extr_{EP(N)}(u))$.
- $Y = extr^{ref}_{EP(M)}(S, s, M) \cup extr^{ref}_{EP(N)}(U, u, N) \cup (\Sigma - extr^{set}(\alpha M \cup \alpha N))$.

Proof. We shall use the following abbreviations in order to ease the presentation:

- $\mathcal{S} \triangleq extr^{ref}_{EP(M)}(S, s, M)$.
- $\mathcal{U} \triangleq extr^{ref}_{EP(N)}(U, u, N)$.
- $\mathcal{Z} \triangleq \Sigma - extr^{set}(\alpha M \cup \alpha N)$.

By SF-DEF2,

- $(extr_{EP(M)}(s), \mathcal{S} \cup (\Sigma - extr^{set}(\alpha M))) \in \phi K$.
- $(extr_{EP(N)}(u), \mathcal{U} \cup (\Sigma - extr^{set}(\alpha N))) \in \phi L$.

We then observe that, by definitions 4.2 and 4.5,

$$\mathcal{Z} = \Sigma - (extr^{set}(\alpha M) \cup extr^{set}(\alpha N)).$$

Let $\mathcal{S}' = \mathcal{S} \cup \mathcal{Z} \cup (\mathcal{U} - B_{Int})$ and $\mathcal{U}' = \mathcal{U} \cup \mathcal{Z} \cup (\mathcal{S} - B_{Int})$. By definition 4.11(2) and proposition B.14, $u \in Dom_{EP(N)}$. Hence, by SF-GLOBAL2, definition 4.10, EP1-FVI and EP5-FVI, $\mathcal{U} \subseteq \bigcup\{B_i \mid ep_i \in EP(N)\}$. Thus, by definitions 4.2 and 4.5, $\mathcal{U} \subseteq extr^{set}(\alpha N)$. By proposition B.1(2), EP-UNI1 and definitions 4.2 and 4.5, $B_{Int} = extr^{set}(\alpha M) \cap extr^{set}(\alpha N)$. Thus, $(\mathcal{U} - B_{Int}) \cap extr^{set}(\alpha M) = \varnothing$. Hence, $\mathcal{Z} \cup (\mathcal{U} - B_{Int}) \subseteq \Sigma - extr^{set}(\alpha M)$ and so $(extr_{EP(M)}(s), \mathcal{S}') \in \phi K$ by SF3. Similarly, $(extr_{EP(N)}(u), \mathcal{U}') \in \phi L$. Moreover,

$$\mathcal{S}' \cup \mathcal{U}' = \mathcal{S} \cup \mathcal{Z} \cup (\mathcal{U} - B_{Int}) \cup \mathcal{U} \cup \mathcal{Z} \cup (\mathcal{S} - B_{Int}) = \mathcal{S} \cup \mathcal{U} \cup \mathcal{Z}.$$

Hence, by the definition of parallel composition in chapter 2.4.2, the only thing we need to show is that $\mathcal{S}' - B_{Int} = \mathcal{U}' - B_{Int}$. In other words, that

$$(\mathcal{S} \cup \mathcal{Z} \cup (\mathcal{U} - B_{Int})) - B_{Int} = (\mathcal{U} \cup \mathcal{Z} \cup (\mathcal{S} - B_{Int})) - B_{Int}$$

which is equivalent to

$$(\mathcal{S} - B_{Int}) \cup (\mathcal{Z} - B_{Int}) \cup (\mathcal{U} - B_{Int}) = (\mathcal{U} - B_{Int}) \cup (\mathcal{Z} - B_{Int}) \cup (\mathcal{S} - B_{Int})$$

which clearly holds. □

Lemma B.21. *Assume that M and N both meet Dom-SF-check and Dom-T-check. Assume that* $extr_{EP(M)}(\phi M) \subseteq \phi K$ *and* $extr_{EP(N)}(\phi N) \subseteq \phi L$. *Then* $extr_{EP(O)}(\phi O) \subseteq \phi J$.

Proof. Let $(t, R) \in \phi_{dom_{EP(O)}}O$ be such that $R \subseteq \alpha O$. By SF-DEF2 and SF3, it suffices to show that

$$(extr_{EP(O)}(t), extr^{ref}_{EP(O)}(R, t, O) \cup (\Sigma - extr^{set}(\alpha O)) \in \phi J. \qquad (**)$$

By proposition B.17, there exist $(s, S) \in \phi M$ and $(u, U) \in \phi N$ such that:

- $t \in (s \parallel_{A_{Int}} u) \setminus A_{Int}$.
- $S \subseteq \alpha M$ and $U \subseteq \alpha N$.
- $R \cup A_{Int} = (S \cup U) \cup Z$, where $Z \subseteq (\Sigma - (\alpha M \cup \alpha N))$.

Moreover, by lemma B.19, $(s, S) \in \phi_{dom_{EP(M)}}M$ and $(u, U) \in \phi_{dom_{EP(N)}}N$, and so, by proposition B.14, $(s, S) \in \phi_{Dom_{EP(M)}}M$ and $(u, U) \in \phi_{Dom_{EP(N)}}N$. We first show that

$$extr^{ref}_{EP(O)}(R, t, O) \subseteq extr^{ref}_{EP(M)}(S, s, M) \cup extr^{ref}_{EP(N)}(U, u, N). \qquad \text{(B.6)}$$

Let $ep_i \in EP(O)$. Wlog and by proposition 4.5, we assume that $ep_i \in EP(M) - EP(N)$. Thus, by SF-GLOBAL2 and since $t \in dom_{EP(O)} \subseteq Dom_{EP(O)}$ by proposition B.14, it suffices to show that

$$extr^{ref}_i(R \cap A_i, t\lceil A_i, O) = extr^{ref}_i(S \cap A_i, s\lceil A_i, M).$$

We first show that $R \cap A_i = S \cap A_i$ and $t\lceil A_i = s\lceil A_i$ before considering each of two cases in turn. We observe the following:

- $A_i \cap A_{Int} = \varnothing$ by proposition B.1(1) and EP-UNI1.
- By definition 4.5, $A_i \subseteq \alpha M$ and, also by EP-UNI1, $A_i \cap \alpha N = \varnothing$.
- $A_i \cap events(u) = \varnothing$ by SF2, PA1 and since $A_i \cap \alpha N = \varnothing$.

Since $A_i \cap events(u) = \varnothing$ and $A_i \cap A_{Int} = \varnothing$, $t\lceil A_i = s\lceil A_i$. Moreover, since $A_i \cap A_{Int} = \varnothing$, $A_i \subseteq \alpha M$ and $A_i \cap \alpha N = \varnothing$, then

$$R \cap A_i = (R \cup A_{Int}) \cap A_i = (S \cup U \cup Z) \cap A_i = S \cap A_i.$$

Case 1: $A_i \subseteq Fvis$. In this case, by EP5-FVI,

$$extr^{ref}_i(R \cap A_i, t\lceil A_i, O) = R \cap A_i = S \cap A_i = extr^{ref}_i(S \cap A_i, s\lceil A_i, M).$$

Case 2: $A_i \cap Fvis = \varnothing$. By definition 4.6, $Comm(A_i, O) = Comm(A_i, M)$. Thus, by definition 4.10,

$$extr^{ref}_i(R \cap A_i, t\lceil A_i, O) = extr^{ref}_i(S \cap A_i, s\lceil A_i, M).$$

Hence, we have shown (B.6).

We now proceed with the remainder of the proof. By lemma B.20 and since $(s, S) \in \phi_{dom_{EP(M)}}M$ and $(u, U) \in \phi_{dom_{EP(N)}}N$, then $(w, Y) \in \phi H$ such that:

- $w \in (extr_{EP(M)}(s) \parallel_{B_{Int}} extr_{EP(N)}(u))$.
- $Y = extr^{ref}_{EP(M)}(S, s, M) \cup extr^{ref}_{EP(N)}(U, u, N) \cup (\Sigma - extr^{set}(\alpha M \cup \alpha N))$.

Since $A_{Int} = \alpha M \cap \alpha N$ and $R \cup A_{Int} = (S \cup U) \cup Z$, where $Z \subseteq (\Sigma - (\alpha M \cup \alpha N))$, then $A_{Int} \subseteq S \cup U$. Thus, by lemma B.16,

$$B_{Int} \subseteq extr^{ref}_{EP(M)}(S, s, M) \cup extr^{ref}_{EP(N)}(U, u, N)$$

and so $(w \setminus B_{Int}, Y) \in \phi J$ for $w \in (extr_{EP(M)}(s) \parallel_{B_{Int}} extr_{EP(N)}(u))$. Thus, by SF2, propositions B.7 and B.8, and since $s \in Dom_{EP(M)}$ and $u \in Dom_{EP(N)}$,

$$extr_{EP(O)}(t) \in (extr_{EP(M)}(s) \parallel_{B_{Int}} extr_{EP(N)}(u)) \setminus B_{Int}$$

and so

$$(extr_{EP(O)}(t), Y) \in \phi J.$$

Hence, by (B.6), (**) and SF3, it remains to show that $(\Sigma - extr^{set}(\alpha O)) \subseteq Y$. We know that $B_{Int} \subseteq Y$ and so we show that $(\Sigma - extr^{set}(\alpha O)) - B_{Int} \subseteq Y$. This follows by the fact that, due to definitions 4.2 and 4.5 and propositions 4.5 and B.1(2),

$$(\Sigma - extr^{set}(\alpha O)) - B_{Int} = \Sigma - (extr^{set}(\alpha O) \cup B_{Int}) = \Sigma - extr^{set}(\alpha M \cup \alpha N).$$

□

Lemma B.22. *If $M \sqsupseteq^{EP(M)}_{SF} K$ and $N \sqsupseteq^{EP(N)}_{SF} L$, then $O \sqsupseteq^{EP(O)}_{SF} J$.*

Proof. We assume that $M \sqsupseteq^{EP(M)}_{SF} K$ and $N \sqsupseteq^{EP(N)}_{SF} L$. Hence, by SF-DEF3, both M and N meet Dom-T-check and Dom-SF-check. Thus, by propositions B.11 and B.18, O meets conditions Dom-T-check and Dom-SF-check. By SF-DEF3 and SF-DEF1, $extr_{EP(M)}(\tau M) \subseteq \tau K$ and $extr_{EP(N)}(\tau N) \subseteq \tau L$ and so, by TR-DEF2, $M \sqsupseteq^{EP(M)}_{T} K$ and $N \sqsupseteq^{EP(N)}_{T} L$. Thus, by proposition B.12, $extr_{EP(O)}(\tau O) \subseteq \tau J$. Also by SF-DEF3 and SF-DEF1, we observe that $extr_{EP(M)}(\phi M) \subseteq \phi K$ and $extr_{EP(N)}(\phi N) \subseteq \phi L$. Thus, by lemma B.21, $extr_{EP(O)}(\phi O) \subseteq \phi J$ and this concludes the proof by SF-DEF3 and SF-DEF1. □

Proof of theorem 4.10

Proof. We assume that $\alpha F_{impl}(Q_1, Q_2, \ldots, Q_n) \subseteq Fvis$ and $Q_i \sqsupseteq^{EP(Q_i)}_{SF} P_i$ for $1 \leq i \leq n$. Let $Q = F_{impl}(Q_1, Q_2, \ldots, Q_n)$ and $P = F_{spec}(P_1, P_2, \ldots, P_n)$. By induction on n using lemma B.22, $Q \sqsupseteq^{EP(Q)}_{SF} P$. Hence, by theorem 4.9 and since $\alpha Q \subseteq Fvis$, $Q \sqsupseteq_{SF} P$. □

B.5 Proofs from section 4.5

Results used in the proof of theorem 4.12

Lemma B.23. *Let Q be an implementation process such that $\alpha Q \subseteq Fvis$. Then:*

1. $extr_{EP(Q)}(\delta Q) = \delta Q$.

2. $extr_{EP(Q)}(\phi_{\perp} Q) = \phi_{\perp} Q$.

Proof. 1. Let $t \in min\delta Q$. By MD, $t \in \tau Q$ and so, by proposition B.4(1,2), $t \in Dom_{EP(Q)}$ and $extr_{EP(Q)}(t) = t$. Thus, by FD-DEF2, FD4 and definition 2.2,

$$extr_{EP(Q)}(\delta Q) = \{t \circ u \mid t \in min\delta Q \ \wedge \ u \in \Sigma^*\} = \delta Q.$$

2. By proposition B.15, $extr_{EP(Q)}(\phi Q) = \phi Q$ and, by part1 of the lemma, $extr_{EP(Q)}(\delta Q) = \delta Q$. Thus, by FD-DEF3 and DR2,

$$extr_{EP(Q)}(\phi_{\perp} Q) = \phi Q \cup \{(t, R) \mid t \in \delta Q \ \wedge \ R \subseteq \Sigma\} = \phi_{\perp} Q.$$

□

Proof of theorem 4.11

Proof. ($\Longrightarrow$) We assume that $Q \sqsupseteq_{FD}^{EP(Q)} P$. Hence, by FD-DEF4 and FD-DEF1, $extr_{EP(Q)}(\phi_{\perp} Q) \subseteq \phi_{\perp} P$ and $extr_{EP(Q)}(\delta Q) \subseteq \delta P$. Thus, $\phi_{\perp} Q \subseteq \phi_{\perp} P$ and $\delta Q \subseteq \delta P$ by lemma B.23.

($\Longleftarrow$) We assume that $Q \sqsupseteq_{FD} P$. Thus, by lemma B.23, $extr_{EP(Q)}(\phi_{\perp} Q) \subseteq \phi_{\perp} P$ and $extr_{EP(Q)}(\delta Q) \subseteq \delta P$. By proposition B.4(1), $t \in Dom_{EP(Q)}$ for all $t \in \tau Q$ and so Q meets Dom-T-check. Q meets Dom-SF-check since, by definition 4.5, $A_i \subseteq Fvis$ for every $ep_i \in EP(Q)$. Thus, by FD-DEF1 and FD-DEF4, $Q \sqsupseteq_{FD}^{EP(Q)} P$. □

Lemma B.24. *Let Q be an implementation process and $\ldots, t_j, \ldots$ an ω-sequence in $Dom_{EP(Q)}$. Then $\ldots, extr_{EP(Q)}(t_j), \ldots$ is also an ω-sequence.*

Proof. Let $w \in \Sigma^\omega$ be the least upper bound of the sequence $\ldots, t_j, \ldots$. Thus, by TR-GLOBAL1, there exists $ep_i \in EP(Q)$ such that $w \lceil A_i \in \Sigma^\omega$. Hence, and also by TR-GLOBAL1, $\ldots, t_j \lceil A_i, \ldots$ is an ω-sequence in Dom_i and so $\ldots, extr_i(t_j \lceil A_i), \ldots$ is an ω-sequence by EP6. By induction on the length of traces using TR-GLOBAL2, $extr_{EP(Q)}(t_j) \in |||_{1 \leq k \leq m} extr_k(t_j \lceil A_k)$ for each t_j. Thus, the length of the $extr_{EP(Q)}(t_j)$ increases unboundedly and so, by proposition B.3(2), $\ldots, extr_{EP(Q)}(t_j), \ldots$ is an ω-sequence. □

Proposition B.25. *Let Q be an implementation process and P a process such that $Q \sqsupseteq_{FD}^{EP(Q)} P$. If $t \in \tau_{Dom_{EP(Q)}} Q$, then $extr_{EP(Q)}(t) \in \tau_{\bot} P$.*

Proof. By FD-DEF4 and FD-DEF1, we observe that $extr_{EP(Q)}(\delta Q) \subseteq \delta P$ and $extr_{EP(Q)}(\phi_{\bot} Q) \subseteq \phi_{\bot} P$. Moreover, by FD-DEF3, $extr_{EP(Q)}(\phi Q) \subseteq \phi_{\bot} P$. Let $t \in \tau_{Dom_{EP(Q)}} Q$ and so $t \in \tau Q$. We now prove two auxiliary results.

$$\text{If } t \in dom_{EP(Q)} \text{ then } extr_{EP(Q)}(t) \in \tau_{\bot} P. \tag{B.7}$$

We assume that $t \in dom_{EP(Q)}$ and consider each of two cases in turn.

Case 1: $t \in \delta Q$. In this case, there exists $v \in min\delta Q$ such that $v \leq t$ and so, by proposition B.3(1), such that $v \in \tau_{Dom_{EP(Q)}} Q$. Thus, $extr_{EP(Q)}(v) \in \delta P$ by FD-DEF2 and since $extr_{EP(Q)}(\delta Q) \subseteq \delta P$. Hence, by proposition B.3(2) and FD4, $extr_{EP(Q)}(t) \in \delta P$ and so $extr_{EP(Q)}(t) \in \tau_{\bot} P$ by DR1.

Case 2: $t \notin \delta Q$ and so $(t, \varnothing) \in \phi Q$ by DR3. Since $extr_{EP(Q)}(\phi Q) \subseteq \phi_{\bot} P$ and by SF-DEF2, $(extr_{EP(Q)}(t), \varnothing) \in \phi_{\bot} P$. Thus, $extr_{EP(Q)}(t) \in \tau_{\bot} P$ by definition.

Hence, we have shown (B.7). We now show the following.

$$\text{Either } extr_{EP(Q)}(t) \in \tau_{\bot} P \text{ or there exists } a \text{ such that } t \circ \langle a \rangle \in \tau_{Dom_{EP(Q)}} Q. \tag{B.8}$$

We consider each of two cases in turn.

Case 1: $t \in \delta Q$. In this case, there exists $v \in min\delta Q$ such that $v \leq t$ and so, by proposition B.3(1), such that $v \in \tau_{Dom_{EP(Q)}} Q$. Thus, $extr_{EP(Q)}(v) \in \delta P$ by FD-DEF2 and since $extr_{EP(Q)}(\delta Q) \subseteq \delta P$. Hence, by proposition B.3(2) and FD4, $extr_{EP(Q)}(t) \in \delta P$ and so $extr_{EP(Q)}(t) \in \tau_{\bot} P$ by DR1.

Case 2: $t \notin \delta Q$ and so $(t, \varnothing) \in \phi Q$ by DR3. In the event that $t \in dom_{EP(Q)}$ then the proof is immediate by (B.7) and so we assume that $t \notin dom_{EP(Q)}$. Hence, by SF-GLOBAL1, let $ep_i \in EP(Q)$ be such that $t\lceil A_i \notin dom_i$. By EP1 we observe that $A_i \cap Fvis = \varnothing$ since otherwise $t\lceil A_i \in dom_i$ by EP3-FVI and EP3A-FVI. Let $(t, R) \in \phi Q$ be refusal-maximal; by SF3 and since $t \in Dom_{EP(Q)}$, $(t, R \cap \alpha Q) \in \phi_{Dom_{EP(Q)}} Q$. By definition 4.5, $A_i \subseteq \alpha Q$. Thus, since $t\lceil A_i \notin dom_i$ and since, by FD-DEF4, Q meets Dom-SF-check,

$$extr_i^{ref}(R \cap A_i, t\lceil A_i, Q) = extr_i^{ref}((R \cap \alpha Q) \cap A_i, t\lceil A_i, Q) = \varnothing.$$

The proof of (B.8) concludes by considering each of two sub-cases in turn.

Case 2a: $Comm(A_i, Q) = Left$. In this case, $R \cap A_i \in ref_i(t\lceil A_i)$ by definition 4.10. Let $X \in ref_i(t\lceil A_i)$ be maximal in the subset-ordering and such that $R \cap A_i \subseteq X$ (if there is more than one such set we choose one arbitrarily.) Thus, by EP5, there exists $a \in A_i$ such that $a \notin X$ and so, also by EP5, $t\lceil A_i \circ \langle a \rangle \in Dom_i$. Hence, $t \circ \langle a \rangle \in Dom_{EP(Q)}$ by TR-GLOBAL1, EP-UNI1 and since $t \in Dom_{EP(Q)}$. Moreover, $a \notin R \cap A_i$ and so, since (t, R) is refusal-maximal, $t \circ \langle a \rangle \in \tau Q$ by SF4.

Case 2b: $Comm(A_i, Q) = Right$. In this case, by definition 4.10, $R \cap A_i \in \overline{ref}_i(t\lceil A_i)$. Thus, by definition 4.9, there does not exist $X \in ref_i(t\lceil A_i)$ such that $(R \cap A_i) \cup X = A_i$. Let $W \in ref_i(t\lceil A_i)$ be maximal in the subset-ordering. Thus, there exists $a \in A_i$ such that $a \notin W$ and $a \notin R \cap A_i$. Hence, by EP5, $t\lceil A_i \circ \langle a \rangle \in Dom_i$ and so $t \circ \langle a \rangle \in Dom_{EP(Q)}$ by TR-GLOBAL1, EP-UNI1 and since $t \in Dom_{EP(Q)}$. Moreover, since $a \notin R \cap A_i$ and (t, R) is refusal-maximal, $t \circ \langle a \rangle \in \tau Q$ by SF4.

Hence, we have proved (B.8). By (B.7), (B.8) and proposition B.14, there exists a trace x such that $extr_{EP(Q)}(t \circ x) \in \tau_\perp P$. The proof then follows by the monotonicity of $extr_{EP(Q)}$ over traces due to proposition B.3(2) and the prefix-closure of $\tau_\perp P$ by FD1. □

Lemma B.26. *Assume that $extr_{EP(M)}(\phi M) \subseteq \phi_\perp K$ and $extr_{EP(N)}(\phi N) \subseteq \phi_\perp L$. Let $(s, S) \in \phi_{dom_{EP(M)}} M$ and $(u, U) \in \phi_{dom_{EP(N)}} N$ be such that $S \subseteq \alpha M$ and $U \subseteq \alpha N$. Then $(w, Y) \in \phi_\perp H$ such that*

- $w \in (extr_{EP(M)}(s) \parallel_{B_{Int}} extr_{EP(N)}(u))$.
- $Y = extr^{ref}_{EP(M)}(S, s, M) \cup extr^{ref}_{EP(N)}(U, u, N) \cup (\Sigma - extr^{set}(\alpha M \cup \alpha N))$.

Proof. The proof is the same as that of lemma B.20 except that:

- $\phi_\perp K$ and $\phi_\perp L$ are substituted for ϕK and ϕL.
- FD2 is used in place of SF3.
- The relevant definition of parallel composition is taken from chapter 2.4.3 rather than chapter 2.4.2.

□

Lemma B.27. *Assume that M and N both meet Dom-SF-check and Dom-T-check. Assume that $extr_{EP(M)}(\phi M) \subseteq \phi_\perp K$ and $extr_{EP(N)}(\phi N) \subseteq \phi_\perp L$. Then $extr_{EP(O)}(\phi O) \subseteq \phi_\perp J$.*

Proof. The proof is the same as that of lemma B.21 except that:

- $\phi_\perp H$ and $\phi_\perp J$ are substituted for ϕH and ϕJ respectively.
- FD2 is used in place of SF3.
- Lemma B.26 is used in place of lemma B.20.

□

Lemma B.28. *Let $s \in \tau_\perp K$ and $u \in \tau_\perp L$. Then $(s \parallel_{B_{Int}} u) \subseteq \tau_\perp H$.*

Proof. By definition of $\tau_\perp$, $(s, \varnothing) \in \phi_\perp K$ and $(u, \varnothing) \in \phi_\perp L$. Thus,

$$\{(w, \varnothing) \mid w \in (s \parallel_{B_{Int}} u)\} \in \phi_\perp H$$

and so $(s \parallel_{B_{Int}} u) \subseteq \tau_\perp H$. □

Lemma B.29. *If $M \sqsupseteq_{FD}^{EP(M)} K$ and $N \sqsupseteq_{FD}^{EP(N)} L$, then $O \sqsupseteq_{FD}^{EP(O)} J$.*

Proof. We assume that $M \sqsupseteq_{FD}^{EP(M)} K$ and $N \sqsupseteq_{FD}^{EP(N)} L$. Hence, by FD-DEF4, both M and N meet Dom-T-check and Dom-SF-check. Thus, by propositions B.11 and B.18, O meets conditions Dom-T-check and Dom-SF-check. By FD-DEF4 and FD-DEF1, we therefore have to show that $extr_{EP(O)}(\delta O) \subseteq \delta J$ and $extr_{EP(O)}(\phi_\perp O) \subseteq \phi_\perp J$.

We first show that $extr_{EP(O)}(\delta O) \subseteq \delta J$. Let $t \in Dom_{EP(O)}$ and $t \in min\delta O$. By FD-DEF2 and FD4, it suffices to show that $extr_{EP(O)}(t) \in \delta J$. According to the semantics of the hiding operator in the failures divergences model, we consider each of two cases in turn. Before we proceed, recall that $O = I \setminus A_{Int}$ and note that, by FD-DEF4 and FD-DEF1, $extr_{EP(M)}(\delta M) \subseteq \delta K$ and $extr_{EP(N)}(\delta N) \subseteq \delta L$.

Case 1: There exists $w \in \delta I$ such that $t = w \setminus A_{Int} \circ y$, for some trace y. In fact, $t = w \setminus A_{Int}$ since $t \in min\delta O$. Moreover, there exists $v \in min\delta I$ such that $v \leq w$ and $v \setminus A_{Int} = w \setminus A_{Int}$ since otherwise $t \notin min\delta O$. We therefore take $t = v \setminus A_{Int}$. Since $v \in min\delta I \subseteq \delta I$, there exist $s \in \tau_\perp M$, $u \in \tau_\perp N$ such that $v \in (s \parallel_{A_{Int}} u)$ and $s \in \delta M$ or $u \in \delta N$. If $s \in \delta M$, then $s \in min\delta M$ since otherwise there exists $x < v$ such that $x \in \delta I$ and so $v \notin min\delta I$. Similarly, if $u \in \delta N$ then $u \in min\delta N$. By MD, $min\delta M \subseteq \tau M$ and $min\delta N \subseteq \tau N$; moreover, $\tau_\perp M = \tau M \cup \delta M$ and $\tau_\perp N = \tau N \cup \delta N$ by DR1. Thus $s \in \tau M$, $u \in \tau N$ and either $s \in min\delta M$ or $u \in min\delta N$. Also by MD, $t \in min\delta O \subseteq \tau O$ and so, by proposition B.9 and since $t \in Dom_{EP(O)}$, $s \in Dom_{EP(M)}$ and $u \in Dom_{EP(N)}$. Wlog, we assume that $s \in min\delta M$. Thus, by FD-DEF2 and since $extr_{EP(M)}(\delta M) \subseteq \delta K$, $extr_{EP(M)}(s) \in \delta K$ and so $extr_{EP(M)}(s) \in \tau_\perp K$ by DR1. Since $u \in \tau N$ and $u \in Dom_{EP(N)}$, $extr_{EP(N)}(u) \in \tau_\perp L$ by proposition B.25. Hence, $(extr_{EP(M)}(s) \parallel_{B_{Int}} extr_{EP(N)}(u)) \subseteq \delta H$ and so, by propositions B.7 and B.8,

$$extr_{EP(O)}(t) \in (extr_{EP(M)}(s) \parallel_{B_{Int}} extr_{EP(N)}(u)) \setminus B_{Int} \subseteq \delta J.$$

Case 2: There exists $w \in \Sigma^\omega$ such that $t = w \setminus A_{Int} \circ x$ for some trace x and, for every $y < w$,

$$y \in \tau_\perp I = \tau I \cup \delta I$$

by DR1. Since $t \in min\delta O$, $x = \langle\rangle$ and so $t = w \setminus A_{Int}$. In the event that there exists $y' < w$ such that $y' \in \delta I$, then $t = y' \setminus A_{Int}$ and the proof proceeds as for Case 1. (We know $t = y' \setminus A_{Int}$ in such a case since otherwise $t \notin min\delta O$.) We therefore assume that $y \notin \delta I$ and so $y \in \tau I$ for every $y < w$. Thus, for every $y < w$, $y \in (p_y \parallel_{A_{Int}} q_y)$ such that $p_y \in \tau M$ and $q_y \in \tau N$. Let $t_y = y \setminus A_{Int}$ for $y < w$. Then $t_y \leq t$ and so $t_y \in \tau O$ for $y < w$; moreover, $t_y \in Dom_{EP(O)}$ by proposition B.14. Thus, for $y < w$, $p_y \in Dom_{EP(M)}$ and $q_y \in Dom_{EP(N)}$ by proposition B.9 and so, by proposition B.25, $extr_{EP(M)}(p_y) \in \tau_\perp K$ and $extr_{EP(N)}(q_y) \in \tau_\perp L$. Hence, by proposition B.7 and lemma B.28, $y \in Dom_{EP(I)}$ and $extr_{EP(I)}(y) \in \tau_\perp H$ for every $y < w$. Hence, by lemma B.24, there exists $k \in \Sigma^\omega$ such that k is the least upper bound of the sequence of $extr_{EP(I)}(y)$ for $y < w$. Moreover, by FD1, $l \in \tau_\perp H$ for every $l < k$. Since $w \in \Sigma^\omega$ and $w \setminus A_{Int}$ is finite, then $w = r \circ s$ where $r \in \Sigma^*$, $s \in (A_{Int})^\omega$ and so $r \setminus A_{Int} = w \setminus A_{Int} = t$. Thus, by proposition B.8, for any trace h such that $r \leq h < w$,

$$extr_{EP(O)}(t) = extr_{EP(O)}(w \setminus A_{Int}) = extr_{EP(O)}(h \setminus A_{Int}) = extr_{EP(I)}(h) \setminus B_{Int}.$$

Hence, $k \setminus B_{Int} = extr_{EP(O)}(t)$ and so $extr_{EP(O)}(t) \in \delta J$.

We now show that $extr_{EP(O)}(\phi_\perp O) \subseteq \phi_\perp J$. By FD-DEF3,

$$extr_{EP(O)}(\phi_\perp O) = extr_{EP(O)}(\phi O) \cup \{(t, R) \mid t \in extr_{EP(O)}(\delta O) \ \wedge \ R \subseteq \Sigma\}.$$

Thus, since $extr_{EP(O)}(\delta O) \subseteq \delta J$ and by FD5, it is sufficient to show that $extr_{EP(O)}(\phi O) \subseteq \phi_\perp J$. By FD-DEF1, FD-DEF3 and FD-DEF4, we observe that $extr_{EP(M)}(\phi M) \subseteq \phi_\perp K$ and $extr_{EP(N)}(\phi N) \subseteq \phi_\perp L$. Hence, by lemma B.27, $extr_{EP(O)}(\phi O) \subseteq \phi_\perp J$. □

Proof of theorem 4.12

Proof. We assume that $\alpha F_{impl}(Q_1, Q_2, \ldots, Q_n) \subseteq Fvis$ and $Q_i \sqsupseteq_{FD}^{EP(Q_i)} P_i$ for $1 \leq i \leq n$. Let $Q = F_{impl}(Q_1, Q_2, \ldots, Q_n)$ and $P = F_{spec}(P_1, P_2, \ldots, P_n)$. By induction on n using lemma B.29, $Q \sqsupseteq_{FD}^{EP(Q)} P$. Hence, by theorem 4.11 and since $\alpha Q \subseteq Fvis$, $Q \sqsupseteq_{FD} P$. □

Appendix C

Proofs from chapter 6

C.1 Proofs from sections 6.2 and 6.3

Proposition C.1. *The following holds:*

$$\tau Q_{Proj} = \{t \in \tau Q \mid t\lceil Proj_{EP(Q)} \in Dom_{EP(Q)}\lceil Proj_{EP(Q)}\}.$$

Proof. Let $i \in inv$. By definition 6.4 and since $\langle\rangle \in Dom_i$ by EP3-T, $\tau D_i = Dom_i$. Also by EP3-T, $Dom_i \subseteq (A_i)^*$ and so:

$$\tau DC_i = \{t\backslash(A_i - Proj_i) \mid t \in Dom_i\} = \{t\lceil Proj_i \mid t \in Dom_i\} = Dom_i\lceil Proj_i.$$

By definition 4.7(1), $Proj_j = \varnothing$ and so $Dom_j\lceil Proj_j = \{\langle\rangle\}$ for $j \notin inv$. Thus, $\tau DC = |||_{1\leq i\leq m} Dom_i\lceil Proj_i$. We then observe the following.

- $Dom_{EP(Q)} = |||_{1\leq i\leq m} Dom_i$ by TR-GLOBAL1.
- $Proj_{EP(Q)} = \bigcup_{1\leq i\leq m} Proj_i$ by definition 4.7(3).
- Let $ep_i \in EP(Q)$. Then the following hold.
  - $events(t) \subseteq A_i$ for every $t \in Dom_i$ by EP3-T.
  - $Proj_i \subseteq A_i$ by definition 4.7 and EP2.
  - $A_i \cap A_j = \varnothing$ for $ep_j \in EP(Q)$ such that $i \neq j$ by EP-UNI1.

Hence, $\tau DC = Dom_{EP(Q)}\lceil Proj_{EP(Q)}$. We then observe that $\beta(D_i) \subseteq A_i$ for $i \in inv$ by EP3-T and definition 6.4, and so $\beta(DC_i) \subseteq Proj_i$ since $Proj_i \subseteq A_i$. Thus, and by definition 4.7, $\beta(DC) \subseteq Proj_{EP(Q)}$ and so we take $\alpha DC = Proj_{EP(Q)}$. By EP2 and definitions 4.5 and 4.7, $Proj_{EP(Q)} \subseteq \alpha Q$ and so $Proj_{EP(Q)} = \alpha Q \cap \alpha DC$. Hence, the proof follows by theorem 2.17. □

Proof of theorem 6.1

Proof. ($\Longrightarrow$) We assume that $Q_{Proj} \setminus Fvis \sqsupseteq_T (|||_{i \in inv} D_i)$. Let $t \in \tau Q$ be such that $t\lceil Proj_{EP(Q)} \in Dom_{EP(Q)}\lceil Proj_{EP(Q)}$. By Dom-T-check, it suffices to show that $t \in Dom_{EP(Q)}$. By proposition C.1, $t \in \tau Q_{Proj}$ and so

$$t \setminus Fvis \in \tau(|||_{i \in inv} D_i) = |||_{i \in inv} Dom_i.$$

Let $i \in inv$. By definition 6.1, $A_i \cap Fvis = \varnothing$. Thus, and by EP3-T and EP-UNI1,

$$t\lceil A_i = (t \setminus Fvis)\lceil A_i \in Dom_i.$$

Moreover, $t\lceil A_j \in Dom_j$ for $j \notin inv$ by EP3-FVI and so $t \in Dom_{EP(Q)}$ by TR-GLOBAL1.

($\Longleftarrow$) We assume that Q meets Dom-T-check. Let $w \in \tau(Q_{Proj} \setminus Fvis)$ be such that $w = t \setminus Fvis$ where $t \in \tau Q_{Proj}$. By proposition C.1 and since Q meets Dom-T-check, $t \in Dom_{EP(Q)}$. Let $j \in inv$. By definition 6.1, $A_j \cap Fvis = \varnothing$. Hence, and by TR-GLOBAL1,

$$w\lceil A_j = (t \setminus Fvis)\lceil A_j = t\lceil A_j \in Dom_j = \tau D_j.$$

Since $t \in Dom_{EP(Q)}$, then $events(t) \subseteq \bigcup_{1 \le i \le m} A_i$ by TR-GLOBAL1 and so

$$events(w) \subseteq (\bigcup_{1 \le i \le m} A_i) - Fvis.$$

Thus, by definition 6.1 and EP1, $events(w) \subseteq \bigcup_{i \in inv} A_i$ and so, by EP-UNI1, $w \in \tau(|||_{i \in inv} D_i)$. □

Proof of theorem 6.2

Proof. By theorem 6.1, it suffices to show that $Q_{Proj} \setminus Fvis \sqsupseteq_T (|||_{i \in inv} D_i)$ if and only if $Q_{Proj} \setminus (\alpha Q - A_i) \sqsupseteq_T D_i$ for every $i \in inv$.

($\Longrightarrow$) We assume that $Q_{Proj} \setminus Fvis \sqsupseteq_T (|||_{i \in inv} D_i)$. Let $i \in inv$ and $t \in \tau(Q_{Proj} \setminus (\alpha Q - A_i))$, where $w \in \tau Q_{Proj}$ is such that $t = w \setminus (\alpha Q - A_i)$. By proposition C.1 and PA1, $events(w) \subseteq \alpha Q$ and so $t = w\lceil A_i$. We also observe that $w \setminus Fvis \in \tau(|||_{j \in inv} D_j)$ and, by definition 6.1, $A_i \cap Fvis = \varnothing$. Thus, and by EP3-T and EP-UNI1, $t = w\lceil A_i = (w \setminus Fvis)\lceil A_i \in \tau D_i$.

($\Longleftarrow$) We assume that $Q_{Proj} \setminus (\alpha Q - A_i) \sqsupseteq_T D_i$ for every $i \in inv$. Let $t \in \tau(Q_{Proj} \setminus Fvis)$ be such that $w \in \tau Q_{Proj}$ and $t = w \setminus Fvis$. Let $j \in inv$. By proposition C.1 and PA1, $events(w) \subseteq \alpha Q$. Thus, $w \setminus (\alpha Q - A_j) = w\lceil A_j$ and so $w\lceil A_j \in \tau D_j$. By definition 6.1, $A_j \cap Fvis = \varnothing$. Hence,

$$t\lceil A_j = (w \setminus Fvis)\lceil A_j = w\lceil A_j \in \tau D_j.$$

Since $events(w) \subseteq \alpha Q$, and by definition 4.5, $events(w) \subseteq \bigcup_{1 \leq i \leq m} A_i$. Thus,

$$events(t) \subseteq (\bigcup_{1 \leq i \leq m} A_i) - Fvis$$

and so, by definition 6.1 and EP1, $events(t) \subseteq \bigcup_{i \in inv} A_i$. Hence, by EP-UNI1, $t \in \tau(|||_{i \in inv} D_i)$. □

Lemma C.2. *Let $s \in \tau Q$, $u \in \tau(|||_{i \in inv} D_i)$ and $t \in (s \parallel_{A_{inv}} u)$. Then $t = s$, $t \in Dom_{EP(Q)}$ and $t \lceil A_{inv} = u$.*

Proof. We first observe that, for $i \in inv$, $\beta(D_i) \subseteq A_i$ by EP3-T and definition 6.4, and so $\beta(|||_{i \in inv} D_i) \subseteq A_{inv}$ by definition 6.2(1). Thus, we assume $\alpha(|||_{i \in inv} D_i) = A_{inv}$ and so $events(u) \subseteq A_{inv}$ by PA1. Hence, $t = s$ and, by TRP, $t \lceil A_{inv} = u \in \tau(|||_{i \in inv} D_i)$. Thus, by EP-UNI1 and EP3-T, $t \lceil A_i \in \tau D_i = Dom_i$ for $i \in inv$. Moreover, $t \lceil A_j \in Dom_j$ for $j \notin inv$ by EP3-FVI. Thus, $t \in Dom_{EP(Q)}$ by TR-GLOBAL1. □

Lemma C.3. *Let $t \in Dom_{EP(Q)}$. Then $t \lceil A_{inv} \in \tau(|||_{i \in inv} D_i)$.*

Proof. By TR-GLOBAL1, $t \lceil A_i \in Dom_i = \tau D_i$ for $i \in inv$. Thus, $t \lceil A_{inv} \in \tau(|||_{i \in inv} D_i)$ by definition 6.2(1) and EP-UNI1. □

Proof of proposition 6.4

Proof. We first observe that, for $i \in inv$, $\beta(D_i) \subseteq A_i$ by EP3-T and definition 6.4, and so $\beta(|||_{i \in inv} D_i) \subseteq A_{inv}$ by definition 6.2(1). Thus, we assume $\alpha D_i = A_i$ for $i \in inv$ and $\alpha(|||_{i \in inv} D_i) = A_{inv}$. Hence, by definitions 4.5 and 6.2(1), $\alpha(|||_{i \in inv} D_i) = A_{inv} \subseteq \alpha Q$ and so $A_{inv} = \alpha Q \cap \alpha(|||_{i \in inv} D_i)$. We now proceed with the proof proper.

1. We observe that

$$\tau \widehat{Q} = \{t \mid (\exists s \in \tau Q, u \in \tau(|||_{i \in inv} D_i))\ t \in (s \parallel_{A_{inv}} u)\}.$$

Thus, by lemmas C.2 and C.3,

$$\tau \widehat{Q} = \{t \in \tau Q \mid t \in Dom_{EP(Q)}\} = \tau_{Dom_{EP(Q)}} Q.$$

2. Let $i \in inv$. We observe that

$$\phi D_i = \{(t, R) \mid t \in Dom_i = \tau D_i \ \wedge \ R \subseteq (A_i - Next_i(t)) \cup (\Sigma - A_i)\}.$$

Let $t_i \in Dom_i$ for $i \in inv$. By EP-UNI1, since $Next_i(t_i) \subseteq A_i$ for $i \in inv$ and by definition 6.2(1), then:

$$\bigcup_{i \in inv} (A_i - Next_i(t_i)) = A_{inv} - \bigcup_{i \in inv} Next_i(t_i).$$

Hence, by definitions 6.2(1), 6.3(1) and 6.5,

$$\begin{aligned}\phi(|||_{i\in inv}D_i) &= \{(t,R) \mid t \in Dom_{inv} = \tau(|||_{i\in inv}D_i) \wedge \\ &\qquad R \subseteq (A_{inv} - Next_{inv}(t)) \cup (\Sigma - A_{inv})\}.\end{aligned} \tag{C.1}$$

Recall that $A_{inv} = \alpha(|||_{i\in inv}D_i) \subseteq \alpha Q$ and $A_{inv} = \alpha Q \cap \alpha(|||_{i\in inv}D_i)$. Thus, by theorem 2.20,

$$\begin{aligned}\phi\widehat{Q} &= \{(t, S \cup U \cup Z) \mid Z \subseteq (\Sigma - \alpha Q) \wedge \\ &\qquad ((\exists(s,S) \in \phi Q, (u,U) \in \phi(|||_{i\in inv}D_i))\ t \in (s\ ||_{A_{inv}}\ u) \wedge \\ &\qquad S \subseteq \alpha Q\ \wedge\ U \subseteq A_{inv})\}.\end{aligned}$$

and so, by PA2 and SF3,

$$\begin{aligned}\phi\widehat{Q} &= \{(t, S \cup U) \mid (\exists(s,S) \in \phi Q, (u,U) \in \phi(|||_{i\in inv}D_i)) \\ &\qquad t \in (s\ ||_{A_{inv}}\ u)\ \wedge\ U \subseteq A_{inv}\}.\end{aligned}$$

The proof follows by this, (C.1), lemmas C.2 and C.3 and SF2.

3. We observe that D_i is guarded for $i \in inv$ and so, by DF, $\delta D_i = \varnothing$. Hence, $\delta(|||_{i\in inv}D_i) = \varnothing$ and so $\tau_\perp(|||_{i\in inv}D_i) = \tau(|||_{i\in inv}D_i)$ by DR1. Moreover, $\delta Q \subseteq \tau_\perp Q$ also by DR1. As a result,

$$\delta\widehat{Q} = \{t \circ v \mid (\exists s \in \delta Q, u \in \tau(|||_{i\in inv}D_i))\ t \in (s\ ||_{A_{inv}}\ u)\ \wedge\ v \in \Sigma^*\}$$

and so, by definition 2.2,

$$min\delta\widehat{Q} = \{t \mid (\exists s \in min\delta Q, u \in \tau(|||_{i\in inv}D_i))\ t \in (s\ ||_{A_{inv}}\ u)\}.$$

Hence, by lemmas C.2 and C.3, and since $min\delta Q \subseteq \tau Q$ by MD,

$$min\delta\widehat{Q} = \{t \mid t \in min\delta Q\ \wedge\ t \in Dom_{EP(Q)}\}.$$

□

Proof of theorem 6.5

Proof. We first show the following. Let $(t,R) \in \phi_{Dom_{EP(Q)}}Q$ and, by proposition 6.4(2), let $(t,S) \in \phi\widehat{Q}$ be such that $S = R \cup U$, where $U \subseteq (A_{inv} - Next_{inv}(t\lceil A_{inv}))$.

$$\text{Let } ep_i \in EP(Q)\text{. Then } extr_i^{ref}(R \cap A_i, t\lceil A_i, Q) = extr_i^{ref}(S \cap A_i, t\lceil A_i, \widehat{Q}). \tag{C.2}$$

We consider each of two cases in turn.

Case 1: $A_i \subseteq Fvis$ and so $i \notin inv$. By definition 6.2(1) and EP-UNI1, $R \cap A_i = S \cap A_i$ and so the proof follows by EP5-FVI.

Case 2: $A_i \cap Fvis = \varnothing$ and so $i \in inv$. By EP3-T and definition 6.4, $Next_j(t\lceil A_j) \subseteq A_j$ for $j \in inv$. Thus, by definitions 6.2(1) and 6.5 and EP-UNI1,

$$S \cap A_i = (R \cap A_i) \cup U', \text{ where } U' \subseteq (A_i - Next_i(t\lceil A_i)).$$

Hence, by EP5 and definition 6.4, $S \cap A_i \in ref_i(t\lceil A_i)$ if and only if $R \cap A_i \in ref_i(t\lceil A_i)$. Moreover, by EP5 and definitions 4.9 and 6.4, $S \cap A_i \in \overline{ref}_i(t\lceil A_i)$ if and only if $R \cap A_i \in \overline{ref}_i(t\lceil A_i)$. Thus, the proof follows by definition 4.10 and the fact that $Comm(A_i, Q) = Comm(A_i, \widehat{Q})$.

Hence, we have shown (C.2) and now proceed with the proof proper.

1. The proof follows by TR-DEF1 and proposition 6.4(1).

2. The proof follows by definition of Dom-SF-check, (C.2) and propositions 6.3(2,3) and 6.4(2).

3. We first observe that, by proposition B.14, if $(t, R) \in \phi_{dom_{EP(Q)}} Q$ then $(t, R) \in \phi_{Dom_{EP(Q)}} Q$. The proof then follows by SF-DEF2, SF-GLOBAL2, (C.2) and propositions 6.3(2,3) and 6.4(2).

4. The proof follows by FD-DEF2 and proposition 6.4(3).

5. The proof follows by FD-DEF3 and parts 3 and 4 of the theorem.

□

C.2 Proofs from section 6.4

Note: Recall that m gives the cardinality of $EP(Q) = EP(\widehat{Q})$ and so $EP(Q) = \{ep_i \mid 1 \leq i \leq m\}$.

Proposition C.4. *The following results hold, where $i \in inv$:*

1. *If $w \in \tau TE_i$, then $domain(w) \in Dom_i$.*

2. *If $t \in Dom_i$, there exists $w \in \tau TE_i$ such that $domain(w) = t$.*

3. *If $w \in \tau TE_i$, then $extract(w \setminus A_i) = extr_i(domain(w))$.*

Proof. We first observe that $\langle\rangle \in Dom_i$ by EP3-T and $extr_i(\langle\rangle) = \langle\rangle$ by EP4. Moreover, recall that $TE_i \triangleq TE_i(\langle\rangle)$.

(1,2) The proof in both of these cases is by induction on the length of traces using definitions 6.4 and 6.7(1).

(3) The proof is once more by induction on the length of traces, this time using DIS and definitions 6.4, 6.6 and 6.7 (note that $extr_i(domain(w))$ is defined by part (1) of the proposition). □

Proposition C.5. *The following hold:*

1. $\beta(TE_i) \subseteq prep(A_i)$ *for* $i \in inv$.
2. $\beta(TE_{inv}) \subseteq prep(A_{inv})$.
3. $\beta(\widehat{Q}[prep]) \subseteq prep(\alpha\widehat{Q})$.

Proof. By proposition 6.6, $\beta(TE_i) = \{\pi_i(a,t) \mid t \circ \langle a\rangle \in Dom_i\}$ for $i \in inv$. Thus, $\beta(TE_i) \subseteq prep(A_i)$ by EP3-T and definition 6.8 and so $\beta(TE_{inv}) \subseteq prep(A_{inv})$ by definition 6.2(1). Since $\beta(\widehat{Q}) \subseteq \alpha\widehat{Q}$, $\beta(\widehat{Q}[prep]) \subseteq prep(\alpha\widehat{Q})$. □

Lemma C.6. *Let* $w \in \tau TE_{inv}$. *Then* $w \in prep(domain(w))$.

Proof. We proceed by induction on the length of w. In the base case, when $w = \langle\rangle$, the proof is immediate. Let $w = u \circ \langle a\rangle$. By the inductive hypothesis, it suffices to show that $a \in prep(domain(a))$. We assume $\alpha TE_{inv} = \beta(TE_{inv}) = \bigcup_{i\in inv}\beta(TE_i)$. Thus, by proposition 6.6 and PA1, $a = \pi_i(b,x)$ for some $i \in inv$ and $x \circ \langle b\rangle \in Dom_i$. Moreover, note that $b \in A_i$ by EP3-T. Hence, by definition 6.7(1), $domain(a) = b$ and so $a \in prep(domain(a))$ by definition 6.8. □

Proposition C.7. *Let* $u \circ \langle a\rangle \in \tau\widehat{Q}$. *If* $b \in prep(a)$ *then* $domain(b) = a$.

Proof. By proposition 6.3(2), definition 4.5 and PA1, $a \in A_i$ for i such that $1 \le i \le m$. We consider each of two cases in turn.

Case 1: $i \in inv$. By proposition 6.4(1) and TR-GLOBAL1, $u\lceil A_i \circ \langle a\rangle \in Dom_i$. Thus, by EP-UNI1 and definition 6.8, $b = \pi_i(a,x)$ for some trace x such that $x \circ \langle a\rangle \in Dom_i$. Hence, by definition 6.7(1), $domain(b) = a$.

Case 2: $i \notin inv$. By EP-UNI1 and definition 6.8, $prep(a) = \{a\}$ (i.e. $prep(a)$ is not defined explicitly). Thus, it suffices to show that $domain(a) = a$ and we consider two sub-cases in turn.

Case 2a: $domain(a)$ is defined explicitly. We show that this case can never hold by proving a contradiction. In this case, by definition 6.7(1), $a = \pi_j(c,x)$ for some trace x and $j \in inv$ such that $x \circ \langle c\rangle \in Dom_j$. Then, by DIS and definition 6.6, $c = a$. Also, by EP3-T, $c \in A_j$ and so $A_i \cap A_j \neq \varnothing$.

However, $i \neq j$ since $i \notin inv$ and $j \in inv$ and so we have a contradiction by EP-UNI1.

Case 2a: $domain(a)$ is not defined explicitly. Then $domain(a) = a$. □

Proposition C.8. *Let $t \in \tau\widehat{Q}$ and $w \in prep(t)$. Then $domain(w) = t$.*

Proof. We proceed by induction on the length of t. In the base case, when $t = w = \langle\,\rangle$, the proof is immediate. Let $t = u \circ \langle a \rangle$ and $w = v \circ \langle b \rangle$. By the inductive hypothesis, it suffices to show that $domain(b) = a$, which follows by proposition C.7. □

Proposition C.9. $(\widehat{Q}[prep] \parallel_{prep(A_{inv})} TE_{inv})[domain] =_T \widehat{Q}$.

Proof. We assume $\alpha TE_i = \beta(TE_i)$ for $i \in inv$ and, by proposition C.5(2), that $\alpha TE_{inv} = prep(A_{inv})$.

($\subseteq$) Let $t \in \tau(\widehat{Q}[prep] \parallel_{prep(A_{inv})} TE_{inv})[domain]$. Then there exists

$$w \in \tau(\widehat{Q}[prep] \parallel_{prep(A_{inv})} TE_{inv})$$

such that $domain(w) = t$. Thus, there exists $s \in \tau(\widehat{Q}[prep])$, $u \in \tau TE_{inv}$ such that $w \in (s \parallel_{prep(A_{inv})} u)$. By PA1, $events(u) \subseteq prep(A_{inv})$ and so $w = s \in \tau(\widehat{Q}[prep])$. Hence, by proposition C.8, $t = domain(w) \in \tau\widehat{Q}$.

($\supseteq$) Let $t \in \tau\widehat{Q}$. Thus, $u \in \tau(\widehat{Q}[prep])$ for every $u \in prep(t)$. Moreover, by proposition C.8, $domain(u) = t$ for all such u. Thus, it suffices to show that there exists $u \in prep(t)$ such that $u\lceil prep(A_{inv}) \in \tau TE_{inv}$. By proposition 6.4(1) and TR-GLOBAL1, $t\lceil A_i \in Dom_i$ for $i \in inv$. Hence, by proposition C.4(2) and for $i \in inv$, there exists $w_i \in \tau TE_i$ such that $domain(w_i) = t\lceil A_i$. Thus, by EP-UNI1 and definition 6.2(1), $t\lceil A_{inv} \in |||_{i\in inv} domain(w_i)$. Hence,

$$t\lceil A_{inv} \in |||_{i\in inv} domain(w_i) = domain(|||_{i\in inv} w_i)$$

and so there exists $w \in \tau TE_{inv}$ such that $domain(w) = t\lceil A_{inv}$. Thus, by lemma C.6, $w \in prep(t\lceil A_{inv})$. Let $a \in events(t)$ be such that $a \notin A_{inv}$. Then $prep(a) = a$ by definitions 6.2(1) and 6.8; moreover, $a \in A_j$ for some $j \notin inv$ by proposition 6.3(2), definition 4.5 and PA1. Hence, $prep(a) = a \notin prep(A_{inv})$ by DIS and definitions 6.6 and 6.8. Thus, since $events(w) \subseteq prep(A_{inv})$ by PA1, there exists $w' \in prep(t)$ such that $w'\lceil prep(A_{inv}) = w \in \tau TE_{inv}$. □

Proposition C.10. *Let $1 \leq i, j \leq m$ be such that $i \neq j$. Then $prep(A_i) \cap prep(A_j) = \varnothing$.*

Proof. The proof follows by DIS, EP-UNI1 and definitions 6.6 and 6.8. □

Proposition C.11. *Let $u \circ \langle a \rangle \in \tau(\widehat{Q}[prep] \parallel_{prep(A_{inv})} TE_{inv})$ and $i \in inv$. Then $domain(a) \in A_i$ if and only if $a \in prep(A_i)$.*

Proof. By proposition C.5(2), we assume $\alpha TE_{inv} = prep(A_{inv})$. We observe there exists $s \in \tau(\widehat{Q}[prep])$, $v \in \tau TE_{inv}$ such that $u \circ \langle a \rangle \in (s \parallel_{prep(A_{inv})} v)$. By PA1, $events(v) \subseteq prep(A_{inv})$ and so

$$u \circ \langle a \rangle = s \in \tau(\widehat{Q}[prep]).$$

Thus, there exists $w \circ \langle b \rangle \in \tau\widehat{Q}$ such that $u \circ \langle a \rangle \in prep(w \circ \langle b \rangle)$ and, by proposition C.7, $domain(a) = b$. Moreover, by proposition 6.3(2), definition 4.5 and PA1, there exists $ep_j \in EP(Q)$ such that $b \in A_j$ and so, since $a \in prep(b)$, $a \in prep(A_j)$. We finally consider each of two cases in turn.

Case 1: $domain(a) = b \in A_i$. Since $a \in prep(b)$, then $a \in prep(A_i)$.

Case 2: $domain(a) = b \notin A_i$. In this case, $i \neq j$. Thus, since $a \in prep(A_j)$, $a \notin prep(A_i)$ by proposition C.10. □

Lemma C.12. *Let $w \in \tau(\widehat{Q}[prep] \parallel_{prep(A_{inv})} TE_{inv})$. Then $domain(w)\lceil A_i = domain(w\lceil prep(A_i))$ for $i \in inv$.*

Proof. We proceed by induction on the length of w. In the base case, when $w = \langle\,\rangle$, the proof is immediate. Let $w = v \circ \langle a \rangle$. By proposition C.11, we observe that

$$domain(\langle a \rangle)\lceil A_i = domain(\langle a \rangle\lceil prep(A_i)).$$

and so the proof follows by the inductive hypothesis. □

Proposition C.13. *Let $w \in \tau(\widehat{Q}[prep] \parallel_{prep(A_{inv})} TE_{inv})$ and $domain(w) = t$. Then $extract(w \setminus A_{inv}) = extr_{EP(Q)}(t)$.*

Proof. The proof proceeds by induction on the length of w. In the base case, when $w = t = \langle\,\rangle$, the proof is immediate by TR-GLOBAL2. Let $w = u \circ \langle a \rangle$. By proposition C.9, $domain(w) = domain(u) \circ \langle domain(a) \rangle \in \tau\widehat{Q}$. Hence, by proposition 6.4(1), $domain(w) \in Dom_{EP(Q)}$. Thus, by TR-GLOBAL2, where $domain(a) \in A_i$ for $ep_i \in EP(Q)$,

$$extr_{EP(Q)}(domain(w)) = extr_{EP(Q)}(domain(u)) \circ r$$

such that $extr_i(domain(w)\lceil A_i) = extr_i(domain(u)\lceil A_i) \circ r$. We also observe that

$$extract(w \setminus A_{inv}) = extract(u \setminus A_{inv}) \circ extract(\langle a \rangle \setminus A_{inv})$$

and so, by the inductive hypothesis, we show that $extract(\langle a \rangle \setminus A_{inv}) = r$. We consider each of two cases in turn.

Case 1: $i \notin inv$. In this case, $domain(a) \notin A_{inv}$ by definition 6.2(1) and EP-UNI1. Thus, $domain(a) = a \notin A_{inv}$ since, by EP3-T and definitions 6.2(1) and 6.7(1), $domain(b) \in A_{inv}$ for all b such that $domain(b)$ is defined explicitly. Hence, $a \in A_i$ and so $extract(a) = a$ by DIS and definition 6.7(2).

Thus, $extract(\langle a\rangle \setminus A_{inv}) = \langle a\rangle$ and so it suffices to show that $r = \langle a\rangle$. Since $i \notin inv$, then $a \in A_i \subseteq Fvis$. Thus, since $domain(a) = a$, we observe that

$$extr_i(domain(u)\lceil A_i \circ \langle a\rangle) = extr_i(domain(u)\lceil A_i) \circ r$$

and so, by EP4-FVI, $r = \langle a\rangle$.

Case 2: $i \in inv$. By proposition C.5(1,2), we assume that $\alpha TE_j = prep(A_j)$ for $j \in inv$ and $\alpha TE_{inv} = prep(A_{inv})$. By definition, there exists $s \in \tau(\widehat{Q}[prep])$, $v \in \tau TE_{inv}$ such that $w \in (s \parallel_{prep(A_{inv})} v)$ and so, by TRP, $w\lceil prep(A_{inv}) = v\lceil prep(A_{inv})$. Thus, by PA1, $w\lceil prep(A_{inv}) = v \in \tau TE_{inv}$. By proposition C.10, $prep(A_j) \cap prep(A_k) = \varnothing$ for $j, k \in inv$ such that $j \neq k$. Hence, by PA1,

$$w\lceil prep(A_i) = (w\lceil prep(A_{inv}))\lceil prep(A_i) \in \tau TE_i.$$

Moreover, since $w = u \circ \langle a\rangle$, $u\lceil prep(A_i) \in \tau TE_i$. We know that

$$extr_i(domain(w)\lceil A_i) = extr_i(domain(u)\lceil A_i) \circ r.$$

Thus, by lemma C.12,

$$extr_i(domain(w\lceil prep(A_i))) = extr_i(domain(u\lceil prep(A_i))) \circ r$$

and so, by proposition C.4(3),

$$extract((w\lceil prep(A_i)) \setminus A_i) = extract((u\lceil prep(A_i)) \setminus A_i) \circ r.$$

Thus, $r = extract((\langle a\rangle\lceil prep(A_i)) \setminus A_i)$. By proposition C.11 and since $domain(a) \in A_i$, then $a \in prep(A_i)$ and so $r = extract(\langle a\rangle \setminus A_i)$. Hence, we have to show that

$$extract(\langle a\rangle \setminus A_{inv}) = extract(\langle a\rangle \setminus A_i).$$

Thus, it suffices to show that $a \in A_i$ if and only if $a \in A_{inv}$. If $a \in A_i$, then $a \in A_{inv}$ by definition 6.2(1). We therefore assume that $a \in A_{inv}$ and so $a \in A_j$ for some $j \in inv$. Since $w\lceil prep(A_i) \in \tau TE_i$ and $a \in prep(A_i)$, then $a \in \beta(TE_i)$ by PA1. Hence, by proposition 6.6, there exists $x \circ \langle b\rangle \in Dom_i$ such that $a = \pi_i(b, x)$. Moreover, $b \in A_i$ by EP3-T. Thus, by DIS, definition 6.6 and since $a \in A_j$ for some $j \in inv$, $a = b \in A_i$. □

Proof of theorem 6.7

Proof. The proof follows by proposition 6.4(1), TR-DEF1, proposition C.9 and proposition C.13. □

C.3 Proofs from section 6.5

Lemma C.14. $\bigcap_{R\in ref_i^M(t)}(\bigcup_{a\in(A_i-R)}\{X\subseteq A_i \mid a\notin X\}) = \overline{ref}_i(t)$ *for* $i\in inv$ *and* $t\in Dom_i$.

Proof. We first observe that $ref_i^M(t)$ is non-empty by definition 6.10 and EP5.

($\subseteq$) Let $S\in\bigcap_{R\in ref_i^M(t)}(\bigcup_{a\in(A_i-R)}\{X\subseteq A_i \mid a\notin X\})$. Thus, for every $R\in ref_i^M(t)$, there exists $a\in(A_i-R)$ such that $a\notin S$. Hence, for every $R\in ref_i^M(t)$, $S\cup R\neq A_i$. By definition 6.10, for every $X\in ref_i(t)$ there exists $R\in ref_i^M(t)$ such that $X\subseteq R$ and so $S\in\overline{ref}_i(t)$ by definition 4.9 (note that $S\subseteq A_i$).

($\supseteq$) Let $S\in\overline{ref}_i(t)$. Then, by definition 4.9, $S\subseteq A_i$ and, for every $R\in ref_i(t)$, $R\cup S\neq A_i$. Thus, for every $R\in ref_i^M(t)$, $R\cup S\neq A_i$. Hence, for every $R\in ref_i^M(t)$, there exists $a\in A_i-R$ such that $a\notin S$ and so the proof follows. □

Proof of lemma 6.9

Proof. We consider each of two cases in turn.

Case 1: $Comm(A_i,\widehat{Q}) = Right$.

Case 1a: $t\in dom_i$. In this case,
$\phi DSF_i^R(t) = \{(\langle a\rangle\circ s, X) \mid a\in Next_i(t) \ \wedge\ (s,X)\in\phi DSF_i^R(t\circ\langle a\rangle)\}$.

Case 1b: $t\in Dom_i - dom_i$. In this case,

$$\begin{aligned}\phi DSF_i^R(t) =& \{(\langle a\rangle\circ s, X) \mid a\in Next_i(t) \ \wedge\ (s,X)\in\phi DSF_i^R(t\circ\langle a\rangle)\}\ \cup\\ & \textstyle\bigcup_{R\in ref_i^M(t)}\{(\langle\rangle, Y\cup Z) \mid Y\subseteq R \ \wedge\ Z\subseteq(\Sigma-A_i)\}.\end{aligned}$$

By definition 6.10 and EP5, $ref_i(t)$ is the subset-closure of $ref_i^M(t)$ and so, by definition 6.9,

$$\begin{aligned}\phi DSF_i^R(t) =& \{(\langle a\rangle\circ s, X) \mid a\in Next_i(t) \ \wedge\ (s,X)\in\phi DSF_i^R(t\circ\langle a\rangle)\}\ \cup\\ & \{(\langle\rangle, Y\cup Z) \mid Y\in RefSet_i(t) \ \wedge\ Z\subseteq(\Sigma-A_i)\}.\end{aligned}$$

Case 2: $Comm(A_i,\widehat{Q}) = Left$.

Case 2a: $t\in dom_i$. In this case,
$\phi DSF_i^L(t) = \{(\langle a\rangle\circ s, X) \mid a\in Next_i(t) \ \wedge\ (s,X)\in\phi DSF_i^L(t\circ\langle a\rangle)\}$.

Case 2b: $t\in Dom_i - dom_i$. In this case,

$$\begin{aligned}\phi DSF_i^L(t) =& \{(\langle a\rangle\circ s, X) \mid a\in Next_i(t) \ \wedge\ (s,X)\in\phi DSF_i^L(t\circ\langle a\rangle)\}\ \cup\\ & \textstyle\bigcap_{R\in ref_i^M(t)}(\bigcup_{a\in(A_i-R)}\{(\langle\rangle, X) \mid a\notin X\}).\end{aligned}$$

Thus, by lemma C.14 and definition 6.9,

$$\begin{aligned}\phi DSF_i^L(t) =& \{(\langle a\rangle\circ s, X) \mid a\in Next_i(t) \ \wedge\ (s,X)\in\phi DSF_i^L(t\circ\langle a\rangle)\}\ \cup\\ & \{(\langle\rangle, Y\cup Z) \mid Y\in RefSet_i(t) \ \wedge\ Z\subseteq(\Sigma-A_i)\}.\end{aligned}$$

We observe that $\langle\rangle \in Dom_i$ by EP3-T and recall that $DSF_i \triangleq DSF_i^R(\langle\rangle)$ or $DSF_i \triangleq DSF_i^L(\langle\rangle)$ as appropriate. The proof then follows from the above by induction on the length of traces using definition 6.4. □

Proof of lemma 6.10

Proof. By proposition 6.3(2) and definition 4.5, $A_i \subseteq \alpha\widehat{Q}$. Thus, $\beta(Proc_i) = \alpha\widehat{Q}$ and so we assume $\alpha Proc_i = \alpha\widehat{Q}$. We then observe that

$$\phi Proc_i = \{(t, R) \mid t \in (\alpha\widehat{Q})^* \ \wedge \ R \subseteq (\Sigma - A_i)\}$$

and so, by SF2 and PA1,

$$\phi(\widehat{Q} \parallel_{\alpha\widehat{Q}} Proc_i) = \{(t, R) \mid (\exists(t, X) \in \phi\widehat{Q}) \ R \subseteq (X \cap A_i) \cup (\Sigma - A_i)\}.$$

Again by SF2 and PA1, $events(t) \subseteq \alpha\widehat{Q}$ and so $t \setminus (\alpha\widehat{Q} - A_i) = t\lceil A_i$ for $(t, X) \in \phi\widehat{Q}$ and so the proof follows. □

Proof of theorem 6.11

Proof. Let $i \in inv$. By definition 6.4 and EP3-T, $\beta(DSF_i) \subseteq A_i$ and so we assume $\alpha DSF_i = A_i$. By proposition 6.3(2) and definition 4.5, $A_i \subseteq \alpha\widehat{Q}$ and so $\beta(Proc_i) = \alpha\widehat{Q}$. Since $\beta(\widehat{Q}) \subseteq \alpha\widehat{Q}$, then $\beta(\widehat{Q}_i) = A_i$ and so we assume $\alpha\widehat{Q}_i = A_i$. Thus, by lemmas 6.9 and 6.10 and since $A_i \subseteq \alpha\widehat{Q}$, $\phi FinalImple_i$ is given by:

$$\{(t\lceil A_i, R) \mid (\exists(t, X) \in \phi\widehat{Q}, Y \in RefSet_i(t\lceil A_i)) \ t\lceil A_i \in Dom_i - dom_i \ \wedge \\ X \subseteq \alpha\widehat{Q} \ \wedge \ R \subseteq (X \cap A_i) \cup Y \cup (\Sigma - A_i)\}.$$

We then observe that, for $(t, X) \in \phi\widehat{Q}$, $t\lceil A_i \in Dom_i$ by proposition 6.4(2) and TR-GLOBAL1. Thus, the proof follows by the definition of Dom-SF-check and definitions 4.9, 4.10 and 6.9. □

C.4 Proofs from section 6.6

Note: Recall that the set of primed events contains only "fresh" events: i.e. it does not contain any events already used in defining $\widehat{Q}$, P or $EP(Q)$, or which are used in any other capacity as part of the verification of $\widehat{Q}$. Recall also that the events in d_{inv} are assumed to be "fresh" in the same sense. A similar condition also holds by DIS from section 6.4. These facts will generally be appealed to implicitly where they are needed in the proofs in this section.

Proof of lemma 6.12

Proof. By definition 6.4 and EP3-T, $\beta(RE_i) \subseteq A_i \cup prime(A_i)$ for $i \in inv$ and so we assume $\alpha RE_i = A_i \cup prime(A_i)$. Thus, by EP-UNI1, $\alpha RE_j \cap \alpha RE_k = \varnothing$ for $j \neq k$, where $j, k \in inv$. Moreover, $\beta(|||_{i\in inv} RE_i) \subseteq A_{inv} \cup prime(A_{inv})$ by definition 6.2(1) and so we assume $\alpha(|||_{i\in inv} RE_i) = A_{inv} \cup prime(A_{inv})$. We also assume $\alpha Trim = A_{inv} \cup prime(A_{inv})$ since $\beta(Trim) = A_{inv} \cup prime(A_{inv})$.

(1) Let $i \in inv$. Wlog, we consider the case that $Comm(A_i, \widehat{Q}) = Right$. In this case, for $t \in Dom_i$, we observe that

$$\tau RE_i^R(t) = \{\langle\rangle\} \cup \{\langle a\rangle \circ s \mid a \in Next_i(t) \ \wedge \ s \in \tau RE_i^R(t \circ \langle a\rangle)\} \ \cup X,$$

where $X \subseteq \{\langle prime(a)\rangle \mid a \in A_i\}$. We observe that $\langle\rangle \in Dom_i$ by EP3-T and recall that $RE_i \triangleq RE_i^R(\langle\rangle)$. Then, by induction on the length of traces using the above and definition 6.4, we observe that

$$\tau RE_i = Dom_i \cup Y, \text{ where } Y \subseteq \{t \circ \langle prime(a)\rangle \mid t \in Dom_i \ \wedge \ a \in A_i\}.$$

We then observe that $Dom_i \subseteq (A_i)^* \subseteq (A_{inv})^*$ by EP3-T and definition 6.2(1). Thus, by definitions 6.2(1) and 6.3(1), $\tau(|||_{j\in inv} RE_j) = Dom_{inv} \cup T$, where

$$T \subseteq \{t \in (A_{inv} \cup prime(A_{inv}))^* \mid t\lceil A_{inv} \in Dom_{inv}\}.$$

We also observe that

$$\tau Trim = (A_{inv})^* \cup \{t \circ \langle prime(a)\rangle \mid t \in (A_{inv})^* \ \wedge \ a \in A_{inv}\}$$

and so the proof of this part follows by definition 6.2(1) (recall also that $Dom_{inv} \subseteq (A_{inv})^*$ by definition 6.3(1)).

(2) Let $i \in inv$. We begin by considering each of two cases in turn.

Case 1: $Comm(A_i, \widehat{Q}) = Right$.

Case 1a: $t \in Dom_i - dom_i$. In this case,
$\phi RE_i^R(t) = \{(\langle a\rangle \circ s, X) \mid a \in Next_i(t) \ \wedge \ (s, X) \in \phi RE_i^R(t \circ \langle a\rangle)\}$.

Case 1b: $t \in dom_i$. In this case,

$$\phi RE_i^R(t) = \{(\langle a\rangle \circ s, X) \mid a \in Next_i(t) \ \wedge \ (s, X) \in \phi RE_i^R(t \circ \langle a\rangle)\} \ \cup$$
$$\bigcup_{R \in ref_i^M(t)} \{(\langle\rangle, prime(S) \cup U) \mid S \subseteq R \ \wedge \ U \subseteq (\Sigma - prime(A_i))\}.$$

By definition 6.10 and EP5, $ref_i(t)$ is the subset-closure of $ref_i^M(t)$ and so, by definition 6.9,

$$\phi RE_i^R(t) = \{(\langle a\rangle \circ s, X) \mid a \in Next_i(t) \ \wedge \ (s, X) \in \phi RE_i^R(t \circ \langle a\rangle)\} \ \cup$$
$$\{(\langle\rangle, prime(S) \cup U) \mid S \in RefSet_i(t) \ \wedge \ U \subseteq (\Sigma - prime(A_i))\}.$$

Case 2: $Comm(A_i, \widehat{Q}) = Left$.

Case 2a: $t \in Dom_i - dom_i$. In this case,
$\phi RE_i^L(t) = \{(\langle a \rangle \circ s, X) \mid a \in Next_i(t) \;\wedge\; (s, X) \in \phi RE_i^L(t \circ \langle a \rangle)\}$.

Case 2b: $t \in dom_i$. In this case,

$$\begin{aligned}\phi RE_i^L(t) = {} & \{(\langle a \rangle \circ s, X) \mid a \in Next_i(t) \;\wedge\; (s, X) \in \phi RE_i^L(t \circ \langle a \rangle)\} \cup \\ & \textstyle\bigcap_{R \in ref_i^M(t)} (\bigcup_{b \in prime(A_i - R)} \{(\langle\rangle, X) \mid b \notin X\}.\end{aligned}$$

Thus, by a proof similar to that of lemma C.14 and also by definition 6.9,

$$\begin{aligned}\phi RE_i^L(t) = {} & \{(\langle a \rangle \circ s, X) \mid a \in Next_i(t) \;\wedge\; (s, X) \in \phi RE_i^L(t \circ \langle a \rangle)\} \cup \\ & \{(\langle\rangle, prime(X) \cup Y) \mid X \in RefSet_i(t) \;\wedge\; Y \subseteq (\Sigma - prime(A_i))\}.\end{aligned}$$

We observe that $\langle\rangle \in Dom_i$ by EP3-T and recall that $RE_i \triangleq RE_i^R(\langle\rangle)$ or $RE_i \triangleq RE_i^L(\langle\rangle)$ as appropriate. Then, by induction on the length of traces using definition 6.4 and the above two cases, ϕRE_i is given by:

$$\{(t, prime(X) \cup Y) \mid t \in dom_i \;\wedge\; X \in RefSet_i(t) \;\wedge\; Y \subseteq (\Sigma - prime(A_i))\}.$$

Hence, by definitions 6.2(1) and 6.3(2),

$$\begin{aligned}\phi(|||_{j \in inv} RE_j) = {} & \{(t, prime(X) \cup Y) \mid t \in dom_{inv} \;\wedge\; X \subseteq A_{inv} \;\wedge \\ & Y \subseteq (\Sigma - prime(A_{inv})) \;\wedge\; ((\forall i \in inv)\; X \cap A_i \in RefSet_i(t \lceil A_i))\}.\end{aligned}$$

We then observe that

$$\phi Trim = \{(t, R) \mid t \in (A_{inv})^* \;\wedge\; R \subseteq \Sigma - (A_{inv} \cup prime(A_{inv}))\}.$$

and so the proof of this part follows by theorem 2.20 and SF3 (recall that $dom_{inv} \subseteq (A_{inv})^*$ by definition 6.3(2)). □

Lemma C.15. *The following hold:*

1. $\tau Interim = \tau\widehat{Q} \cup \{t \circ \langle prime(a) \rangle \mid t \circ \langle a \rangle \in \tau\widehat{Q} \;\wedge\; ((\exists i \in inv)\; a \in A_i)\}$.

2. $\phi Interim = \{(t, R) \mid (\exists\; (t, X) \in \phi\widehat{Q})\; X \cap prime(A_{inv}) = \varnothing \;\wedge$
   $R \subseteq X \cup prime(X \cap A_{inv})\}$.

Proof. Since $\beta(\widehat{Q}) \subseteq \alpha\widehat{Q}$, then $\beta(\widehat{Q}[p^Q]) \subseteq p^Q(\alpha\widehat{Q})$. By proposition 6.3(2), definition 4.5 and definition 6.2(1), $A_{inv} \subseteq \alpha\widehat{Q}$. Hence, by definition 6.11, we assume

$$\alpha(\widehat{Q}[p^Q]) = p^Q(\alpha\widehat{Q}) = \alpha\widehat{Q} \cup prime(A_{inv}).$$

Moreover, we assume $\alpha TrimTwo = \beta(TrimTwo) = \alpha\widehat{Q} \cup prime(A_{inv})$.

(1) We observe that:

- $\tau(\widehat{Q}[p^Q]) \;=\; \{t \mid (\exists s \in \tau\widehat{Q})\; t \in p^Q(s)\}$.

- $\tau TrimTwo \;=\; (\alpha\widehat{Q})^* \cup \; \{t \circ \langle prime(a)\rangle \mid t \in (\alpha\widehat{Q})^* \wedge ((\exists i \in inv)\; a \in A_i)\}$.

- $\tau\widehat{Q} \subseteq \tau(\widehat{Q}[p^Q])$ by definition 6.11(2).

Hence, the proof follows by PA1 and definition 6.11.

(2) We first observe that

$$\phi(\widehat{Q}[p^Q]) = \{(t,R) \mid (\exists s)\; t \in p^Q(s) \;\wedge\; (s,(p^Q)^{-1}(R)) \in \phi\widehat{Q}\}.$$

Let $(t,R) \in \phi\widehat{Q}$. Then, by PA2 and SF3, $(t, R \cup S) \in \phi\widehat{Q}$ where $S \subseteq prime(A_{inv})$. Hence, by definition 6.11,

$$\begin{array}{rcl}\phi(\widehat{Q}[p^Q]) & = & \{(t,R) \mid (\exists\; (s,X) \in \phi\widehat{Q})\; t \in p^Q(s) \;\wedge \\ & & X \cap prime(A_{inv}) = \varnothing \;\wedge \\ & & R \subseteq X \cup prime(X \cap A_{inv})\}.\end{array}$$

We also observe that:

$$\phi TrimTwo = \{(t,R) \mid t \in (\alpha\widehat{Q})^* \;\wedge\; R \subseteq \Sigma - (\alpha\widehat{Q} \cup prime(A_{inv}))\}.$$

Thus, the proof follows by definition 6.11(2), SF2, PA1, theorem 2.20 and SF3. □

Proof of lemma 6.13

Proof. Since $\beta(\widehat{Q}) \subseteq \alpha\widehat{Q}$, then $\beta(\widehat{Q}[p^Q]) \subseteq p^Q(\alpha\widehat{Q})$. We also observe that, by proposition 6.3(2) and definition 4.5, $\alpha\widehat{Q} = A_{Fvis} \cup A_{inv}$. Thus, by definition 6.11, $\beta(\widehat{Q}[p^Q]) \subseteq \alpha\widehat{Q} \cup prime(A_{inv})$. Moreover, $\beta(TrimTwo) = \alpha\widehat{Q} \cup prime(A_{inv})$. Hence, we assume

$$\alpha Interim = \alpha\widehat{Q} \cup prime(A_{inv}) = A_{Fvis} \cup A_{inv} \cup prime(A_{inv}).$$

Moreover,

- $\beta(|||_{i \in inv} RE_i) \subseteq A_{inv} \cup prime(A_{inv})$.

- $\beta(Trim) = A_{inv} \cup prime(A_{inv})$.

Hence, we assume

$$\alpha RE_{inv} = A_{inv} \cup prime(A_{inv}).$$

(1) By proposition 6.4(1), TR-GLOBAL1, EP-UNI1, definition 6.2(1) and definition 6.3(1), $t\lceil A_{inv} \in Dom_{inv}$ for every $t \in \tau\widehat{Q}$. Moreover, for such t,

$t\lceil(A_{inv} \cup prime(A_{inv})) = t\lceil A_{inv}$. Note also that, for $s \in \tau RE_{inv}$, $events(s) \subseteq A_{inv} \cup prime(A_{inv})$ by PA1. Thus, by lemmas 6.12(1) and C.15(1),

$$\tau(Interim \parallel_{A_{inv} \cup prime(A_{inv})} RE_{inv}) = \tau\widehat{Q} \cup T,$$

where $T \subseteq \{t \circ \langle prime(a)\rangle \mid t \in \tau\widehat{Q} \;\wedge\; ((\exists i \in inv)\; a \in A_i)\}$. Hence, the proof follows by definition 6.13(1).

(2) We observe that, if $(t, R) \in \phi\widehat{Q}$, then $t\lceil(A_{inv} \cup prime(A_{inv})) = t\lceil A_{inv}$ by SF2 and PA1. Moreover, if $t\lceil A_{inv} \in dom_{inv}$, then $t \in dom_{EP(Q)}$ by SF-GLOBAL1, definitions 6.2(1) and 6.3(1), and the fact that $dom_i = A_i^*$ for $i \notin inv$ by EP3-FVI and EP3A-FVI. And if $t \in dom_{EP(Q)}$ then $t\lceil A_{inv} = t\lceil(A_{inv} \cup prime(A_{inv})) \in dom_{inv}$. Also, if $(w, X) \in \phi RE_{inv}$ then, by SF2 and PA1, $events(w) \subseteq A_{inv} \cup prime(A_{inv})$. Thus, by lemmas 6.12(2) and C.15(2) and theorem 2.20, $\phi(Interim \parallel_{A_{inv} \cup prime(A_{inv})} RE_{inv})$ is given by:

$$\begin{aligned}\{(t,R) \mid\; & (\exists (t,X) \in \phi_{dom_{EP(Q)}}\widehat{Q}, Z \subseteq A_{inv})\\ & X \cap prime(A_{inv}) = \varnothing \;\wedge\; ((\forall i \in inv)\; Z \cap A_i \in RefSet_i(t\lceil A_i)) \;\wedge\\ & R \subseteq ((X \cup prime(X \cap A_{inv})) \cap \alpha Interim) \cup\\ & A_{inv} \cup prime(Z) \cup (\Sigma - (A_{Fvis} \cup A_{inv} \cup prime(A_{inv})))\}.\end{aligned}$$

Thus, $\phi(Interim \parallel_{A_{inv} \cup prime(A_{inv})} RE_{inv})$ is given by:

$$\begin{aligned}\{(t,R) \mid\; & (\exists (t,X) \in \phi_{dom_{EP(Q)}}\widehat{Q}, Z \subseteq A_{inv})\\ & X \cap prime(A_{inv}) = \varnothing \;\wedge\; ((\forall i \in inv)\; Z \cap A_i \in RefSet_i(t\lceil A_i)) \;\wedge\\ & R \subseteq (X \cap (A_{Fvis} \cup A_{inv})) \cup prime(X \cap A_{inv}) \cup prime(Z) \cup\\ & (\Sigma - (A_{Fvis} \cup prime(A_{inv})))\}\end{aligned}$$

and so it is given by:

$$\begin{aligned}\{(t,R) \mid\; & (\exists (t,X) \in \phi_{dom_{EP(Q)}}\widehat{Q}, Z \subseteq A_{inv})\\ & X \subseteq \alpha\widehat{Q} \;\wedge\; ((\forall i \in inv)\; Z \cap A_i \in RefSet_i(t\lceil A_i)) \;\wedge\\ & R \subseteq (X \cap A_{Fvis}) \cup prime(X \cap A_{inv}) \cup prime(Z) \cup\\ & (\Sigma - (A_{Fvis} \cup prime(A_{inv})))\}.\end{aligned}$$

Thus, by definition 6.13(1),

$$\begin{aligned}\phi PreImple = \{(t,R) \mid\ & (\exists\ (t,X) \in \phi_{dom_{EP(Q)}}\widehat{Q})\ X \subseteq \alpha\widehat{Q} \wedge \\ & R \subseteq (X \cap A_{Fvis}) \cup \\ & \{d_i \in d_{inv} \mid (\exists Y \in RefSet_i(t\lceil A_i)) \\ & \qquad\qquad (X \cap A_i) \cup Y = A_i\} \cup \\ & (\Sigma - (A_{Fvis} \cup d_{inv}))\}.\end{aligned}$$

Let $(t,R) \in \phi_{dom_{EP(Q)}}\widehat{Q}$ be such that $R \subseteq \alpha\widehat{Q}$. Then, by definition 6.14 and definitions 4.9, 4.10 and 6.9, $extrFDR^{ref}_{EP(Q)}(R,t,\widehat{Q})$ is given by:

$$(R \cap A_{Fvis})\ \cup \{d_i \in d_{inv} \mid (\exists Y \in RefSet_i(t\lceil A_i))\ (R \cap A_i) \cup Y = A_i\}.$$

□

Lemma C.16. *The following hold:*

1. $(PreImple[prep]\ \|_{prep(A_{inv})}\ TE_{inv})[domain] =_T PreImple.$
2. *Let* $w \in \tau(PreImple[prep]\ \|_{prep(A_{inv})}\ TE_{inv})$ *be such that either:*
   - $w = \langle\rangle$ *or;*
   - $w \neq \langle\rangle$ *and* $tail(w) \notin d_{inv}$.

   Then $extract(w \setminus A_{inv}) = extr_{EP(Q)}(domain(w))$.
3. *Let* $w \in \tau(PreImple[prep]\ \|_{prep(A_{inv})}\ TE_{inv})$ *be such that* $w = u \circ \langle d_i\rangle$, *where* $d_i \in d_{inv}$. *Then* $extract(w \setminus A_{inv}) = extr_{EP(Q)}(domain(u)) \circ \langle d_i\rangle$.

Proof. By proposition C.5(2), we assume that $\alpha TE_{inv} = prep(A_{inv})$.

(1) The proof is similar to that of proposition C.9, using lemma 6.13(1).

(2) By PA1, $events(t) \subseteq prep(A_{inv})$ for $t \in \tau TE_{inv}$. Thus, and by lemma 6.13(1) and TRP, $w \in \tau(\widehat{Q}[prep]\ \|_{prep(A_{inv})}\ TE_{inv})$. Hence, the proof follows by proposition C.13.

(3) By lemma 6.13(1), PA1 and TRP, $u \in \tau(\widehat{Q}[prep]\ \|_{prep(A_{inv})}\ TE_{inv})$. Hence, $extract(u \setminus A_{inv}) = extr_{EP(Q)}(domain(u))$ by proposition C.13 and so $extract(w \setminus A_{inv}) = extr_{EP(Q)}(domain(u)) \circ \langle d_i\rangle$.

□

Proposition C.17. *If* $(t,R) \in \phi PreImple$ *and* $S \subseteq A_{inv} \cup B_{inv} \cup prep(A_{inv})$, *then* $(t, R \cup S) \in \phi PreImple$.

Proof. We observe the following:

- $A_{inv} \cap A_{Fvis} = \varnothing$ by EP-UNI1 and definitions 6.2(1) and 6.15.
- By EP1-FVI and definition 6.15, $A_{Fvis} = \bigcup_{i \notin inv} A_i = \bigcup_{i \notin inv} B_i$ and so $B_{inv} \cap A_{Fvis} = \varnothing$ by EP-UNI1 and definition 6.2(2).
- $prep(A_{inv}) \cap A_{Fvis} = \varnothing$ by EP-UNI1, DIS and definitions 6.2(1), 6.6, 6.8 and 6.15.
- By definition, $(A_{inv} \cup B_{inv} \cup prep(A_{inv})) \cap d_{inv} = \varnothing$.

Hence, $(A_{inv} \cup B_{inv} \cup prep(A_{inv})) \cap (A_{Fvis} \cup d_{inv}) = \varnothing$ and so the proof follows by lemma 6.13(2). □

Proposition C.18. *The following holds:*

$$\phi(PreImple[prep]) = \{(t, R) \mid (\exists s)\ t \in prep(s)\ \wedge\ (s, R) \in \phi PreImple\}.$$

Proof. We first observe that

$$\phi(PreImple[prep]) = \{(t, R) \mid (\exists s)\, t \in prep(s) \wedge (s, prep^{-1}(R)) \in \phi PreImple\}.$$

Thus, the proof follows by proposition C.17 and definitions 6.2(1) and 6.8. □

Lemma C.19. *$\phi(PreImple[prep])\ \|_{prep(A_{inv})}\ TE_{inv})$ is given by:*

$$\{(t, R) \mid (t, R) \in \phi(PreImple[prep]) \wedge t \in \tau(PreImple[prep])\ \|_{prep(A_{inv})}\ TE_{inv})\}.$$

Proof. By proposition C.5(2), we assume that $\alpha TE_{inv} = prep(A_{inv})$. We also assume that $\alpha(PreImple[prep]) = \beta(PreImple[prep]) \cup prep(A_{inv})$.

($\subseteq$) Let $(t, R) \in \phi(PreImple[prep])\ \|_{prep(A_{inv})}\ TE_{inv})$. Then, by theorem 2.20, there exists $(s, S) \in \phi(PreImple[prep])$, $(u, U) \in \phi TE_{inv}$ such that:

- $t \in (s\ \|_{prep(A_{inv})}\ u)$.
- $S \subseteq \alpha(PreImple[prep])$ and $U \subseteq \alpha TE_{inv}$.
- $R = S \cup U \cup Z$, where, since $\alpha TE_{inv} \subseteq \alpha(PreImple[prep])$, $Z \subseteq (\Sigma - \alpha(PreImple[prep]))$.

By SF2, $t \in \tau(PreImple[prep])\ \|_{prep(A_{inv})}\ TE_{inv})$. Moreover, by SF2 and PA1, $events(u) \subseteq prep(A_{inv})$. Hence, $t = s$ and so $(t, S) \in \phi(PreImple[prep])$. Thus, by PA2 and SF3, $(t, S \cup Z) \in \phi(PreImple[prep])$. Moreover, by propositions C.17 and C.18, and since $U \subseteq prep(A_{inv})$, $(t, S \cup U \cup Z) \in \phi(PreImple[prep])$.

($\supseteq$) Let $(t, R) \in \phi(PreImple[prep])$ be such that

$$t \in \tau(PreImple[prep]) \parallel_{prep(A_{inv})} TE_{inv}).$$

Thus, by TRP and PA1, $t\lceil prep(A_{inv}) \in \tau TE_{inv}$. We then observe that, for $i \in inv$, TE_i is guarded and so $\delta TE_i = \varnothing$ by DF. Hence, $\delta TE_{inv} = \varnothing$ and so $\tau TE_{inv} = \{t \mid (t, \varnothing) \in \phi TE_{inv}\}$ by proposition 2.3(2). Thus, $(t\lceil prep(A_{inv}), \varnothing) \in \phi TE_{inv}$. We also observe $R = (R \cap \alpha(PreImple[prep])) \cup Z$, where $Z \subseteq \Sigma - \alpha(PreImple[prep])$. Moreover, $(t, R \cap \alpha(PreImple[prep])) \in \phi(PreImple[prep])$ by SF3. Hence, by theorem 2.20 and since $\alpha TE_{inv} \subseteq \alpha(PreImple[prep])$,

$$(t, R) \in \phi(PreImple[prep]) \parallel_{prep(A_{inv})} TE_{inv}).$$

□

Lemma C.20. *$\phi(PreImple[prep]) \parallel_{prep(A_{inv})} TE_{inv})$ is given by:*

$$\{(t, R) \mid (domain(t), R) \in \phi PreImple \wedge t \in \tau(PreImple[prep]) \parallel_{prep(A_{inv})} TE_{inv})\}.$$

Proof. By proposition C.5(2), we assume that $\alpha TE_{inv} = prep(A_{inv})$.

($\subseteq$) Let $(t, R) \in \phi(PreImple[prep]) \parallel_{prep(A_{inv})} TE_{inv})$. Then, by lemma C.19, $(t, R) \in \phi(PreImple[prep])$ and $t \in \tau(PreImple[prep]) \parallel_{prep(A_{inv})} TE_{inv})$. Thus, by proposition C.18, there exists s such that $t \in prep(s)$ and $(s, R) \in \phi PreImple$. Hence, by lemma 6.13(2) and SF2, $s \in \tau\widehat{Q}$ and so, by proposition C.8, $domain(t) = s$.

($\supseteq$) Let (t, R) be such that $t \in \tau(PreImple[prep]) \parallel_{prep(A_{inv})} TE_{inv})$ and $(domain(t), R) \in \phi PreImple$. Thus, by PA1, $t \in \tau(PreImple[prep])$ and so there exists $s \in \tau PreImple$ such that $t \in prep(s)$. Moreover, by lemma 6.13(2) and SF2, $domain(t) \in \tau\widehat{Q}$. Hence, if $t \neq \langle\rangle$, then by definition 6.7(1) $tail(t) \notin d_{inv}$ and so, by definition 6.8, $tail(s) \notin d_{inv}$. Moreover, if $t = \langle\rangle$ then $s = \langle\rangle$. As a result, by lemma 6.13(1), $s \in \tau\widehat{Q}$. Thus, by proposition C.8, $domain(t) = s$ and so the proof follows by proposition C.18 and lemma C.19. □

Proof of lemma 6.14

Proof. By proposition C.5(2), we assume that $\alpha TE_{inv} = prep(A_{inv})$.

(1) The proof of this part follows by proposition 6.4(1), TR-DEF1 and lemmas C.16 and 6.13(1). (Note also that, by definition 6.7(1), $domain(d_i) = d_i$ for $d_i \in d_{inv}$ and $domain(a) \notin d_{inv}$ for $a \notin d_{inv}$.)

(2) By lemma C.20, $\phi(\mathit{PreImple}[\mathit{prep}]) \parallel_{\mathit{prep}(A_{inv})} TE_{inv})$ is given by:

$$\{(t,R) \mid (\mathit{domain}(t),R) \in \phi\mathit{PreImple} \wedge t \in \tau(\mathit{PreImple}[\mathit{prep}]) \parallel_{\mathit{prep}(A_{inv})} TE_{inv})\}.$$

Thus, by proposition C.17, $\phi(\mathit{PreImple}[\mathit{prep}]) \parallel_{\mathit{prep}(A_{inv})} TE_{inv}) \setminus A_{inv}$ is given by:

$$\{(t \setminus A_{inv},R) \mid (\mathit{domain}(t),R) \in \phi\mathit{PreImple} \wedge t \in \tau(\mathit{PreImple}[\mathit{prep}]) \parallel_{\mathit{prep}(A_{inv})} TE_{inv})\}.$$

Hence, by proposition C.17, EP3-T, EP4 and definitions 6.2, 6.7(2) and 6.8,

$$\phi\mathit{FinalImple} = \{(\mathit{extract}(t \setminus A_{inv}),R) \mid (\mathit{domain}(t),R) \in \phi\mathit{PreImple} \wedge t \in \tau(\mathit{PreImple}[\mathit{prep}]) \parallel_{\mathit{prep}(A_{inv})} TE_{inv})\}.$$

Let $(\mathit{domain}(t),R) \in \phi\mathit{PreImple}$. Thus, by lemma 6.13(2) and definition 6.7(1), if $t \neq \langle\rangle$ then $\mathit{tail}(t) \notin d_{inv}$. Hence, by lemma C.16(2),

$$\phi\mathit{FinalImple} = \{(\mathit{extr}_{EP(Q)}(\mathit{domain}(t)),R) \mid (\mathit{domain}(t),R) \in \phi\mathit{PreImple} \wedge t \in \tau(\mathit{PreImple}[\mathit{prep}]) \parallel_{\mathit{prep}(A_{inv})} TE_{inv})\}.$$

Thus, $\phi\mathit{FinalImple} = \{(\mathit{extr}_{EP(Q)}(w),R) \mid (w,R) \in \phi\mathit{PreImple}\}$ by SF2 and lemma C.16(1) and so the proof follows by lemma 6.13(2). □

Proposition C.21. $\mathit{extr}^{set}(\alpha\widehat{Q}) = A_{Fvis} \cup B_{inv}$.

Proof. By proposition 6.3(2) and definitions 4.2 and 4.5, $\mathit{extr}^{set}(\alpha\widehat{Q}) = \bigcup_{1\leq i\leq m} B_i$. Thus, the proof follows by EP1-FVI and definitions 6.2(2) and 6.15. □

Proof of lemma 6.15

Proof. Since $\beta(P) \subseteq \alpha P$, then $\beta(P[p^P]) \subseteq p^P(\alpha P)$. Moreover, by proposition C.21, $B_{inv} \subseteq \alpha P$. Hence, by definition 6.11, we assume

$$\alpha(P[p^P]) = p^P(\alpha P) = \alpha P \cup \mathit{prime}(B_{inv}).$$

We also assume $\alpha\mathit{Proc} = \beta(\mathit{Proc}) = \alpha P \cup \mathit{prime}(B_{inv}) \cup d_{inv}$.

(1) We first observe that

$$\begin{aligned}\tau Proc = \; & (\alpha P)^* \cup \\ & \{t \circ \langle a\rangle \mid t \in (\alpha P)^* \;\wedge\; a \in (\alpha P - B_{inv})\} \cup \\ & \{t \circ prime(\langle a\rangle) \mid t \in (\alpha P)^* \;\wedge\; a \in B_{inv}\} \cup \\ & \{t \circ \langle d_i\rangle \mid t \in (\alpha P)^* \;\wedge\; d_i \in d_{inv}\}.\end{aligned}$$

We also observe that $\tau(P[p^P]) = \{t \mid (\exists s \in \tau P)\; t \in p^P(s)\}$ and so, by definition 6.11(3), $\tau P \subseteq \tau(P[p^P])$. Hence, by PA1 and definition 6.11, $\tau(P[p^P] \parallel_{\alpha P \cup prime(B_{inv})} Proc)$ is given by:

$$\begin{aligned}\tau P \; & \cup \{t \circ \langle prime(a)\rangle \mid t \circ \langle a\rangle \in \tau P \;\wedge\; a \in B_{inv}\} \\ & \cup \{t \circ \langle d_i\rangle \mid t \in \tau P \;\wedge\; d_i \in d_{inv}\}.\end{aligned}$$

Thus, the proof of this part follows by definitions 6.2(2) and 6.13(2).

(2) We first observe that

$$\phi Proc = \{(t,R) \mid t \in (\alpha P)^* \;\wedge\; R \subseteq \Sigma - ((\alpha P - B_{inv}) \cup prime(B_{inv}))\}.$$

By definition,

$$\phi(P[p^P]) = \{(t,R) \mid (\exists s)\; t \in p^P(s) \;\wedge\; (s, (p^P)^{-1}(R)) \in \phi P\}.$$

Let $(t,R) \in \phi P$. Then, by PA2 and SF3, $(t, R \cup S) \in \phi P$ where $S \subseteq prime(B_{inv})$. Hence, by definition 6.11,

$$\begin{aligned}\phi(P[p^P]) = \; & \{(t,R) \mid (\exists\; (s,X) \in \phi P)\; t \in p^P(s) \;\wedge \\ & X \cap prime(B_{inv}) = \varnothing \;\wedge \\ & R \subseteq X \cup prime(X \cap B_{inv})\}.\end{aligned}$$

Thus, by theorem 2.20, SF2, PA1 and SF3, $\phi(P[p^P] \parallel_{\alpha P \cup prime(B_{inv})} Proc)$ is given by:

$$\begin{aligned}\{(t,R) \mid \; & (\exists (t,X) \in \phi P)\; X \cap prime(B_{inv}) = \varnothing \;\wedge \\ & R \subseteq ((X \cup prime(X \cap B_{inv})) \cap (\alpha P \cup prime(B_{inv}))) \cup \\ & ((\Sigma - ((\alpha P - B_{inv}) \cup prime(B_{inv}))) \cap (\alpha P \cup prime(B_{inv}) \cup d_{inv})) \\ & \cup (\Sigma - (\alpha P \cup prime(B_{inv}) \cup d_{inv}))\}\end{aligned}$$

and so, since $B_{inv} \subseteq \alpha P$ by proposition C.21, it is given by:

$$\begin{aligned}\{(t,R) \mid \; & (\exists (t,X) \in \phi P)\; X \cap prime(B_{inv}) = \varnothing \;\wedge \\ & R \subseteq (X \cap \alpha P) \cup prime(X \cap B_{inv}) \cup B_{inv} \cup d_{inv} \cup \\ & (\Sigma - (\alpha P \cup prime(B_{inv}) \cup d_{inv}))\}.\end{aligned}$$

Hence, $\phi(P[p^P] \parallel_{\alpha P \cup prime(B_{inv})} Proc)$ is given by:

$$\begin{aligned}\{(t,R) \mid\ & (\exists(t,X) \in \phi P) \\ & R \subseteq (X \cap (\alpha P - B_{inv})) \cup prime(X \cap B_{inv}) \cup \\ & (\Sigma - ((\alpha P - B_{inv}) \cup prime(B_{inv})))\}\end{aligned}$$

and so the proof of this part follows by definition 6.13(2). □

Proof of theorem 6.16

Proof. We recall that $\alpha P = \beta(P) \cup extr^{set}(\alpha\widehat{Q})$.

($\Longrightarrow$) We assume that $extr_{EP(Q)}([\![\widehat{Q}]\!]_{SF} \subseteq [\![P]\!]_{SF}$. Thus, by SF-DEF1, $extr_{EP(Q)}(\tau\widehat{Q}) \subseteq \tau P$ and $extr_{EP(Q)}(\phi\widehat{Q}) \subseteq \phi P$. Hence, by TR-DEF1 and lemmas 6.14(1) and 6.15(1), $\tau FinalImple \subseteq \tau NewSpec$. We now show, therefore, that $\phi FinalImple \subseteq \phi NewSpec$.

Let $(t, X \cup Y) \in \phi FinalImple$, where, by lemma 6.14(2), there exists $(w,R) \in \phi_{dom_{EP(Q)}}\widehat{Q}$ such that:

- $extr_{EP(Q)}(w) = t$ and $R \subseteq \alpha\widehat{Q}$.
- $X \subseteq extrFDR^{ref}_{EP(Q)}(R, w, \widehat{Q})$.
- $Y \subseteq (\Sigma - (A_{Fvis} \cup d_{inv}))$.

Since $extr_{EP(Q)}(\phi\widehat{Q}) \subseteq \phi P$ then, by SF-DEF2, $(t,Z) \in \phi P$ such that:

$$Z = extr^{ref}_{EP(Q)}(R, w, \widehat{Q}) \cup (\Sigma - extr^{set}(\alpha\widehat{Q})).$$

Thus, by lemma 6.15(2), $(t,S) \in \phi NewSpec$, where:

$$S = (Z \cap (\alpha P - B_{inv})) \cup DB(Z) \cup (\Sigma - ((\alpha P - B_{inv}) \cup d_{inv})).$$

By proposition C.21 and definition 6.2(2), $B_i \subseteq extr^{set}(\alpha\widehat{Q})$ for $i \in inv$. Moreover, for $i \in inv$ and by definition 4.10, EP5-FVI, EP1-FVI, SF-GLOBAL2 and EP-UNI1, $B_i \subseteq extr^{ref}_{EP(Q)}(R, w, \widehat{Q})$ if and only if $extr^{ref}_i(R \cap A_i, w\lceil A_i, \widehat{Q}) = B_i$. Thus, by definitions 6.14 and 6.16,

$$DB(extr^{ref}_{EP(Q)}(R, w, \widehat{Q}) \cup (\Sigma - extr^{set}(\alpha\widehat{Q}))) = \bigcup_{i \in inv} extrFDR^{ref}_i(R \cap A_i, w\lceil A_i, \widehat{Q}).$$

By SF-GLOBAL2, EP-UNI1, EP1-FVI, EP5-FVI and definitions 4.10 and 6.2(2),

$$extr^{ref}_{EP(Q)}(R, w, \widehat{Q}) - B_{inv} = \bigcup_{i \notin inv} extr^{ref}_i(R \cap A_i, w\lceil A_i, \widehat{Q}).$$

Moreover, $extr^{ref}_{EP(Q)}(R, w, \widehat{Q}) \subseteq extr^{set}(\alpha\widehat{Q}) \subseteq \alpha P$. Thus,

$$extr^{ref}_{EP(Q)}(R, w, \widehat{Q}) \cap (\alpha P - B_{inv}) = \bigcup_{i \notin inv} extr^{ref}_i(R \cap A_i, w\lceil A_i, \widehat{Q})$$

and so, by EP5-FVI and definition 6.14(1),

$$extr^{ref}_{EP(Q)}(R, w, \widehat{Q}) \cap (\alpha P - B_{inv}) = \bigcup_{i \notin inv} extrFDR^{ref}_i(R \cap A_i, w\lceil A_i, \widehat{Q}).$$

In addition, $(\alpha P - B_{inv}) \cap (\Sigma - extr^{set}(\alpha\widehat{Q})) = \alpha P - extr^{set}(\alpha\widehat{Q})$, since $B_{inv} \subseteq extr^{set}(\alpha\widehat{Q})$ by proposition C.21. Thus, and by definition 6.14,

$$S = extrFDR^{ref}_{EP(Q)}(R, w, \widehat{Q}) \cup (\alpha P - extr^{set}(\alpha\widehat{Q})) \cup (\Sigma - ((\alpha P - B_{inv}) \cup d_{inv})).$$

Moreover, since $extr^{set}(\alpha\widehat{Q}) = A_{Fvis} \cup B_{inv}$ by proposition C.21, since $A_{Fvis} \cap B_{inv} = \varnothing$ and since $extr^{set}(\alpha\widehat{Q}) \subseteq \alpha P$,

$$(\alpha P - extr^{set}(\alpha\widehat{Q})) \cup (\Sigma - ((\alpha P - B_{inv}) \cup d_{inv})) = \Sigma - (A_{Fvis} \cup d_{inv}).$$

Hence, $S = extrFDR^{ref}_{EP(Q)}(R, w, \widehat{Q}) \cup (\Sigma - (A_{Fvis} \cup d_{inv}))$ and so the proof in this direction follows by SF3.

($\Longleftarrow$) We assume that *FinalImple* $\sqsupseteq_{SF}$ *NewSpec*. Thus, by lemmas 6.14(1) and 6.15(1), $extr_{EP(Q)}(\tau\widehat{Q}) \subseteq \tau P$. Hence, by SF-DEF1, it suffices to show that $extr_{EP(Q)}(\phi\widehat{Q}) \subseteq \phi P$.

Let $(t, R) \in \phi_{dom_{EP(Q)}}\widehat{Q}$, where $R \subseteq \alpha\widehat{Q}$. By SF-DEF2 and SF3, we show that $(extr_{EP(Q)}(t), X \cup Y) \in \phi P$, where $X = extr^{ref}_{EP(Q)}(R, t, \widehat{Q})$ and $Y = \Sigma - extr^{set}(\alpha\widehat{Q})$. By lemma 6.14(2) and since $\phi FinalImple \subseteq \phi NewSpec$, then $(extr_{EP(Q)}(t), V \cup W) \in \phi NewSpec$, where:

- $V = extrFDR^{ref}_{EP(Q)}(R, t, \widehat{Q})$.
- $W = \Sigma - (A_{Fvis} \cup d_{inv})$.

Thus, by lemma 6.15(2), there exists U such that $(extr_{EP(Q)}(t), U) \in \phi P$ and:

$$V \cup W \subseteq DB(U) \cup (U \cap (\alpha P - B_{inv})) \cup (\Sigma - ((\alpha P - B_{inv}) \cup d_{inv})). \quad (**)$$

We observe that $d_i \notin \alpha P$ for $i \in inv$ since all such d_i are "fresh" events. Hence, $\bigcup_{i \in inv} extrFDR^{ref}_i(R \cap A_i, t\lceil A_i, \widehat{Q}) \subseteq DB(U)$ by definitions 6.12, 6.14 and 6.16 and so, again by definitions 6.14 and 6.16,

$$\bigcup_{i \in inv} extr^{ref}_i(R \cap A_i, t\lceil A_i, \widehat{Q}) \subseteq U.$$

By proposition C.21, since $extr^{set}(\alpha\widehat{Q}) \subseteq \alpha P$ and since $A_{Fvis} \cap B_{inv} = \varnothing$, then $A_{Fvis} \subseteq (\alpha P - B_{inv})$. Hence, by definitions 6.14(1) and 6.15,

$$\bigcup_{i \notin inv} extrFDR_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) \subseteq (\alpha P - B_{inv})$$

and so, by definitions 6.14 and 6.16, and (**),

$$\bigcup_{i \notin inv} extrFDR_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) \subseteq (U \cap (\alpha P - B_{inv})).$$

Thus, by definition 6.14(1) and EP5-FVI,

$$\bigcup_{i \notin inv} extr_i^{ref}(R \cap A_i, t\lceil A_i, \widehat{Q}) \subseteq U.$$

Hence, by SF-GLOBAL2, $extr_{EP(Q)}^{ref}(R, t, \widehat{Q}) \subseteq U$.

By definition 6.16, we observe that $DB(U) \subseteq d_{inv}$. Thus, by (**) and since $(\Sigma - (A_{Fvis} \cup d_{inv})) \cap d_{inv} = \varnothing$,

$$(\Sigma - (A_{Fvis} \cup d_{inv})) \subseteq (U \cap (\alpha P - B_{inv})) \cup (\Sigma - ((\alpha P - B_{inv}) \cup d_{inv})).$$

If we subtract B_{inv} from both sides, we have, by proposition C.21 and since $extr^{set}(\alpha\widehat{Q}) \subseteq \alpha P$:

$$(\Sigma - (extr^{set}(\alpha\widehat{Q}) \cup d_{inv})) \subseteq (U \cap (\alpha P - B_{inv})) \cup (\Sigma - (\alpha P \cup d_{inv})).$$

If we then add d_{inv} to both sides, we have that:

$$(\Sigma - extr^{set}(\alpha\widehat{Q})) \subseteq (U \cap (\alpha P - B_{inv})) \cup (\Sigma - \alpha P).$$

Hence, $(\Sigma - extr^{set}(\alpha\widehat{Q})) \subseteq U \cup (\Sigma - \alpha P)$. By PA2,

$$(extr_{EP(Q)}(t), U \cup (\Sigma - \alpha P)) \in \phi P$$

and so, by SF3,

$$(extr_{EP(Q)}(t), extr_{EP(Q)}^{ref}(R, t, \widehat{Q}) \cup (\Sigma - extr^{set}(\alpha\widehat{Q}))) \in \phi P.$$

□

C.5 Proofs from section 6.7

Proof of theorem 6.17

Proof. We first observe that TE_i is guarded and so $\delta TE_i = \varnothing$ by DF.

($\Longrightarrow$) We assume that ep_i meets EP6 and proceed by assuming that $\delta(TE_i \backslash A_i) \neq \varnothing$. Thus, since $\delta TE_i = \varnothing$, there exists $w \in \Sigma^\omega$ such that $w \backslash A_i$ is finite and, for every $v < w$, $v \in \tau_\perp TE_i$. Hence, $v \in \tau TE_i$ for every $v < w$ by DR1. Thus, the sequence of $domain(v)$ for $v < w$ is an ω-sequence in Dom_i by proposition C.4(1). But, by proposition C.4(3) and the fact that $w \backslash A_i$ is finite, the sequence of $extr_i(domain(v))$ for $v < w$ is *not* an ω-sequence and so we have a contradiction.

($\Longleftarrow$) We assume that $\delta(TE_i \backslash A_i) = \varnothing$. Let $\ldots, t_j, \ldots$ be an ω-sequence in Dom_i. Let $v, w \in Dom_i$ and, by proposition C.4(2), let $x, y \in \tau TE_i$ be such that $domain(x) = v$ and $domain(y) = w$. If $v \leq w$ then $x \leq y$ by the definition of TE_i and definitions 6.4 and 6.7(1). Thus, there exists an ω-sequence $\ldots, u_j, \ldots$ in τTE_i, where $domain(u_j) = t_j$ for each u_j. We proceed by assuming that $\ldots, extr_i(domain(u_j)), \ldots$ is *not* an ω-sequence. Hence, by proposition C.4(3), $\ldots, extract(u_j \backslash A_i), \ldots$ is not an ω-sequence and so $\ldots, u_j \backslash A_i, \ldots$ is not an ω-sequence. Thus, since $\ldots, u_j, \ldots$ is an ω-sequence in $\tau TE_i = \tau_\perp TE_i$ by DR1, then $\delta(TE_i \backslash A_i) \neq \varnothing$ and so we have a contradiction. □

Proof of theorem 6.18

Proof. We first observe that $\phi_\perp P = \phi P$ by DR2 and since $\delta P = \varnothing$; moreover, $\tau P = \tau_\perp P$ by DR1.

($\Longrightarrow$) We assume that $extr_{EP(Q)}([\![\widehat{Q}]\!]_{FD}) \subseteq [\![P]\!]_{FD}$. Thus, by FD-DEF1, $extr_{EP(Q)}(\phi_\perp \widehat{Q}) \subseteq \phi_\perp P = \phi P$ and $extr_{EP(Q)}(\delta \widehat{Q}) \subseteq \delta P = \varnothing$. Hence, by FD-DEF3, $extr_{EP(Q)}(\phi \widehat{Q}) \subseteq \phi P$. Moreover, by FD-DEF2 and proposition 6.4(3), $min\delta\widehat{Q} = \varnothing$. Thus, $\delta\widehat{Q} = \varnothing$ by definition 2.2. Since $\widehat{Q}$ meets conditions Dom-T-check and Dom-SF-check, then $\widehat{Q} \sqsupseteq_{FD}^{EP(Q)} P$ by FD-DEF4. Hence, by propositions 6.4(1) and B.25, $extr_{EP(Q)}(t) \in \tau_\perp P = \tau P$ for every $t \in \tau\widehat{Q}$ and so $extr_{EP(Q)}(\tau\widehat{Q}) \subseteq \tau P$ by TR-DEF1. Thus, $extr_{EP(Q)}([\![\widehat{Q}]\!]_{SF}) \subseteq [\![P]\!]_{SF}$ by SF-DEF1 and the proof in this direction follows by theorem 6.16.

($\Longleftarrow$) We assume that *FinalImple* $\sqsupseteq_{SF}$ *NewSpec* and $\delta\widehat{Q} = \varnothing$. Thus, by definition 2.2 and FD-DEF2, $extr_{EP(Q)}(\delta\widehat{Q}) = \varnothing \subseteq \delta P$. Hence, by FD-DEF3, $extr_{EP(Q)}(\phi_\perp\widehat{Q}) = extr_{EP(Q)}(\phi\widehat{Q})$. Since *FinalImple* $\sqsupseteq_{SF}$ *NewSpec* then, by theorem 6.16 and SF-DEF1,

$$extr_{EP(Q)}(\phi_\perp\widehat{Q}) = extr_{EP(Q)}(\phi\widehat{Q}) \subseteq \phi P = \phi_\perp P.$$

Thus, the proof in this direction follows by FD-DEF1. □

Appendix D

Processes used in verification from chapter 7

The following channels are needed here.[1]

- *channel extractWriteSlot* : *slots.slots.dataint*
- *channel extractWriteSlotRead* : *slots.slots.dataint*
- *channel extractWriteLatest* : *slots.slots.dataint*
- *channel extractReadLatest* : *slots.slots.dataint*
- *channel extractReadReading* : *slots.slots.dataint*
- *channel extractReadSlot* : *slots.slots.dataint*
- *channel extra* : *dataint*

Those channels containing *Write* as a substring of their identifier are used in the extraction of write events. *extractWriteSlot* is used when we extract to a single write event on the occurrence of *slot.writer.x.wr.y*. We then observe that *extractWriteSlotRead* is used when we extract to *both* a read *and* a write event on the occurrence of *slot.writer.x.wr.y*. *extractWriteLatest* is used when we extract on the occurrence of *latest.wr.x*. Those channels containing *Read* as a substring of their identifier are used in the extraction

[1]Disregarding *extra* for the moment, these channels are used to represent "pairs of events" which will be used in the extraction of the events of *FSlot*. However, the respective *types* of these channels do not give sufficient information to allow us, as described in chapter 6.4, to reclaim the name of the channel of the right-hand event of the pair: i.e the channel on which the (specification) event being extracted to will occur. This is acceptable since the new channel names themselves give us all the information we need: see the definition of $extract_{ar}$ below.

of read events. The implementation events which they are used to extract are immediate from their respective identifiers. The channel *extra* is used when we must extract to a write event and a read event together.

Four processes are composed in parallel — with each individual composition synchronizing on common events — to define the process TE_{ar} used in the extraction of the traces of *FSlot*. These are *WrExt*, *RdExt*, *EDATA* and *SlotCopy* (they are composed in that order). *EDATA* is given in figure D.1 and is a copy of *Data* from figure 7.6. *SlotCopy* is given in figure D.2 and is a copy of the process *Slots* from figure 7.5. These two processes are needed for the following reason. According to the definition of $extr_{ar}$ from figure 7.12, we never extract on the occurrence of an actual data transfer event: when we do extract, we therefore need some means of discovering the data value which was transferred (in the case of write events) or will be transferred (in the case of read events). This is the purpose of the process *EDATA*. In order to use *EDATA* to find the relevant data value, it is necessary to know the pair and slot — i.e. the position in the 2-dimensional data array — at which it is to be found. On the reader side, although we always know the value of the necessary pair by the time that we must extract, we do not always know the value of the slot within that pair: *SlotCopy* is used to help us find it. More detail on how exactly these two processes are used may be found in section D.1 below, after all necessary processes have been presented.

The processes which are directly responsible for extracting events are *WrExt* and *RdExt*. They are each presented over two figures because of the length of their respective descriptions and also because this split helps partition the clauses of each process in a useful way. *WrExt*, given in figures D.3 and D.4, is used to extract write events. *RdExt*, given in figures D.5 and D.6, is used to extract read events.

WrExt has a number of parameters, corresponding to the variables used in the definition of $extr_{ar}$ in figure 7.12. They are explained as follows:

- *wrp* — The value of *pair* in the writer.
- *wrs* — The value of *index* in the writer (i.e. the value of the slot into which we shall write or into which we have just written).
- *late* — The value stored by the variable *latest*.
- *wrExtract* — A variable to indicate whether or not we have extracted yet on the current call to write.
- *rp* — The value of *pair* in the reader.
- *readingVal* — The value stored by the variable *reading*.

- *ExtractData(x,y,V)=*
```
  let ED(v)=
    data.wr.x.y?val→ED(val)
    □
    extractWriteSlot.x.y.v→ED(v)
    □
    extractWriteSlotRead.x.y.v→ED(v)
    □
    extractWriteLatest.x.y.v→ED(v)
    □
    extractReadLatest.x.y.v→ED(v)
    □
    extractReadSlot.x.y.v→ED(v)
    □
    extractReadReading.x.y.v→ED(v)
  within ED(V).
```

- *EDATA* $=|||_{x\in\mathcal{A}}$*ExtractData(fst(x),sec(x),0)*, where $\mathcal{A}=\{(\mathit{first},\mathit{first}),(\mathit{first},\mathit{second}),(\mathit{second},\mathit{first}),(\mathit{second},\mathit{second})\}$.

Figure D.1: A copy of the data array

- *SC(x,Y)* =
```
  let SY(y)=
    slot.writer.x.wr?new→SY(new)
    □
    extractWriteSlot.x?new?val→SY(new)
    □
    extractWriteSlotRead.x?new?val→SY(new)
    □
    extractWriteLatest.x.y?val→SY(y)
    □
    extractReadLatest.x.y?val→SY(y)
    □
    extractReadReading.x.y?val→SY(y)
  within SY(Y).
```

- *SlotCopy* $=|||_{x\in\{\mathit{first},\mathit{second}\}}$*SC(x,first)*.

Figure D.2: A copy of *Slots*

- *WrExt = WE(first,first,first,no,first,first,no,1).*
- *WE(wrp,wrs,late,wrExtract,rp,readingVal,rdExtract,posn) =*

  latest.rd?val $\rightarrow$
  WE(wrp,wrs,late,wrExtract,val,readingVal,rdExtract,2)
  $\square$
  extractReadLatest?x?y?val $\rightarrow$
  WE(wrp,wrs,late,wrExtract,x,readingVal,yes,2)
  $\square$
  reading.wr?val $\rightarrow$ *WE(wrp,wrs,late,wrExtract,rp,val,rdExtract,3)*
  $\square$
  extractReadReading?x?y?val $\rightarrow$
  WE(wrp,wrs,late,wrExtract,rp,x,yes,3)
  $\square$
  slot.reader?x!rd?y $\rightarrow$
  WE(wrp,wrs,late,wrExtract,rp,readingVal,rdExtract,4)
  $\square$
  extractReadSlot?x?y?val $\rightarrow$
  WE(wrp,wrs,late,wrExtract,rp,readingVal,yes,4)
  $\square$
  data.rd?x?y?val $\rightarrow$ *WE(wrp,wrs,late,wrExtract,rp,readingVal,no,1)*
  $\square$
  reading.rd?p $\rightarrow$
  WE(not(p),wrs,late,wrExtract,rp,readingVal,rdExtract,posn)
  $\square$
  slot.writer?x!rd?s $\rightarrow$
  WE(wrp,not(s),late,wrExtract,rp,readingVal,rdExtract,posn)
  $\square$
  data.wr?x?y?val $\rightarrow$
  WE(wrp,wrs,late,wrExtract,rp,readingVal,rdExtract,posn)
  $\square$
  .
  .
  .
  .

Figure D.3: Process used to extract write events - part 1

```
.
.
.
.
(if (rdExtract == yes and wrp == late) then
   (extractWriteSlot?x?y?val→
   WE(wrp,wrs,late,yes,rp,readingVal,rdExtract,posn))
else
   (if (posn == 1 and wrp == late) then
      (extractWriteSlot?x?y?val→
      WE(wrp,wrs,late,yes,rp,readingVal,rdExtract,posn))
   else
      (if ((posn == 2 or posn == 3) and rp == wrp) then
         (if (rp == readingVal) then
            (extractWriteSlotRead?x?y?val →extra.val→
            WE(wrp,wrs,late,yes,rp,readingVal,yes,posn))
         else
            (extractWriteSlot?x?y?val→
            WE(wrp,wrs,late,yes,rp,readingVal,rdExtract,posn)))
      else
         (slot.writer?x!wr?y→
         WE(wrp,wrs,late,wrExtract,rp,readingVal,rdExtract,posn)))))

□

(if wrExtract == no then
   (extractWriteLatest?x?y?val→
   WE(wrp,wrs,x,wrExtract,rp,readingVal,rdExtract,posn))
else
   (latest.wr?x→ WE(wrp,wrs,x,no,rp,readingVal,rdExtract,posn)))
```

Figure D.4: Process used to extract write events - part 2

- *rdExtract* — A variable indicating whether or not we have extracted yet on the current call to read.
- *posn* — The current position of the reader.

The part of *WrExt* which is given in figure D.3 is simply used to update these parameters (*data.wr?x?y?val* is included for ease of defining synchronization with *RdExt*). The part of *WrExt* given in figure D.4 actually deals with the extraction proper and is related directly to the detail given in figure 7.12.[2] In general, we offer the "extracted" version of an implementation event a — i.e. an "event pair" where a is the left-hand component and the right-hand component is non-null — if the necessary conditions are met; otherwise, we offer a itself. If the original event a is offered, it indicates that its occurrence at this point does not cause extraction to a high-level write event (recall that a will be hidden in the construction of the final implementation process to be supplied as input to FDR2). Note also that *extractWriteSlotRead?x?y?val* followed by *extra.val* (from figure D.4) is used when we must extract to both a read event and a write event on the occurrence of *slot.writer.not(p).wr.not(i)*. The event occurring on *extractWriteSlotRead* is renamed to the write event and that occurring on *extra* is renamed to the read event; this uses the renaming $extract_{ar}$ defined below. (When an event occurs on channel *extra*, it effectively represents an "event pair" with a null *left-hand* — i.e. implementation — component but with a non-null right-hand component.)

Process *RdExt* is similar in concept to *WrExt*; its parameters are explained as follows:

- *rp* — The value of *pair* in the reader.
- *rdng* — The value of the variable *reading.*
- *wrp* — The value of *pair* in the writer.
- *late* — The value of the variable *latest.*
- *posn* — The position of the writer.
- *extract* — This indicates whether or not we have extracted yet on the current call to read.

[2]Note that, in figure 7.12, *wVal* is used to give directly the value of the last data item written into the mechanism. Using that approach here would have required an extra parameter for *WrExt* and so we instead use the process *EDATA* — needed anyway for the extraction of read events — to provide the necessary information.

- *RdExt = RE(first,first,first,first,1,no).*
- *RE(rp,rdng,wrp,late,posn,extract) =*

  reading.rd?p→RE(rp,rdng,not(p),late,2,extract)
  □
  slot.writer?x!rd?s→RE(rp,rdng,wrp,late,3,extract)
  □
  data.wr?x?y?val→RE(rp,rdng,wrp,late,4,extract)
  □
  extractWriteSlot?x?y?val→RE(rp,rdng,wrp,late,5,extract)
  □
  extractWriteSlotRead?x?y?val→extra.val→
  RE(rp,rdng,wrp,late,5,yes)
  □
  slot.writer?x!wr?y→ RE(rp,rdng,wrp,late,5,extract)
  □
  extractWriteLatest?x?y?val→ RE(rp,rdng,wrp,x,1,extract)
  □
  latest.wr?val→ RE(rp,rdng,wrp,val,1,extract)
  □
  data.rd?x?y?val→ RE(rp,rdng,wrp,late,posn,no)
  □
  .
  .
  .
  .

Figure D.5: Process used to extract read events - part 1

```
.
.
.
.
(if ((posn == 1 or posn == 5) and (late == rdng)) then
   (extractReadLatest?x?y?val→ RE(x,rdng,wrp,late,posn,yes))
else
   (if ((posn == 2 or posn == 3 or posn == 4)
      and (wrp != late) and (late == rdng)) then
      (extractReadLatest?x?y?val→ RE(x,rdng,wrp,late,posn,yes))
   else
      (latest.rd?p→ RE(p,rdng,wrp,late,posn,extract))))

□

(if ((extract == no) and (posn == 1 or posn == 5 or rp != wrp)) then
   (extractReadReading?x?y?val→ RE(rp,x,wrp,late,posn,yes))
else
   (reading.wr?val→ RE(rp,val,wrp,late,posn,extract)))

□

(if extract == no then
   (extractReadSlot?x?y?val→RE(rp,rdng,wrp,late,posn,yes))
else
   (slot.reader?x!rd?y→RE(rp,rdng,wrp,late,posn,extract)))
```

Figure D.6: Process used to extract read events - part 2

We define $prep_{ar} \triangleq \bigcup_{1 \leq i \leq 10} prep_i^{ar}$, where:

- $prep_1^{ar} \triangleq \{(slot.writer.x.wr.y, slot.writer.x.wr.y) \mid slot.writer.x.wr.y \in \alpha slot.writer\}$.
- $prep_2^{ar} \triangleq \{(slot.writer.x.wr.y, extractWriteSlot.x.y.z) \mid slot.writer.x.wr.y \in \alpha slot.writer \ \wedge \ z \in dataint\}$.
- $prep_3^{ar} \triangleq \{(slot.writer.x.wr.y, extractWriteSlotRead.x.y.z) \mid slot.writer.x.wr.y \in \alpha slot.writer \ \wedge \ z \in dataint\}$.
- $prep_4^{ar} \triangleq \{(latest.x.y, latest.x.y) \mid latest.x.y \in \alpha latest\}$.
- $prep_5^{ar} \triangleq \{(latest.wr.x, extractWriteLatest.x.y.z) \mid latest.wr.x \in \alpha latest.wr \ \wedge \ y \in \{first, second\} \wedge z \in dataint\}$.
- $prep_6^{ar} \triangleq \{(latest.rd.x, extractReadLatest.x.y.z) \mid latest.rd.x \in \alpha latest.rd \ \wedge \ y \in \{first, second\} \wedge z \in dataint\}$.
- $prep_7^{ar} \triangleq \{(reading.wr.x, reading.wr.x) \mid reading.wr.x \in \alpha reading.wr\}$.
- $prep_8^{ar} \triangleq \{(reading.wr.x, extractReadReading.x.y.z) \mid reading.wr.x \in \alpha reading.wr \ \wedge \ y \in \{first, second\} \wedge z \in dataint\}$.
- $prep_9^{ar} \triangleq \{(slot.reader.x.rd.y, slot.reader.x.rd.y) \mid slot.reader.x.rd.y \in \alpha slot\}$.
- $prep_{10}^{ar} \triangleq \{(slot.reader.x.rd.y, extractReadSlot.x.y.z) \mid slot.reader.x.rd.y \in \alpha slot \ \wedge \ z \in dataint\}$.

Figure D.7: Defining $prep_{ar}$

We define $extract_{ar} \triangleq \bigcup_{1 \leq i \leq 7} extract_i^{ar}$, where:

- $extract_1^{ar} \triangleq \{(extractWriteSlot.x.y.z, write.z) \mid$
  $extractWriteSlot.x.y.z \in \alpha extractWriteSlot\}$.
- $extract_2^{ar} \triangleq \{(extractWriteSlotRead.x.y.z, write.z) \mid$
  $extractWriteSlotRead.x.y.z \in \alpha extractWriteSlotRead\}$.
- $extract_3^{ar} \triangleq \{(extractWriteLatest.x.y.z, write.z) \mid$
  $extractWriteLatest.x.y.z \in \alpha extractWriteLatest\}$.
- $extract_4^{ar} \triangleq \{(extractReadLatest.x.y.z, read.z) \mid$
  $extractReadLatest.x.y.z \in \alpha extractReadLatest\}$.
- $extract_5^{ar} \triangleq \{(extractReadReading.x.y.z, read.z) \mid$
  $extractReadReading.x.y.z \in \alpha extractReadReading\}$.
- $extract_6^{ar} \triangleq \{(extractReadSlot.x.y.z, read.z) \mid$
  $extractReadSlot.x.y.z \in \alpha extractReadSlot\}$.
- $extract_7^{ar} \triangleq \{(extra.z, read.z) \mid extra.z \in \alpha extra\}$.

Figure D.8: Defining $extract_{ar}$

We define $domain_{ar} \triangleq \bigcup_{1 \leq i \leq 6} domain_i^{ar}$, where:

- $domain_1^{ar} \triangleq \{(extractWriteSlot.x.y.z, slot.writer.x.wr.y) \mid extractWriteSlot.x.y.z \in \alpha extractWriteSlot\}$.
- $domain_2^{ar} \triangleq \{(extractWriteSlotRead.x.y.z, slot.writer.x.wr.y) \mid extractWriteSlotRead.x.y.z \in \alpha extractWriteSlotRead\}$.
- $domain_3^{ar} \triangleq \{(extractWriteLatest.x.y.z, latest.wr.x) \mid extractWriteLatest.x.y.z \in \alpha extractWriteLatest\}$.
- $domain_4^{ar} \triangleq \{(extractReadLatest.x.y.z, latest.rd.x) \mid extractReadLatest.x.y.z \in \alpha extractReadLatest\}$.
- $domain_5^{ar} \triangleq \{(extractReadReading.x.y.z, reading.wr.x) \mid extractReadReading.x.y.z \in \alpha extractReadReading\}$.
- $domain_6^{ar} \triangleq \{(extractReadSlot.x.y.z, slot.reader.x.rd.y) \mid extractReadSlot.x.y.z \in \alpha extractReadSlot\}$.

Figure D.9: Defining $domain_{ar}$

The renaming $prep_{ar}$ used here is given in figure D.7; the renaming $extract_{ar}$ is defined in figure D.8.[3] Each of the channels used for the extraction of events (apart from *extra*) has three data fields, denoted x, y and z. Fields x and y are used to indicate the pair and slot combination in the data array at which the data value to be extracted is stored: these fields are used in the synchronization with *EDATA*. z denotes the data value itself and so $extract_{ar}$ simply renames to *read.z* or *write.z* as appropriate. Finally, the renaming $domain_{ar}$ is defined in figure D.9: it may be used to reclaim the domain of the mapping which is represented by TE_{ar}.

D.1 Explaining TE_{ar} further

Further comment is necessary on the role played by *EDATA* and *SlotCopy* and the way they interact with the various channels used for extraction: i.e. those whose identifier begins with the string "extract". We use the extraction to a read event on the occurrence of *latest.rd.p* as an example. According to the renaming $prep_{ar}$, this event (in the implementation) will be renamed to

[3]Those events on the occurrence of which we never extract to a high-level read or write event, such as those on channel *data*, need only be renamed to themselves by $prep_{ar}$ and so are omitted from its definition.

the set of *extractReadLatest.p.y.z* where $y \in \{\mathit{first}, \mathit{second}\}$ and $z \in$ ***dataint***. If a synchronization occurs between such an event *extractReadLatest.p.y.z* in the modified implementation and the same event in the process TE_{ar}, we know that we are to extract on the occurrence of *latest.rd.p*. More than that, however, we know that y gives us the value of *slot*[*p*], because synchronization with TE_{ar} means that we have effectively synchronized with *SlotCopy* as well. This means that the reader, on this call to read, will read the data value stored at position (p, y) in the data array. In fact, z gives us the data value currently stored in that location because we have effectively synchronized with *EDATA* as well. Extraction on the occurrence of other events works in a similar way. As a result, once we have renamed the implementation process and composed it with TE_{ar}, the events of the resulting process contain sufficient information that we may carry out extraction using only hiding and renaming.

Appendix E

Lists of conditions, notations and processes

E.1 General notation

In the following, $t, u, t_1, t_2, \ldots$ are traces; A is a set of actions; $\mathcal{T}, \mathcal{T}'$ are non-empty sets of traces; $G \subseteq \Sigma \times \Sigma$ is a relation; and $\mathcal{X}$ is a set of sets. Note that traces are assumed to be finite unless otherwise stated.

- $t = \langle a_1, \ldots, a_n \rangle$ is the trace whose i-th element is action a_i, and length, $|t|$, is n. Moreover, *events*$(t) \triangleq \{a_1, \ldots, a_n\}$ and, provided that $n \geq 1$, *tail*$(t) \triangleq a_n$. If $n = 0$ then t is the empty trace, denoted $\langle\,\rangle$.
- $|A|$ denotes the cardinality of A.
- $t \circ u$ is the trace obtained by appending u to t.
- A^* is the set of all traces — i.e. sequences — of actions from A, including the empty trace, $\langle\,\rangle$.
- A^ω is the set of all *infinite* traces of actions from A.
- $\mathcal{T}^*$ is the set of all traces $t = t_1 \circ \cdots \circ t_n$ $(n \geq 0)$ such that $t_1, \ldots, t_n \in \mathcal{T}$ (note that $t = \langle\,\rangle$ when $n = 0$).
- $\leq$ denotes the prefix relation on traces, and $t < u$ if $t \leq u$ and $t \neq u$.
- *Pref*$(\mathcal{T}) \triangleq \{u \mid (\exists t \in \mathcal{T})\ u \leq t\}$ is the *prefix-closure* of $\mathcal{T}$. (In the event that $\mathcal{T}$ is the singleton set $\{t\}$, we may use *Pref*(t) in lieu of *Pref*$(\mathcal{T})$.)
- $\mathcal{T}$ is *prefix-closed* if $\mathcal{T} =$ *Pref*$(\mathcal{T})$.

- $t \lceil A$ is a trace obtained by deleting from t all the actions that do not occur in A.
- $t \setminus A$ is a trace obtained by deleting from t all the actions that *do* occur in A.
- The definitions of $\lceil$ and $\setminus$ may be lifted to *sets* of traces in the obvious way: $\mathcal{T} \lceil A \triangleq \{t \lceil A \mid t \in \mathcal{T}\}$ and $\mathcal{T} \setminus A \triangleq \{t \setminus A \mid t \in \mathcal{T}\}$.
- $t_1, t_2, \ldots$ is an *ω-sequence* of traces if $t_1 \leq t_2 \leq \ldots$ and $\lim_{i \to \infty} |t_i| = \infty$.
- A mapping $f : \mathcal{T} \to \mathcal{T}'$ is *monotonic* if $t, u \in \mathcal{T}$ and $t \leq u$ implies $f(t) \leq f(u)$, and *strict* if $\langle\rangle \in \mathcal{T}$ and $f(\langle\rangle) = \langle\rangle$.
- The definition of G may be lifted to sets of events, traces and sets of traces:

  - $G(A) \triangleq \bigcup\{G(a) \mid a \in A\}$.
  - $\langle a_1, \ldots, a_n \rangle \; G \; \langle b_1, \ldots, b_m \rangle \Leftrightarrow n = m \wedge \forall i \leq n, \; a_i \; G \; b_i$.
  - $G(\mathcal{T}) \triangleq \{u \mid (\exists t \in \mathcal{T}) \; t \; G \; u\}$.

  In the event that $\mathcal{T}$ is the singleton set $\{t\}$, we may use $G(t)$ in lieu of $G(\mathcal{T})$. Moreover, if $G(t) = \{u\}$ for some trace u then we shall denote this $G(t) = u$. Similarly, if $G(a) = \{b\}$ for some action a then we write $G(a) = b$.

- $G^{-1} \triangleq \{(b, a) \mid a \; G \; b\}$ is the *inverse* of G.
- $\mathit{Sub}(\mathcal{X}) \triangleq \{W \subseteq X \mid X \in \mathcal{X}\}$ is the *subset-closure* of $\mathcal{X}$.
- $\mathcal{X}$ is *subset-closed* if $\mathcal{X} = \mathit{Sub}(\mathcal{X})$.
- $2^S \triangleq \{X \mid X \subseteq S\}$ gives the *power set* of S. For purposes of presentation, we will sometimes use $\mathbb{P}(S)$ in lieu of 2^S.
- We introduce containment and equality between *pairs* of sets in the obvious way. Let B, B', C, C' be sets.

  - $(B, C) \subseteq (B', C')$ if and only if $B \subseteq B'$ and $C \subseteq C'$.
  - $(B, C) = (B', C')$ if and only if $B = B'$ and $C = C'$.

- For an arbitrary set of objects O and a partial ordering[1] $\preceq$ over the elements of O,

$$\mathit{max}_{\preceq}(O) \triangleq \{e \in O \mid (\nexists d \in O) \; e \preceq d \; \wedge \; e \neq d\}.$$

[1]A partial ordering is reflexive, transitive and antisymmetric.

In the event that $max_{\preceq}(O) = \{e\}$ for some element e, we shall write $max_{\preceq}(O) = e$.

- $max(\mathcal{F}) = \{(t, R) \in \mathcal{F} \mid (t, R)$ is maximal $\}$ for a set of failures $\mathcal{F}$.
- For $X \subseteq \Sigma_{impl}$, $[[X]]$ denotes the smallest set $A \in AllSet$ such that $X \subseteq A$.

E.2 List of labelled conditions and definitions

DE	page 35
DF	page 28
DIS	page 131
DOM-SF-CHECK	page 98
DOM-T-CHECK	page 90
DR1-3	page 29
EP1	page 81
EP1-FVI	page 81
EP2	page 81
EP3A-FVI	page 93
EP3-FVI	page 81
EP3-SF	page 93
EP3-T	page 81
EP4	page 81
EP4-FVI	page 81
EP5	page 93
EP5-FVI	page 96
EP6	page 101
EP-UNI1-2	page 83
FD1-5	page 24
FD-DEF1-4	page 102
FDI	page 74

E.3 List of processes

E.4 Notation from chapter 3

E.5 Notation from concrete notion of refinement-after-hiding

$\tau_{Dom_{EP(Q)}}Q$	page 91
$\phi_{dom_{EP(Q)}}Q$	page 98
$\phi_{Dom_{EP(Q)}}Q$	page 98
A	page 81
B	page 81
$Comm(A, Q)$	page 86
dom	page 93
Dom	page 81
$dom_{EP(Q)}$	page 97
$Dom_{EP(Q)}$	page 85
EP	page 83
$EP(Q)$	page 85
$extr$	page 81
$extr^{ref}$	page 93
$extr^{ref}_{EP(Q)}$	page 97
$extr^{set}$	page 83
F_{impl}	page 84
F_{spec}	page 84
$ImplSet$	page 83
$Proj_{EP(Q)}$	page 89
ref	page 93
$\overline{ref}$	page 94

E.6 Notation from chapter 6

A_{Fvis}	page 145
A_{inv}	page 123
B_{inv}	page 123
d_{inv}	page 142
DB	page 146

E.7 Semantic notations

E.8 Operators

$STOP$	the immediately deadlocking process
DIV	the immediately diverging process
$a \rightarrow P$	the prefix operator
$P \,\Box\, Q$	deterministic choice
$P \sqcap Q$	non-deterministic choice
$P \setminus A$	hiding
$P \parallel_Y Q$	parallel composition
$P \otimes_Y Q$	network composition
$P[G]$	renaming

Bibliography

[1] M. Abadi and L. Lamport: The Existence of Refinement Mappings. *Theoretical Computer Science* 82 (1991) 253–284.

[2] P. Bernstein, V. Hadzilacos and N. Goodman: *Concurrency Control and Recovery in Database Systems.* Addison-Wesley (1987).

[3] E. A. Boiten and J. Derrick: IO - refinement in Z. Proc. of *3rd BCS-FACS Northern Formal Methods Workshop*, A. Evans, D. Duke and T. Clark (Eds.). Electronic Workshops in Computing, Springer Verlag (1998) .

[4] E. A. Boiten and J. Derrick: Liberating data refinement. Proc. of *Mathematics of Program Construction, 5th International Conference*, R. C. Backhouse and J. N. Oliveira (Eds.). LNCS 1837 (2000) 144–166.

[5] E. A. Boiten and J. Derrick: Unifying concurrent and relational refinement. Proc. of *REFINE 02: The BCS FACS Refinement Workshop*, J. Derrick, E. Boiten, J. Woodcock and JDT von Wright (Eds.). volume 70(3) of Electronic Notes in Theoretical Computer Science (2002) 38–75.

[6] E. Brinksma: A Theory for the Derivation of Tests. In: *Protocol Specification, Verification and Testing, VIII*, S. Aggarwal and K. Sabnani(Eds.). North-Holland (1988) 63–74.

[7] E. Brinksma, B. Jonsson and F. Orava: Refining Interfaces of Communicating Systems. Proc. of *TAPSOFT '91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Developmemnt (CCPSD)*, S. Abramsky and T. S. E. Maibaum (Eds.). LNCS 494 (1991) 297–312.

[8] S. D. Brookes, C. A. R. Hoare and A. W. Roscoe: A Theory of Communicating Sequential Process. *Journal of ACM* 31 (1984) 560–599.

[9] S. D. Brookes and A. W. Roscoe: An Improved Failures Model for Communicating Sequential Processes. Proc. of *Seminar on Concurrency*,

S. D. Brookes, A. W. Roscoe and G. Winskel (Eds.). Springer-Verlag, Lecture Notes in Computer Science 197 (1985) 281–305.

[10] J. Burton, M. Koutny and G. Pappalardo: Modelling and Verification of Communicating Processes in the Event of Interface Difference. Technical Report 696, Dept. of Computing Science, University of Newcastle upon Tyne (2000).

[11] J. Burton, M. Koutny and G. Pappalardo: Verifying Implementation Relations in the Event of Interface Difference. Proc. of *FME 2001: Formal Methods for Increasing Software Productivity*, J. N. Oliveira and P. Zave (Eds.). LNCS 2021 (2001) 364–383.

[12] J. Burton, M. Koutny and G. Pappalardo: Implementing Communicating Processes in the Event of Interface Difference. Proc. of *ACSD 2001: Second International Conference on Application of Concurrency to System Design*, A. Valmari and A. Yakovlev (Eds.). IEEE Computer Society (2001) 87–96.

[13] J. Burton, M. Koutny and G. Pappalardo: Compositional Verification of a Network of CSP Processes. Technical Report 757, Dept. of Computing Science, University of Newcastle upon Tyne (2002).

[14] J. Burton: Compositional Verification of a Network of CSP Processes: Using FDR2 to Verify Refinement in the Event of Interface Difference. Technical Report 758, Dept. of Computing Science, University of Newcastle upon Tyne (2002).

[15] J. Burton, M. Koutny, G. Pappalardo and M. Pietkiewicz-Koutny: Compositional Development in the Event of Interface Difference. In: *Concurrency in Dependable Computing*, P. Ezhilchelvan and A. Romanovsky (Eds.). Kluwer Academic Publishers (2002) 1–20.

[16] J. Burton, M. Koutny and G. Pappalardo: Relating Communicating Processes with Different Interfaces. *Fundamenta Informaticae* 59(1) (2004) 1–37.

[17] I. Clark: *A Unified Approach to the Study of Asynchronous Communication Mechanisms in Real Time Systems.* King's College, London University (PhD Thesis) (2000).

[18] P. Collette and C. B. Jones: Enhancing the Tractability of Rely/Guarantee Specifications in the Development of Interfering Operations. Technical Report CUMCS-95-10-3, Department of Computing Science, Manchester University (1995).

[19] J. Derrick and E. A. Boiten: Non-atomic refinement in Z. Proc. of *FM'99 World Congress on Formal Methods in the Development of Computing Systems*, J. M. Wing, J. C. P. Woodcock and J. Davies (Eds.). LNCS 1708 (1999) 1477–1496.

[20] J. Derrick and E. A. Boiten: Refinement of objects and operations in Object-Z. Proc. of *Formal Methods for Open Object-based Distributed Systems IV*, S. F. Smith and C. L. Talcott (Eds.). Kluwer Academic Publishers (2000) 257–277.

[21] J. Dingel: Systematic parallel programming (PhD thesis). Technical Report CMU-CS-99-172, School of Computer Science, Carnegie Mellon University (2000).

[22] W. de Roever and K. Engelhardt: *Data Refinement: Model-Oriented Proof Methods and their Comparison.* CUP (1998).

[23] W. de Roever *et al*: *Concurrency Verification: Introduction to Compositional and Noncompositional Methods.* Cambridge University Press (2001).

[24] R. Gerth, R. Kuiper and J. Segers: Interface Refinement in Reactive Systems (Extended Abstract). Proc. of *CONCUR '92*, W. R. Cleaveland (Ed.). LNCS 630 (1992) 77–93.

[25] R. Gorrieri and A. Rensink: Action Refinement. In: *Handbook of Process Algebra*, J. A. Bergstra, A. Ponse and S. A. Smolka (Eds.). Elsevier (2001) 1047–1147.

[26] N. Henderson and S. Paynter: The Formal Classification and Verification of Simpson's 4-Slot Asynchronous Communication Mechanism. Proc. of *FME 2002: Formal Methods - Getting IT Right*, L-H. Eriksson and P. Lindsay (Eds.). LNCS 2391 (2002) 350–369.

[27] N. Henderson: Proving the Correctness of Simpson's 4-slot ACM Using an Assertional Rely-Guarantee Proof Method. Technical Report 800, School of Computing Science, University of Newcastle upon Tyne (2003).

[28] M. C. B. Hennessy: *Algebraic Theory of Processes.* MIT Press (1988).

[29] M. Herlihy and J. Wing: Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12(3) (1990) 463–492.

[30] C. A. R. Hoare: Proof of Correctness of Data Representations. *Acta Informatica* 1 (1972) 271–281.

[31] C. A. R. Hoare: *Communicating Sequential Processes.* Prentice Hall (1985).

[32] W. Janssen, M. Poel and J. Zwiers: Action Systems and Action Refinement in the Development of Parallel Systems - An Algebraic Approach. Proc. of *CONCUR '91*, J. C. M. Baeten and J. F. Groote (Eds.). LNCS 527 (1991) 298–316.

[33] C. B. Jones: *Systematic Software Development Using VDM.* Prentice Hall (1990).

[34] C. B. Jones: An Object-Based Design Method for Concurrent Programs. Technical Report UMCS-92-12-1, Department of Computer Science, University of Manchester (1992).

[35] C. B. Jones: Constraining Interference in an Object-Based Design Method. Proc. of *TAPSOFT '93: Theory and Practice of Software Development*, M-C. Gaudel and J-P. Jouannaud (Eds.). LNCS 668 (1993) 136–150.

[36] C. B. Jones: Process-Algebraic Foundations for an Object-Based Design Notation. Technical Report UMCS-93-10-1, Department of Computer Science, University of Manchester (1993).

[37] C. B. Jones: Reasoning about Interference in an Object-Based Design Method. Proc. of *FME '93: Industrial Strength Formal Methods*, J. Woodcock and P. Larsen (Eds.). LNCS 670 (1993) 1–18.

[38] B. Jonsson: Compositional Specification and Verification of Distributed Systems. *ACM TOPLAS* 16 (1994) 259–303.

[39] M. Koutny, L. Mancini and G. Pappalardo: Two Implementation Relations and the Correctness of Communicating Replicated Processes. *Formal Aspects of Computing* 9 (1997) 119–148.

[40] M. Koutny and G. Pappalardo: Behaviour Abstraction for Communicating Sequential Processes. *Fundamenta Informaticae* 48 (2001) 21–54.

[41] L. Lamport: Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering SE-3* 2 (1977) 125–143.

[42] L. Lamport: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers* 28(9) (1979) 690–691.

[43] L. Lamport: On interprocess communication: Part I - Basic formalism. *Distributed Computing* 1 (1986) 77–85.

[44] L. Lamport: On interprocess communication: Part II - Algorithms. *Distributed Computing* 1 (1986) 86–101.

[45] K. G. Larsen: A context dependent equivalence between processes. *Theoretical Computer Science* 49(2) (1987) 185–215.

[46] R. Lazic: *A Semantic Study of Data Independence with Applications to Model Checking.* Oxford University Computing Laboratory (PhD Thesis) (1999).

[47] N. A. Lynch and M. R. Tuttle: Hierarchical Correctness Proofs for Distributed Algorithms. Proc. of *6th ACM Symposium on Principles of Distributed Computing,* ACM (1987) 137–151.

[48] N. Lynch, M. Merritt, W. Weihl and A. Fekete: *Atomic Transactions.* Morgan Kaufmann (1994).

[49] L. V. Mancini and G. Pappalardo: Towards a Theory of Replicated Processing. Proc. of *Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems,* M. Joseph (Ed.). LNCS 331 (1988) 175–192.

[50] R. Milner: *Communication and Concurrency.* Prentice Hall (1989).

[51] R. Milner and D. Sangiorgi: Barbed Bisimulation. Proc. of *19th International Colloquium on Automata, Languages and Programming (ICALP '92),* W. Kuich(Ed.). LNCS 623 (1992) 685–695.

[52] J. Parrow and P. Sjoedin: Multiway Synchronization Verified with Coupled Simulation. Proc. of *CONCUR '92,* W. R. Cleaveland(Ed.). LNCS 630 (1992) 518–533.

[53] J. Parrow and P. Sjoedin: The Complete Axiomatization of Cs-Congruence. Proc. of *11th Annual Symposium on Theoretical Aspects of Computer Science, STACS '94,* R. Enjalbert, E. Mayr and K. Wagner(Eds.). LNCS 775 (1994) 557–568.

[54] S. E. Paynter, N. Henderson and J. M. Armstrong: Ramifications of Metastability in Bit Variables Explored Via Simpson's 4-Slot Mechanism. Technical Report 789, School of Computing Science, University of Newcastle upon Tyne (2003).

[55] W. Reisig: *Petri Nets: an Introduction.* EATCS Monographs on Theoretical Computer Science (1985).

[56] A. Rensink: Action Contraction. Proc. of *CONCUR 2000 — Concurrency Theory*, C. Palamidessi(Ed.). LNCS 1877 (2000) 290–304.

[57] A. Rensink and R. Gorrieri: Action refinement as an implementation relation. Proc. of *TAPSOFT '97: Theory and Practice of Software Development*, M. Bidoit and M. Dauchet(Eds.). LNCS 1214 (1997) 772–786.

[58] A. Rensink and R. Gorrieri: Vertical Bisimulation. Technical Report Hildesheimer Informatik-Bericht 9/98, University of Hildesheim (1998).

[59] A. Rensink and R. Gorrieri: Vertical Implementation. *Information and Computation* 170 (2001) 95–133.

[60] A. Rensink and H. Wehrheim: Dependency-based Action Refinement. Proc. of *Mathematical Foundations of Computer Science 1997*, I. Privara and P. Ruzicka (Eds.). LNCS 1295 (1997) 468–477.

[61] A. W. Roscoe: Model-Checking CSP. In: *A Classical Mind, Essays in Honour of C.A.R. Hoare.* Prentice-Hall (1994) 353–378.

[62] A. W. Roscoe, P. H. B. Gardiner, M. H. Goldsmith, J. R. Hulance, D. M. Jackson and J. B. Scattergood: Hierarchical Compression for Model-checking CSP, *or* How to Check 10^{20} Dining Philosophers for Deadlock. Proc. of *Workshop on Tools and Algorithms for The Construction and Analysis of Systems, TACAS*, U. H. Engberg, K. G. Larsen and A. Skou (Eds.). BRICS Notes Series NS-95-2 (1995) 187–200.

[63] A. W. Roscoe: *The Theory and Practice of Concurrency.* Prentice-Hall (1998).

[64] *FDR2 User Manual* : Available at:
http : //www.formal.demon.co.uk/fdr2manual

[65] J. Rushby: Model Checking Simpson's Four-Slot Fully Asynchronous Communication Mechanism. Technical Report, Computer Science Laboratory, SRI International (2002).

[66] D. Sangiorgi: Typed π-calculus at work: a correctness proof of Jones's parallelisation transformation on concurrent objects. *Theory and Practice of Object-Oriented Systems* 5(1) (1999) 25–33.

[67] D. Sangiorgi and D. Walker: *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press (2001).

[68] H. R. Simpson: Four-slot Fully Asynchronous Communication Mechanism. *IEE Proceedings* 137, Pt E(1) (January 1990) 17–30.

[69] H. R. Simpson: Correctness Analysis for Class of Asynchronous Communication Mechanisms. *IEE Proceedings* 139, Pt E(1) (January 1992) 35–49.

[70] J. M. Spivey: *The Z Notation: A Reference Manual.* Prentice-Hall (1992).

[71] H. Wehrheim: Parametric Action Refinement. Proc. of *Programming Concepts, Methods and Calculi - PROCOMET '94*, E.-R. Olderog(Ed.). IFIP Transactions A-56, North Holland (1994) 247-266.

Index