Theories of translation correctness for concurrent programming languages

Mark Millington

Doctor of Philosophy
University of Edinburgh
1985
This thesis addresses the problems of defining and proving translation correctness for programming languages describing concurrent processes. Taking an operationally motivated approach two distinct theories are proposed, a bisimulation theory and a testing theory, and both are articulated on substantial examples.

The bisimulation approach is applied to an example translation from a variant of CSP to CCS by defining a syntax-directed context-sensitive translation from CSP to CCS and establishing the correctness criteria of the bisimulation approach.

The testing approach provides two possible correctness criteria; implementation and complete-implementation, of which the latter implies the former. A substantial example of each is given. In demonstrating complete-implementation the example translates handshake communication in a CSP-like language to shared variable communication in an artificial language manipulating a communication state. For implementation we consider the design and specification of a concurrent sorting machine in CCS for which the design does not completely implement the specification.
Acknowledgements

My first thanks must go to Matthew Hennessy without whose patient advice and encouragement this thesis would certainly have been very much poorer and probably never finished at all. I would also like to express my gratitude to Rocco de Nicola and Wei Li for many talks which have had a fundamental influence on this research.

Most of all I would like to acknowledge the tremendous support given so freely by my family at all the times I needed it most, not just over the duration of this research but over all the years I can remember.

Lastly my thanks go to John Jones and other members of the Edinburgh AI department for putting up with me while I slowly finished this work.

The work reported here was supported by an SERC studentship.

Declaration

This thesis was composed by myself under the guidance of my supervisor Matthew Hennessy.

Mark Millington
Contents

Chapter 0 Introduction

0.1 The problem of translation correctness
0.2 Previous work in this area
0.3 The approach presented here
0.4 A chapter summary

Chapter 1 Labelled transition systems and operational semantics

1.1 Structural Operational Semantics
   1.1.1 The general approach
   1.1.2 Labelled transition systems

1.2 An example semantics for CCS
   1.2.1 A common core of definitions
   1.2.2 CCS

Chapter 2 A bisimulation theory of translation correctness

2.1 Review
2.2 Bisimulations
2.3 Problems with bisimulations
2.4 Translation correctness
2.5 Remarks

Chapter 3 An example translation in the bisimulation theory

3.1 A variant of CSP
   3.1.1 Syntax
   3.1.2 Static semantics
   3.1.3 Dynamic semantics
3.2 The translation
  3.2.1 The troublesome variables
  3.2.2 Auxilliary CCS definitions
  3.2.3 The translation
  3.2.4 An example command translated

3.3 The statement of correctness

3.4 The proof of correctness
  3.4.1 The bisimulation
  3.4.2 The proof

Chapter 4 A testing theory of translation correctness

4.1 Motivations
  4.1.1 Experiment systems
  4.1.2 Translations

4.2 Translation correctness
  4.2.1 Games observers play
  4.2.2 Comparing results
  4.2.3 Complete implementation
  4.2.4 Translation correctness
  4.2.5 Implementations in CCS

4.3 Summary and comparison
  4.3.1 A comparison with the bisimulation theory
  4.3.2 A comparison with Li's theory

Chapter 5 An example in the testing theory

5.1 The translation
  5.1.1 The Source language
  5.1.2 Translation motivation
5.1.3 The Target language
5.1.4 The translation $[\_]_1$
5.1.5 $[\_]_2$ and the experiment systems SEXP, TEXP

5.2 Proving the correctness of the translation $[\_]_2$
5.2.1 Why the bisimulation approach won't work
5.2.2 Four conditions for complete-implementation
5.2.3 Proving the four conditions for $[\_]_2$
5.2.4 Alternative results
5.2.5 Extending the translation

Chapter 6 Weavesort; an example implementation

6.1 Extending EXP(CCS) to NEXP(CCS)
6.1.1 Adding indeterminacy
6.1.2 Extending the set of tests

6.2 The weavesort machine
6.2.1 An informal description of weavesort
6.2.2 The description and specification in CCS
6.2.3 The statement of correctness

6.3 Proving implementation
6.3.1 Proving the may part
6.3.2 Proving the must part
6.3.2.1 Proving part 3
6.3.2.2 Proving part 2
6.3.2.3 Proving part 1
6.3.2.4 Gathering the parts

Conclusion
References
Notation
0. Introduction

It is commonly agreed that the subject of computer science is a fine example of layers of abstraction, each of which is intended to be self-contained and meaningful independently of the others but most of which may be expressed in terms of other, in some sense lower, levels. Consider for example a hypothetical implementation of the programming language Prolog.

Prolog is implemented in the programming language C.

C is implemented in assembly language.

Assembly language is implemented in machine-code.

Machine-code is implemented in register-transfer systems [Gordon 81].

Register-transfer systems are implemented in VLSI structures [Gordon 81, Mead and Conway 80].

VLSI structures are implemented in physical semiconductor devices [Mead and Conway 80].

The aim of this thesis is to investigate rigorous methods by which such claims might be given accurate statement and certain proof.

0.1. The problem of translation correctness

This has been an oft ignored problem for three reasons at least:

1. It has been largely traditional to say "the translation defines the semantics of the source language", so the
translation is correct by definition and the correctness problem goes away since in this case the meaning of a source program is (by definition) the meaning of its translation. At least one reason for adopting this approach has been the difficulties encountered in giving even simple languages an independent semantics. The advantage of an independent semantics is that it can be abstractly stated without irrelevant reference to particular implementation details and a number of different translations (to possibly different languages) proved correct relative to it.

2. Any non-trivial translation involves a great deal of case analysis and detail which carries over dramatically into any correctness proof; it's basically hard work.

3. It is often unclear in what exact sense a source program and its target translation have an acceptably similar meaning. Consider for example the much vaunted 'sameness' of a transistor and a (perfect) switch. They are different at least in the sense that one is physical and the other notional, and one manipulates continuous voltages over continuous time while the other manipulates discrete values (ones and zeros) over discrete time instants. The sense in which they are the same is really determined by an asymmetry: the transistor is used in such a way that its behaviour is analogous to that of a switch, so the switch is the senior partner in this sameness and the transistor is considered not in all its glory but rather in a diminished fashion. The particular diminution in this instance is the insistence that the transistor be used in
its saturation mode [Mead and Conway 80].

0.2. Some previous work in this area

Suppose there are two syntactic languages (sets of valid strings called programs) $L_1$ and $L_2$, the former of which is called the source and the latter the target. An exact formulation of the translation correctness problem will depend on how 'meaning' is ascribed to a program and what aspects of this meaning are to be preserved by a correct translation.

In the denotational approach to programming language semantics ([Gordon 79], [Stoy 77]) each language $L$ is given an independent semantics by a semantic domain $SD(L)$ of program meanings (denotations) and a semantic mapping $M_L$ which associates with each program of $L$ a meaning in $SD(L)$.

The approach of Morris [Morris 73] and the ADJ group [ADJ 79] is that a (syntactic) translation can be viewed as a mapping $\mathcal{T}$ from source programs to target programs and its correctness can be investigated by considering the relationship between the meaning of the original and the meaning of its translation. The statement of correctness for this approach is that the meaning of the translation is a (semantic) translation $tr$ of the meaning of the original where $tr$ is some function from $SD(L_1)$ to $SD(L_2)$. This can be represented as a commuting diagram

$$
\begin{array}{ccc}
L_1 & \xrightarrow{\mathcal{T}} & L_2 \\
M_{L_1} \downarrow & & \downarrow M_{L_2} \\
SD(L_2) & \xrightarrow{tr} & SD(L_2)
\end{array}
$$
to express
\[ \forall p \in L_1, \text{tr}(M_{L_1}(p)) = M_{L_2}(p) \]

This is nicely parameterised on \( \text{tr} \), the exact relationship between the original and its correct translation depends on an acceptable \( \text{tr} \). However, it is expressed in terms of denotational semantics whereas we are interested mainly in the semantics of concurrent programming languages for which there is (as yet) no adequate denotational approach.

In the operational approach to programming language semantics there are no meaning functions or semantic domains as such, but rather each syntactic program is associated with a semantic object, a configuration, by the choice of certain semantic components such as the initial state upon which the program will be executed.

The set \( C \) of all possible configurations forms one part of a labelled transition system, a quadruple \( \langle C, \rightarrow, A, T \rangle \), which defines the possible actions in \( A \) by which one configuration can progress to another. This is achieved in the definition of the transition relation \( \rightarrow : C \times A \times C \) for which we employ \( c_1 \rightarrow a c_2 \), read as "configuration \( c_1 \) can perform the action \( a \), thereby becoming the resulting configuration \( c_2 \)", to denote \( (c_1, a, c_2) \rightarrow \). The set \( T \subseteq C \) of terminal configurations is a set of configurations which are regarded as having terminated and can progress no further.

Thus an operational semantics describes how a program acts rather than associating it with some 'meaning' object and so necessitates a different approach to translation correctness where 'behaviours'
rather than 'meanings' are compared.

A common operational approach to translation is that any syntactic program translation induces a semantic translation between the transition systems associated with each language by its (independent) operational semantics. For example it may be that a configuration comprising a program with a particular state is semantically translated to a configuration comprising the syntactically translated program together with some function of the particular state. Further, there may be some translation from actions in the source system to actions in the target system.

The work of Astesiano and Zucca [Aistesiano and Zucca 82] is an attempt to generalise previous research on comparing behaviours in an operational setting. A semantic translation \( tr \) is taken to be a function from configurations in the source transition system \( <C_1, \rightarrow_1, A, T_1> \) to configurations in the target transition system \( <C_2, \rightarrow_2, A, T_2> \). Note the actions are taken to be from the same set \( A \) and no use is made of the terminal sets.

The method of comparing behaviours is derived from the observational equivalence of [Milner 80] which may be loosely paraphrased as

Two configurations are (behaviourally, observationally) equivalent if whenever either can perform a particular action so can the other in such a way that the resulting configurations are also equivalent.

For observational equivalence this loose paraphrase is formally stated by considering the relator \( F \) defined, for \( R \subseteq C_1 \times C_2 \) by
<c_1, c_2> ∈ F(R) if
1. c_1 \text{-} \mathcal{R}_1 c_1' \text{ implies } \exists c_2'. c_2 \text{-} \mathcal{R}_2 c_2', c_1' \text{-} \mathcal{R} c_2'
2. c_2 \text{-} \mathcal{R}_2 c_2' \text{ implies } \exists c_1'. c_1 \text{-} \mathcal{R}_1 c_1', c_1' \text{-} \mathcal{R} c_2'

Then the observational equivalence of Milner, written here as ~_M, is defined by considering "equivalence up to n observations":

\[ n = F^R(C_1 \times C_2), n ≥ 0 \text{ (where } F^0(C_1 \times C_2) = C_1 \times C_2) \]

then defining

\[ c_1 ~_M c_2 \text{ iff } \forall n \in \text{Nat. } c_1 ~_n c_2 \]

A refinement introduced by Astesiano and Zucca was to recognise that some properties of configurations have no behavioural expression in the transition relation and thus cannot be discerned by the observational equivalence as stated. One particular property of concern is the ability of some agents to indulge in an infinite amount of internal work to the exclusion of all else. If a configuration \( c ∈ C_i (i ∈ \{1, 2\}) \) has this ability then we say it is divergent and write \( \text{div}_i(c) \). Then, to distinguish divergent agents from non-divergent (convergent) agents we can replace \( C_1 \times C_2 \) in the above definition by a subset which will not allow equivalence of convergent and divergent configurations:

\[ \text{CONV} = \{ (c_1, c_2) : \text{div}_1(c_1) \text{ iff div}_2(c_2) \} ⊆ C_1 \times C_2 \]

To achieve this (re)define

\[ F^0(C_1 \times C_2) = \text{CONV} \]

and then

\[ c_1 ~_{AZ} c_2 \text{ iff } \forall n ≥ 0. c_1 ~_n c_2 \]

so that

\[ ~_{AZ} \subseteq \text{CONV} \]

since
The correctness statement for Astesiano and Zucca can then be formally stated as:

\[ \forall n \geq 0. \quad _{n+1}^C \subseteq _n^C \]

The main thrust of their work was to use translations as a means of providing a semantics for the source language, thus they had none of the problems of reconciling a semantics for the translated source object with an (independent) semantics for that source object. If we now consider our transistor example we find a number of weaknesses to this approach.

1. The actions of a switch are expressed in terms of zeros and ones while the actions of a transistor are commonly expressed in terms of continuously varying voltage waveforms. It might be very difficult to find a bijection between the two at all, let alone one for which the statement of correctness holds.

2. The switch is given no seniority in this approach, if it can be proved that a transistor implements a switch then it can also be proved that a switch implements a transistor.

3. It is not at all obvious that observational equivalence is the appropriate vehicle for a translation theory; under certain circumstances it can be too restrictive [de Nicola and Hennessy 84] or too liberal (in which case certain other requirements such as equality of final states in the configurations may need to be given an artificially behavioural aspect [Astenian and Zucca 82]).
The operational approach is also adopted by Li [Li 83]; for Li a semantic translation is a function \( tr \) between configurations in labelled transition systems together with an injective function \( f \) from actions of the source to actions of the target. Two definitions of translation correctness are given, correctness and adequacy of which the latter implies the former and is employed mainly as a proof technique. The aims of Li's work are much closer to those of this thesis than Astesiano and Zuccas', being concerned with translation correctness for languages with independent semantics.

The essential approach of correctness is that the execution of a program can only be represented by a finite sequence of transitions ending in a terminated configuration (a successful computation), by a finite transition sequence ending in a non-terminated configuration (a deadlocked computation which is not terminated but cannot proceed) or by an infinite transition sequence. Saying that a translation is correct amounts to saying that all these three kinds of computations for a program in the source system and in the target system correspond to each other. In particular, the possible final configurations of the translation of a program should be just the translations of the final configurations of the possible computations of the program.

Where the concept of correctness focusses on the result (especially the input-output relation) of a program and its translation, the concept of adequacy pays more attention to the detailed simulation of the behaviour of a program by its translation. Adequacy is stated in six clauses PO to P5 [Li 83
page 212]; loosely PO says that terminated configurations must be translated only to terminated configurations and conversely. P1 ensures that any transition of the source system can be simulated in the target system. P2 and P4 imply a weak version of the converse of P1, and P1 and P3 cover the simulation of infinite transition sequences. Finally, clause P5 implies that deadlocked configurations in the transition systems correspond with each other.

A number of inadequacies come to light if we consider applying these approaches to our transistor example, firstly correctness;

1. Transistors and switches don't have very interesting input-output relations, it's how they behave that is important.

2. The seniority of the switch has been overlooked, for example transistors can burn out if used improperly; is there any corresponding switch state?

Similar difficulties arise in the application of adequacy,

1. The requirements of P2 and P4 disregard the superior role of the switch over the transistor; the switch is required to act like a transistor.

2. It is not obvious that an injective function f can be found between actions of the source and actions of the target, we might expect many waveforms to correspond to (say) a one.

Another approach to the general translation problem as stated is to merge L₁ and L₂ into a language L sufficiently descriptive to span all the levels we are interested in. Languages in which this
approach have been taken contain the family of languages centred on CCS \cite{Milner80}, including Circal \cite{Milne82}, SCCS \cite{Milner82} and Real Time Agents \cite{Cardelli82}, and the work by Gordon \cite{Gordon81} on register transfer systems. In such a case the problem is much reduced since there is typically a well-defined general (to \text{L}) sense in which two programs in \text{L} are "the same", for example they may have the same denotation or have equivalent behaviours.

0.3. The approaches presented here

This thesis contains two new theories of translation correctness within the framework of operational semantics. The first is a bisimulation theory of translation correctness in which the approach is very similar to that of Astesiano and Zueca but for two points:

1. The equivalence from which it is derived is the bisimulation equivalence of Park \cite{Park81b} rather than the observational equivalence of Milner. These two equivalences are very similar, indeed the earlier (loose) paraphrase of observational equivalence stands equally well for bisimulation equivalence. For bisimulation equivalence the paraphrase is again formally stated by considering the relator \( F \) defined earlier; the difference now is that the bisimulation equivalence, written here as \( \sim_B \), is taken to be the maximal fixed-point of \( F \), that is

\[
\sim_B = F(\sim_B),
\]

\[
\forall R \subseteq C_1 \times C_2. (R \subseteq F(R) \text{ implies } R \subseteq \sim_B)
\]

In fact
so a simple proof technique for \( |-B \) is thus

\[ c_1 |-B c_2 \text{ iff } \exists R. c_1 \stackrel{R}{\rightarrow} c_2, R \subseteq F(R). \]

It can be seen since

\[ \forall n \in \mathbb{N}. |-B \models c_n \]

that

\[ |-B \models M \]

and under a natural assumption that

\[ \forall i \in \{1, 2\}, \forall c \in C_i. \{c': c \xrightarrow{a} c'\} \text{ is finite} \]

it can be shown [Hennessy and Milner 85] that

\[ |-B \models M \]

2. Following Astesiano and Zucca we would like to replace \( C_1 \times C_2 \) in the above definition by some subset of \( C_1 \times C_2 \). However, our approach is that the requirement of convergence is removed and replaced by a general parameter \( P \subseteq C_1 \times C_2 \) which is a relation between source configurations and target configurations relative to which correctness is to be established. The choice of \( P \) is part of the statement of correctness for any particular translation, allowing non-behavioural properties of configurations to be simply compared. The advantage of this approach over Astesiano and Zucca is that it is not necessary to distort the translation in an attempt to give certain properties (such as the state component of a terminated configuration) a behavioural expression (by communicating the state contents in a sequence of actions), the required constraints on acceptable correspondence can be expressed in \( P \).
The statement of correctness for a semantic translation, a function \( tr \) from source configurations to target configurations, is given relative to a choice of \( P \subseteq C_1 \times C_2 \) by

\[
tr \text{ is } P\text{-good} \iff \exists R \subseteq P. R \subseteq F(R), \forall c \in C. c \in R tr(c)
\]

In the second theory we adapt the testing equivalence of [de Nicola and Hennessy 84] to a testing theory of translation correctness. The basic idea is to translate not only objects but also how they are to be used (or, equivalently, how they can be tested). Thus for our switch and transistor example the object switch would be translated to the object transistor, and the uses of a switch described in terms of switch positions, zero's and one's will be translated to (say) gate voltages and continuous waveforms for use with the transistor.

In general then the source system will consist of source objects each paired with a prospective use, and their translations will consist of translated source objects paired with translated prospective uses.

A translation is now correct if it is a complete implementation, that is

1. A (source) object must not be amenable to a (source) use

\[
\text{iff}
\]

the translated object must not be amenable to the translated use

and

2. A (source) object must be amenable to a (source) use

\[
\text{iff}
\]

the translated object must be amenable to the translated use.
This approach respects the seniority of the switch since it only demands the transistor behave 'like' a switch when it is being used 'like' a switch.

All this is formalised by introducing the notion of an experiment system \(<E, \rightarrow, S>\) akin to a labelled transition system wherein the configurations \(e \in E\) are called experiments (being applications of tests to objects) and there are no action labels on the transition relation \(\rightarrow\) between experiments since experiments are assumed to progress by the communion of test and object with no external interference or communication. Further, some experiments \(e \in S \subseteq E\) are considered to be successful in the sense that the test has been satisfied.

The next step is to extend the notion of success to computations of experiments. A computation \(c \in \text{comp}(e)\) of an experiment \(e\) is a maximal (i.e. if it is finite then it cannot be extended) sequence of transition steps:

\[ e \rightarrow e_1 \rightarrow e_2 \rightarrow ... \]

Successful computations contain some \(e_1 \in S\), i.e. at some finite point they reach a successful experiment configuration. We can define the result of computation \(c\) by:

\[
\text{result}(c) = \begin{cases} 
\top & \text{if } c \text{ is successful} \\
\bot & \text{otherwise} 
\end{cases}
\]

and then

\[
\text{resset}(e) = \sum_{c \in \text{comp}(e)} \text{result}(c)
\]

Now

\(e \text{ must (succeed)}\) iff \(\text{resset}(e) = \{\top\}\),

\(e \text{ mustnot (succeed)}\) iff \(\text{resset}(e) = \{\bot\}\).
An implementation preserves just these guarantees, so \( \text{tr} \) is an implementation if \( \forall e \in E_1 \):

\[
\begin{align*}
\text{e must} & \implies \text{tr(e) must}, \\
\text{e mustnot} & \implies \text{tr(e) mustnot}
\end{align*}
\]

In some cases we might like a stronger statement of complete implementation in which the above implications are replaced by biconditionals.

0.4. A chapter summary

In chapter 1 the basic concepts and notations concerning labelled transition systems are introduced, and an example semantics is given for the concurrent programming language CCS [Milner 80]. Through this example it is shown how a structural operational semantics can be given using labelled transition systems.

Chapter 2 introduces a bisimulation theory of translation correctness in which the standard notion of bisimulation is extended and refined to a theory of translation correctness.

In chapter 3 the bisimulation theory is articulated on an example syntax-directed translation from CSP [Li 83] to CCS; a complete syntactic and semantic description is given of CSP and the translation is proved correct in the sense of the bisimulation theory. The translation employed is essentially that of Li [Li 83 chapter 6] except that certain restrictions have been lifted through the introduction of a context-sensitive aspect to the translation.
Chapter 4 motivates and presents a testing theory of translation correctness from which two (compatible) correctness criteria emerge; implementation and complete-implementation, of which the latter implies the former.

In chapter 5 complete-implementation is established for an example translation from a simple CSP-like language employing handshake style communication to a language in which communication takes place by altering shared variables. Both languages are given semantics, the translation is proved to be a complete-implementation, and it is shown why this example would present severe difficulties to the other approaches mentioned here.

Chapter 6 demonstrates how the notion of implementation introduced in chapter 4 can be employed within CCS to facilitate the task of specification. The example chosen is a machine to sort a multiset of numbers in a highly concurrent fashion; it is shown how certain aspects of the machine's behaviour may be abstracted in its specification if the notion of implementation is employed rather than the more usual equivalence approach.
1. Labelled transition systems and operational semantics

This chapter is intended to introduce the concepts and techniques employed in the structural operational approach to programming language semantics [Plotkin 81], for use throughout the remainder of this thesis. The first section motivates the general approach and then goes on to introduce labelled transition systems as a formal representation for the approach. The second section presents, for later use, an example semantics for CCS.

1.1. Structural Operational Semantics

For sequential languages there have been four major approaches to programming language semantics: denotational semantics ([Gordon 79], [Stoy 77]), algebraic semantics [Guessarian 81], axiomatic semantics ([Hoare 69], [Dijkstra 76]), and operational semantics ([Landin 63], [Plotkin 81]). The most successful attempts, thus far, at providing a semantics for languages with concurrent and non-deterministic features have followed the operational approach, and in particular the structural operational approach. To introduce this method we can do little better than reproduce, with minor changes, the introduction given by Li in [Li 83 Introduction].

1.1.1. The general approach

Roughly speaking, the operational approach is to formally describe the execution of programs, i.e. to formalise the "operational nature" of programs. In general this purpose is achieved by specifying some convenient abstract machine and modelling the
execution of programs on that machine. This can indicate a way in which the language may be implemented.

Fortunately an operational semantics differs from many other approaches in that it does not require a lot of heavy mathematical machinery and is easy to understand.

A weakness of operational semantics is that because the semantics is based on an abstract machine it usually specifies some irrelevant details. This tends to make the semantics of any non-trivial language very obscure and detailed from the mathematical point of view. To overcome this weakness, or at least reduce it to a minimum, in this thesis we follow the structural operational approach or axiomatic operational approach developed by Plotkin and his colleagues. The basic ideas of this approach are:

1. To abstract away from the irrelevant details of the abstract machines we adopt some of the successful features of the denotational approach such as the use of abstract syntax to replace concrete syntax, and the viewing of states (stores) and environments as functions. Thus a simple configuration of an abstract machine can be written as

\[ \langle S, s \rangle \]

where \( S \) denotes either the current statement to be executed or some token signifying that the program has terminated, for example

\[ \langle \text{done}, s \rangle, \quad \langle \text{abort}, s \rangle \]

may denote (respectively) a normal termination resulting in the state \( s \) and an abnormal termination resulting in the state \( s \). We use \( C \) to denote the set of all configurations used within this
introduction.

To distinguish the successful executions from other computations (deadlocked and infinite computations) we introduce a set $T$ of terminal configurations which is a subset of $C$. For example we can take

$$T = \{<\text{done}, s>, <\text{abort}, s> : s \text{ is a state}\}$$

2. We use labelled transition relations to model computation; thus a transition

$$p \xrightarrow{a} q$$

models one elementary execution step. This transition may be read "the configuration $p$ may perform the action $a$, thereby becoming the configuration $q$". Here the action $a$ denotes an internal action or interactive communication with some super system or the outside world. Let $A$ be the set of possible transition actions in this introduction, then the labelled transition relation

$$\rightarrow : C \times C \times A \times C$$

describes the possible executions of programs. Execution of a program can be viewed as a transition sequence:

$$p_1 \xrightarrow{a_1} p_2 \xrightarrow{a_2} p_3 \xrightarrow{a_3} \ldots$$

which is either infinite or finite. The crux of the matter lies in how to define the labelled transition relation which describes the semantics of a language. Let us consider how we could deal with two typical sequential programming language constructs in this approach:

The semantics of an assignment statement may be defined by the following axiom:

$$<x=e, s> \xrightarrow{\xi} <\text{done}, s[v/x]>$$
where \( v \) is the value taken by the expression \( e \) in the state \( s \), and 
\( s[v/x] \) is the state which is identical with \( s \) except that it 
associates the value \( v \) with variable \( x \). This transition can be 
interpreted as saying that the execution of the statement '\( x := e \)' 
in the state \( s \) results in a new configuration where the program 
has successfully terminated and the new state is the same as 
before except at \( x \) where it takes the value of \( e \). The transition 
action \( c \) should be interpreted to mean that the execution is 
performed without interaction with the outside world.

The semantics of a compound statement \( S_1;S_2 \) may be defined by the 
following rules:

1. If \( <S_1,s> \rightarrow^a <S'_1,s'> \) 
   then \( <S_1;S_2,s> \rightarrow^a <S'_1;S_2,s'> \)

2. If \( <S_1,s> \rightarrow^a <\text{done},s'> \) 
   then \( <S_1;S_2,s> \rightarrow^a <S_2,s'> \)

3. If \( <S_1,s> \rightarrow^a <\text{abort},s'> \) 
   then \( <S_1;S_2,s> \rightarrow^a <\text{abort},s'> \)

Notation

In the remainder of this thesis such rules will be written in the 
form

\[
\begin{array}{c}
\text{Antecedent} \\
\hline
\text{Consequent}
\end{array}
\]

so that, for instance, rule 1 above is written

\[
< S_1,s > \rightarrow^a < S'_1,s' > \\
\hline
< S_1;S_2,s > \rightarrow^a < S'_1;S_2,s' >
\]
As usual, these rules signify that if the hypothesis of the rules define transitions then the conclusions define transitions. How can these rules be interpreted? They tell us that the execution completely depends on the execution of the first statement. The configuration \(<S_1,s>\) can be transformed via transition action \(a\) in three ways; the result is either a normal configuration \(<S_1',s'>\) or a normal terminal configuration \(<\text{done},s'>\) or an abnormal terminal configuration \(<\text{abort},s'>\). Rule 1 says that in the first case \(<S_1;S_2,s>\) is transformed to \(<S_1';S_2',s'>\) via the same action \(a\). Rule 2 says that in the second case \(<S_1;S_2,s>\) is transformed to \(<S_2,s'>\). Rule 3 deals with the third case and says that if the first statement aborts then so does the composition. To summarise we see that the above three rules formalise the description given in the Pascal report "The compound statement specifies that its component statements be executed in the same sequence as they are written" ([Jensen and Wirth 78]).

These two examples reflect the typical character of the structural approach. Two points are worth noting:

1. As with formal deductive systems of the kind employed in mathematical logic this approach defines transitions using axioms and rules. Axioms (which have no hypotheses) define the transitions directly, and rules define the transitions indirectly; that is if all hypotheses define transitions then the conclusion of the rule defines a transition. A definition of this type is called a generalised inductive definition. This feature makes the approach rigorously mathematical, allows a semantics to be set
up with few preconceptions, and also determines the inductive features of the proof techniques.

2. The definition of the transition relation is based on syntactic transformations of programs and simple operations on the discrete data (state and environment). So methods of proof rely heavily on structural induction which might easily be automated; and since programmers and language designers are already familiar with "symbol pushing" this form of the semantics should be more acceptable to them.

These two characteristics will become more pronounced as we progress, so for now we just summarise the above: a labelled transition system can be defined by a quadruple $<C,\rightarrow,A,T>$ and the operational semantics of a language can be given by a labelled transition system.

1.1.2. Labelled transition systems

The use of labelled transition systems to define operational semantics for concurrent programming languages is due to Plotkin [Plotkin 81] and assumes the following computational paradigm

Processes (configurations) perform actions thereby becoming, by some transition relation, other processes. Certain processes which cannot act are regarded as having terminated.

This paradigm is embodied in the mathematical definition of a labelled transition system, a generalisation of the notion of a binary relation where a transition label (or transition action) is
associated with each pair in the relation. In fact a labelled transition system can be viewed as a labelled directed graph with a distinguished set of terminal nodes, though there have been other definitions of labelled transition systems which omit the terminal nodes.

Definition 1.1
A labelled transition system $L$ is a quadruple $L = \langle C, \rightarrow, A, T \rangle$ where

\begin{align*}
C & \quad \text{is a set of process configurations}. \\
\rightarrow & \quad C \times A \times C \quad \text{is a labelled transition relation}.
\end{align*}

$A$ is a set of action labels.

$T \subseteq C$ is a set of terminal configurations with the property: $t \in T$ implies $\exists t', a. \ (t, a, t') \in \rightarrow$.

Given the strong similarity between a labelled transition system and a labelled directed graph with a distinguished set of (terminal) nodes we may describe very simple labelled transition systems by a diagram of their corresponding graphs.

A few more notational points before we proceed:
1. We may write
   \begin{align*}
   \text{config}(L) & \quad \text{for } C, \text{ ranged over by } c. \\
   \text{trans}(L) & \quad \text{for } \rightarrow. \\
   \text{labels}(L) & \quad \text{for } A, \text{ ranged over by } a. \\
   \text{term}(L) & \quad \text{for } T.
   \end{align*}

2. $(c, a, c') \in \rightarrow$ may be written $c \xrightarrow{a} c'$, and read as
"configuration c may perform the action a, thereby becoming the configuration c' ".

It will be useful to have some definitions concerning transition relations for future use,

**Definition 1.2**

Let A be some arbitrary set of actions; the sets A\(^+\) and A\(^*\) of action sequences are defined by

\[ A^+ = \{ (a_1, a_2, ..., a_n) : n \geq 0, a_i \in A, i = 1, ..., n \} \]
\[ A^* = A^+ + \{ \phi \} \]

and ranged over by u, v and w. We employ \( \phi \) to denote the empty sequence of actions rather than the more usual \( \varepsilon \) since that symbol is employed later on with a different meaning.

**Definition 1.3**

Let \( L = \langle C, \rightarrow, A, T \rangle \) be a labelled transition system, then

1. If \( c \rightarrow a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow ... \rightarrow a_n \rightarrow c' \)
   and \( w = (a_1, a_2, ..., a_n) \in A^+ \)
   or
   \( w = \phi \) and \( c = c' \)
   then we write \( c \rightarrow^w c' \) which may be read "the configuration c may perform the transition sequence w, thereby becoming the configuration c' ".

2. If \( c \rightarrow^w c' \) for some \( w \in A^+ \) then \( c' \) is a \( (w-) \)derivative of c and we say \( c \rightarrow^w c' \) expresses a derivation .

3. We may write
   \[ c \rightarrow \]
   \[ for \underline{\frac{1}{2}} \ a, c'. \ c \rightarrow a, c' \]
   \[ c \rightarrow \]
   \[ for \underline{\frac{1}{2}} \ a, c'. \ c \rightarrow a, c' \]
For each $c \in C$ define the set $\text{comp}(c)$ of computations of $c$ by

1. $\langle c_k \rangle_{1, \ldots, n} \in \text{comp}(c)$ if $\frac{1}{j} a_1, \ldots, a_n \in A$ such that
   
   $c = c_1$ and $c_1 \overset{a_1}{\rightarrow} c_2 \overset{a_2}{\rightarrow} \ldots \overset{a_{(n-1)}}{\rightarrow} c_n \overset{a_n}{\rightarrow}$

2. $\langle c_k \rangle_{\text{Nat}} \in \text{comp}(c)$ if $\frac{1}{j} \langle a_j \rangle_{\text{Nat}}$ such that
   
   $c = c_0$ and $\forall k \in \text{Nat}. c_k \overset{a_k}{\rightarrow} a_{k+1}$

Thus $\text{comp}(c)$ contains all (and only) records of maximal derivations, i.e. derivations which are either finite and cannot be extended or are infinite. We may write $c_k$ for $\langle c_k \rangle_k$.

1.2. An example semantics for CCS

In this section we present an example semantics for the concurrent programming language CCS, and in so doing we introduce a number of definitions for use throughout this thesis. We begin by giving definitions concerning the representation and evaluation of expressions. These definitions will be considered common to all languages defined thereafter, starting with the definition of CCS which follows.
1.2.1. A common core of definitions

The languages we will define in this thesis share a common core of definitions dealing with the syntactic and semantic aspects of expressions denoting values. We present them here together with the various properties we would like them to have.

1. Syntax

We assume the following (disjoint unless otherwise stated) sets:

- **Vars** a given countably infinite set \( \{x_i : i \in \text{Nat} \} \) of variables, ranged over by (the metavariables) \( x, y, z \).
- **Exp** a given countably infinite set of expressions ranged over by \( e \) and assumed to contain numerals for the natural numbers.
- **Val C Exp**, a non-empty set of values ranged over by \( v \).
- **Bexp**, a given countably infinite set of boolean expressions, ranged over by \( b \) and assumed to contain the truth values \( tt \) and \( ff \).

It is not our intent to be concerned with the details of expressions so we merely state our assumptions and refer the reader to [Hindley et al 72] for their justification. We assume

1.1 All expressions \( e \) or \( b \) have finite sets \( FV(e), FV(b) \subseteq Vars \) of free variables.

1.2 The substitution of an expression \( e' \) for a (free) variable \( x \) in an expression \( e \) or \( b \) is defined as usual, giving expressions \( e[e'/x], b[e'/x] \). We assume that the following standard facts hold:
a) \( FV(e'[x/x]) = \begin{cases} (FV(e) \cup \{x\}) + FV(e') & \text{if } x \in FV(e) \\ FV(e) & \text{otherwise} \end{cases} \)

b) \( e'[x/x][e''/x] = e'[e''/x] \)

c) \( e'[x/x][e''/y] = e'[e''[e''/y]/x] \) if \( x \notin FV(e''), x = y \).

2. Semantics

We will assume we are given:

States \( S \) Vars \( \rightarrow \) Val, the set of total functions, ranged over by \( s \) and \( \$ \), delivering for each variable \( x \) an associated value \( v \).

All expressions \( e \) or \( b \) can be evaluated with respect to a state \( s \), the state providing bindings for the free variables of \( e \) or \( b \). To this end we assume a total function as in the following definition:

Definition 1.4

We assume a total function

\[ \text{eval}: \text{Exp} \times \text{States} \rightarrow \text{Val} \]

and a partial function

\[ \text{eval}: \text{Exp} \rightarrow \text{Val} \]

with \( \text{eval}(e) \) defined if \( FV(e) = \emptyset \). Further, we require

\[ \text{eval}(e, s) = \text{eval}(e[\text{sub}(X, s)]) \]

where \( \text{sub}(X, s) \) substitutes \( s(x) \) for \( x \) for all \( x \in X \). We assume similar properties for

\[ \text{beval}: \text{Bexp} \times \text{States} \rightarrow \{tt, ff\}, \text{beval}: \text{Bexp} \rightarrow \{tt, ff\}. \]

1.2.2. CCS

In a series of papers (see for example [Hennessy and Milner 85], [Milner 80], [Milner 82], and [de Nicola and Hennessy 84]), Milner
and his colleagues have studied a model of parallelism in which concurrent systems communicate by sending and receiving values along named 'channels'. Communication is synchronised in that the transmission of a value from sender to receiver takes place only when the sender and receiver are both ready, and this transmission is considered as a single event. This kind of communication, called handshake communication, is also found in a large group of modern languages such as CSP [Hoare 78] and Ada [Ichbiah et al 79].

In [Milner 80] an applicative Lambda-calculus style notation called CCS (for Calculus of Communicating Systems) is introduced for describing concurrent processes in terms of their dynamic action capability and static interconnection. More precisely there is a family of languages incorporating these ideas in various forms; in this section we present one such flavour close to that used in [Milner 80].

The language CCS, as presented here, consists of a set of operators for constructing new terms from terms which are already defined. We first give an informal description of each of these operators.

1. Inaction
'Nil' is a term representing the totally idle process which can never perform any action.

2. Action
Let AD be a set of predefined action descriptions ranged over by a,α, and assumed to include the distinguished 'silent' or
'internal' action '1'. Then 'a.' for each a ∈ AD is a unary (prefixed) operator, and if p is a term then 'a.p' is a term. This new term a.p represents a process which can perform the action described by a and then proceed as the process described by p.

3. Choice
If p, q are terms, 'p+q' is a term. The process represented by p+q can act either as the process represented by p or the process represented by q. The choice depends at least to a certain extent on the environment in which p+q finds itself.

4. Asynchronous parallel composition
If p, q are terms then 'p|q' is a term. This represents a composite process consisting of two sub-processes, those represented by p and q, loosely connected. The composite process performs an action when either sub-process performs an action, while its partner remains idle, or when both sub-processes simultaneously engage in complementary actions, i.e. one communicates a value to the other.

5. Renaming
Let Ren be a set of action renamings, ranged over by E. From each E we can define a partial function F from AD to AD such that F[1] = 1. If p is a term then 'p[E]' is a term describing the process represented by p modified so that if p performs action a then p[E] performs F(a) if F(a) is defined. Any action a such that F(a) is not defined is inhibited.

6. Conditional execution
Assuming a set Bexp of boolean expressions ranged over by b we
introduce 'if b then p else q' as a term, whenever p and q are terms, to represent the process which acts like the process represented by p if b evaluates as being true, and like the process represented by q if b evaluates as being false.

7. Procedure call

Assuming a set Proc of procedure names ranged over by P, 'P(e₁,...,eₙ)' is a term. The behaviour of the process represented by P(e₁,...,eₙ) depends on a definition set D containing a pair of the form (P(x₁,...,xₙ),p) where p is a term with free variables in {x₁,...,xₙ}. The behaviour of P(e₁,...,eₙ) is then that of the term p with the values of e₁,...,eₙ substituted for x₁,...,xₙ.

The syntax of CCS

The following sets are needed to define the syntax of CCS:

- **Act**, a given countable set of action names ranged over by A, B, and C.
- **Proc**, a given countable set of procedure names ranged over by P, R, and S.

Action names may also be called 'line names' or 'channels' and in later chapters we may refer to CCS with different sets of action names by subscripting e.g. CCS\_\text{Act}.

The three main syntactic categories of CCS can now be specified as follows:

- **AD**, the set of action descriptions, ranged over by a, α and defined by....
The intent is that
1 denotes "silent" or "internal" action.
Ale denotes the action sending the value of
expression e on channel A.
A?x denotes the action receiving a value on
channel A and binding it to x.

As a notational convenience we may write
A! for A!0
A? for A?x0

where x0 is some distinguished variable occurring in terms only in
these action descriptions. These forms are used when we are
mainly interested in synchronisation rather than value passing.
(Note x0 is distinguished so we don't 'accidentally' bind
variables).

Ren, the set of renaming, ranged over by E;
partial functions from Act to Act.

Terms, the set of terms, ranged over by p,q,r and
defined in a BNF-like notation

\[
p ::= \text{Nil} \mid a.p \mid p+q \mid p|q \mid p[E] \mid
\text{if } b \text{ then } p \text{ else } q \mid P(e_1,\ldots,e_n)
\]

We will allow the case \( n = 0 \) in the latter form, which
will then be written \( P() \) or just \( P \).

A number of notational devices will be employed, we give them here
for convenient reference;

1. It will be found from the derivation rules that the following
pairs need not be distinguished:
\[ p + q \text{ and } q + p, \quad (p + q) + r \text{ and } p + (q + r), \]
\[ p \mid q \text{ and } q \mid p, \quad (p \mid q) \mid r \text{ and } p \mid (q \mid r). \]

We therefore allow the usual (finite) summation and product notation over indexed sets, ignoring order and association:

\[
\sum_{k \in K} p_k = p_{k_1} + p_{k_2} + \cdots + p_{k_n}
\]
\[
\prod_{k \in K} p_k = p_{k_1} \cdot p_{k_2} \cdot \cdots \cdot p_{k_n}
\]

where \( K = \{k_1, k_2, \ldots, k_n\}, n \geq 1 \)

2. Let \( X \subseteq \text{Act}, \) then
\[ p \setminus X \text{ denotes } p[E_X] \]
where \( E_X \) is the renaming defined by

\[
E_X(A) = \begin{cases} 
A & \text{if } A \notin X \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

The process represented by \( p \setminus X \) is obtained from that represented by \( p \) by inhibiting actions whose name occurs in \( X \).

As a special case of the above we define
\[ p \setminus A \text{ denotes } p \setminus \{A\} \]

3. Let \( A, A' \in \text{Act}, \) then
\[ p[A'/A] \text{ denotes } p[E_{A,A'}] \]
where \( E_{A,A'} \) is the renaming defined by

\[
E_{A,A'}(A'') = \begin{cases} 
A' & \text{if } A'' = A \\
A'' & \text{otherwise}
\end{cases}
\]

The process represented by \( p[A'/A] \) is obtained from that represented by \( p \) by renaming the action \( A \) as \( A' \). We will allow
\[ p[A_1', A_2', \ldots, A_n'/A_1, A_2, \ldots, A_n] \]
to denote
4. We will employ an abbreviated form of the conditional construct

\[
\text{if } b \text{ then } p \quad \text{to denote} \quad \text{if } b \text{ then } p \text{ else Nil}
\]

A similar abbreviation for the action construct allows

\[
a \quad \text{to denote} \quad a.\text{Nil}
\]

5. To avoid excessive use of parentheses we assume the following operator precedences

\[
\]

Thus for example

\[
P \mid 1. P' \backslash \lambda + P''[E] \text{ means } (P \mid (1.(P' \backslash \lambda))) + (P''[E])
\]

For each term \( p \) we define the following sets

\[
\begin{align*}
\text{FV}(p) & \subseteq \text{Vars}, \text{the free variables of } p. \\
\text{FP}(p) & \subseteq \text{Proc} \times \text{Nat}, \text{the free procedure names of } p \text{ with their arity.} \\
\text{FL}(p) & \subseteq \text{Act}, \text{the action names mentioned in } p \text{ and often referred to as \text{sort}(p).}
\end{align*}
\]

in a tabular fashion (We will assume that for \((P,n) \in \text{Defined}(D)\) we have a set \(\text{FL}(P,n) \subseteq \text{Act} \) which defines its sort).

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{Nil} & 1.p & \text{A!p} & \text{A?q} & p+q \\
\hline
\text{FV} & \emptyset & \text{FV}(p) & \text{FV}(p)+\text{FV}(p) & \text{FV}(p)\backslash \{x\} & \text{FV}(p)+\text{FV}(q) \\
\text{FP} & \emptyset & \text{FP}(p) & \text{FP}(p) & \text{FP}(p) & \text{FP}(p)+\text{FP}(q) \\
\text{FL} & \emptyset & \text{FL}(p) & \text{FL}(p)+\{A\} & \text{FL}(p)+\{A\} & \text{FL}(p)+\text{FL}(q) \\
\hline
\end{array}
\]
where $E(FL(p))$ denotes the image under $E$ of $FL(p)$.

<table>
<thead>
<tr>
<th>if $b$ then $p$ else $q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p \mid q$</td>
</tr>
<tr>
<td>$p[E]$</td>
</tr>
<tr>
<td>$P(e_1, \ldots, e_n)$</td>
</tr>
<tr>
<td>$FV$</td>
</tr>
<tr>
<td>$FV(p) + FV(q) + FV(b)$</td>
</tr>
<tr>
<td>$FP$</td>
</tr>
<tr>
<td>$FP(p) + FP(q)$</td>
</tr>
<tr>
<td>$FL$</td>
</tr>
<tr>
<td>$FL(p) + FL(q)$</td>
</tr>
</tbody>
</table>

Note we also assume the substitution of expressions for variables in expressions is extended in the usual way to allow the substitution of expressions for variables in terms, written $p[e/x]$, and the substitution of closed terms for parameterless procedure calls $P()$, written $p[q/P]$. (Of course in the former case such substitutions may require a change of bound variables to avoid clashes).

Finally, a **definition set** $D$ is a partial function

$$D: \{P(x_1, \ldots, x_n): P \in \text{Proc}, x_1 \ldots x_n \in \text{Vars} \} \rightarrow \text{Terms}$$

such that if $D(P(x_1, \ldots, x_n))$ is defined then,

1. $FV(p) \subseteq \{x_1, \ldots, x_n\}$
2. $FP(p) \subseteq \text{Defined}(D)$, where
   $$\text{Defined}(D) = \{(p,n): x_1 \ldots x_n \in D(P(x_1, \ldots, x_n)) \text{ is defined}\}$$
3. $x_i = x_j$ if $i = j$.
4. $FL(p) \subseteq FL(P,n)$.

We will assume that

$$D(P(x_1, \ldots, x_n)) = p, \ D(P(y_1, \ldots, y_n)) = q$$
implies
\[ p = q, y_1 = x_1, i = 1, \ldots, n \]
so that \((P, n)\) uniquely identifies the definition.

To describe a definition set \(D\) we introduce the following notation for definitions:
\[ P(x_1, \ldots, x_n) \leftarrow p \text{ denotes } ([P(x_1, \ldots, x_n), p] \in D \]

We shall impose a slight constraint on the collection \(D\), forbidding definition sets containing such definitions as
\[ P(x) \leftarrow A!x + P(x+1) \]
or
\[
\begin{cases}
    P(x) \leftarrow P'(x) \\
    P'(x) \leftarrow P(x)
\end{cases}
\]
in which a process may "call itself recursively without first performing an action". Thus the following are permitted
\[ P(x) \leftarrow A!x + 1.P(x+1) \]
or
\[
\begin{cases}
    P(x) \leftarrow P'(x) \\
    P'(x) \leftarrow 1.P(x)
\end{cases}
\]

We won't here go into the details of how such guardedly well-defined definition sets are formally defined, instead we refer the reader to \([\text{Milner 80}]\).

The operational semantics of CCS

To define the operational semantics of CCS we define a labelled transition system which must of course be parameterised on the syntactic parameters \(\text{Act}\) and \(\text{Proc}\). Further, in giving a semantics
for terms of the form \( P(e_1, \ldots, e_n) \) we need to refer to a definition set \( D \). Thus our labelled transition system should properly be written with three subscripts. We eschew this notational horror, dropping all subscripts unless one or more is of interest, whenever no confusion can arise.

Now assume we have sets \( \text{Act}, \text{Proc} \) and \( D \) as described previously, we define the labelled transition system \( L[CCS] \) by

\[
\text{config}(L[CCS]) = \{ p : \text{FV}(p) = \emptyset, \text{FP}(p) \subseteq \text{Defined}(D) \}
\]

\[
\text{labels}(L[CCS]) = \{ e, \text{A?v}, \text{A?v} : A \in \text{Act}, v \in \text{Val} \}
\]

\[
\text{term}(L[CCS]) = \emptyset
\]

Thus configurations are terms with no free variables and no undefined procedure calls. It is common when defining the configurations of a labelled transition system to define a predicate \( \vdash \) signifying syntactic validity over some set of potential programs. Thus in this case

\[
\vdash p \iff \text{FV}(p) = \emptyset, \text{FP}(p) \subseteq \text{Defined}(D), p \in \text{Terms}.
\]

The definition of this predicate is often referred to as a static semantics since it deals with syntactic properties of the program text. In contrast we now turn, in the definition of \( \text{trans}(L[CCS]) \) to the dynamic semantics of (valid) programs.

To aid in the description of handshake communication (rule 4.3 to follow) we formalise a notion of complementary action. Two action labels \( a \) and \( \bar{a} \) are complementary if one describes the sending of a value on some channel and the other describes the receipt of the same value on the same channel. The simultaneous performance of complementary actions by parallel processes
constitutes a handshake communication. So define

\[
\begin{cases}
  \text{Alv} & \text{if } a = \text{A?v} \\
a & = \begin{cases}
  \text{A?v} & \text{if } a = \text{Alv} \\
  \text{undefined} & \text{otherwise}
\end{cases}
\end{cases}
\]

In order to define \(\text{trans}(L[CCS])\) we first define a more primitive transition relation \(\rightarrow\) which deals explicitly with the internal actions, denoted by \(1\), of a process description. Then \(\text{trans}(L[CCS])\) will be defined so as to ignore internal \(1\) actions. The transition relation \(\rightarrow\) (not \(\text{trans}(L[CCS])\)) is defined by

1. **Inaction**
   
   Nil has no transitions.

2. **Action**
   
   2.1 \(1.p \rightarrow p\)
   
   2.2 \(\text{eval}(e)=v\)
   
   \[\text{A}e.p \rightarrow_{\text{A}lv} p\]
   
   2.3 \(\text{A?x}.p \rightarrow_{\text{A?v}} p[v/x]\)

3. **Choice**
   
   3.1 \(p \rightarrow r\)
   
   \[p+q \rightarrow r\]
   
   3.2 \(q \rightarrow r\)
   
   \[p+q \rightarrow r\]

4. **Parallel**
   
   4.1 \(p \rightarrow r\)
   
   \[p|q \rightarrow r|q\]
   
   4.2 \(q \rightarrow r\)
   
   \[p|q \rightarrow p|r\]
   
   4.3 \(p \rightarrow p', q \rightarrow q'\)
   
   \[p|q \rightarrow p'|q'\]
5. Renaming

5.1 \( p \overset{a}{\rightarrow} q \)

\[
p[E] \overset{F(a)}{\rightarrow} q[E]
\]

if \( F(a) \) is defined where

\[
F(a) = \begin{cases} 
1 & \text{if } a = 1 \\
E(A)!v & \text{if } a = A!v, E(A) \text{ defined} \\
E(A)?v & \text{if } a = A?v, E(A) \text{ defined} \\
\text{undefined otherwise}
\end{cases}
\]

6. Conditional

6.1 \( \text{beval}(b) = \text{tt}, p \overset{a}{\rightarrow} r \)

\[
\text{if } b \text{ then } p \text{ else } q \overset{a}{\rightarrow} r
\]

6.2 \( \text{beval}(b) = \text{ff}, q \overset{a}{\rightarrow} r \)

\[
\text{if } b \text{ then } p \text{ else } q \overset{a}{\rightarrow} r
\]

7. Procedure call

7.1 Suppose \( D(P(x_1, \ldots, x_n)) = p \) is defined then

\[
p[v_1, \ldots, v / x_1, \ldots, x_n] \overset{a}{\rightarrow} r
\]

\[
p[e_1, \ldots, e_n] \overset{a}{\rightarrow} r
\]

if \( \text{eval}(e_i) = v_i \) for \( i = 1, \ldots, n \).

Now, having defined \( \rightarrow \), we can define

\( \text{trans}(L[\text{CCS}]) = \rightarrow \)

by ignoring the internal 1 actions of \( \rightarrow \) as follows:

\( p \overset{\varepsilon}{\rightarrow} q \) if \( p \overset{\varepsilon}{\rightarrow} q, w \varepsilon \{1\}^* \)

\( p \overset{a}{\rightarrow} q \) if \( a \varepsilon \text{ labels}(L[\text{CCS}]), p \overset{\varepsilon}{\rightarrow} \overset{a}{\rightarrow} \overset{\varepsilon}{\rightarrow} q \)

In keeping with the intended interpretation of 1 and \( \varepsilon \) as 'silent'
action we may often drop their occurrence as action labels, writing
-> for 1>, and => for =>

An example
To demonstrate most of the features of CCS we consider a very
simple CCS program to compute the n'th fibonacci number, fib(n),
defined for n ≥ 0 by
\[ fib(0) = 1 \]
\[ fib(1) = 1 \]
\[ fib(x) = fib(x-1) + fib(x-2) \text{ if } x ≥ 2 \]
We would like to define a CCS agent 'fibber(x)' that behaves as if
defined by:
\[ fibber(x) \leftarrow \text{result}!fib(x) \]
Consider the agent defined by:
\[ fibber(x) \leftarrow \]
\[ \text{if } x < 2 \]
\[ \text{then result}!1 \]
\[ \text{else } ((\text{fibber}(x-1) | \text{fibber}(x-2)) [\text{subresult}/\text{result}] | \]
\[ \text{subresult}?x.\text{subresult}?y.\text{result}(x+y) \]
\] \text{\textbackslash subresult}

The idea is that if x < 2 then fibber(x) knows the result is 1 and
can signal that fact and stop. Otherwise it starts up two
subprocesses, fibber(x-1) and fibber(x-2), in parallel to
calculate fib(x-1) and fib(x-2). To prevent these sub-agents
signalling their results to anything other than their caller their
outputs are renamed to be subresults and restricted so they can
only be accessed by their caller. The caller receives these two
subresults and binds them in the variables x and y, finally
signalling their sum as the result.
To see how the operational semantics works consider a possible first step of fibber(2):

\[
\text{fibber}(2) \rightarrow ((\text{fibber}(2-1)|\text{Nil})[\text{subresult/result}] |
\text{subresult?y.result!x+y}
\]

because by rule 2.2

\[
\text{result!1.Nil } \rightarrow \text{Nil}
\]

so by rule 6.1

\[
\text{if } 0 < 2 \text{ then result!1.Nil}
\]

else ((fibber(0-1)|fibber(0-2))[subresult/result] |
\[
\text{subresult?x.subresult?y.result!x+y}
\]

\[
\text{result!1.}
\]

so by rule 7.1

\[
\text{fibber}(2-2) \rightarrow \text{Nil}
\]

so by rule 4.2

\[
\text{fibber}(2-1)|\text{fibber}(2-2) \rightarrow (\text{fibber}(2-1)|\text{Nil}
\]

so by rule 5.1

\[
(\text{fibber}(2-1)|\text{fibber}(2-2))[\text{subresult/result}]
\]

\[
(\text{fibber}(2-1)|\text{Nil})[\text{subresult/result}]
\]

so by rule 4.3

\[
((\text{fibber}(2-1)|\text{fibber}(2-2))[\text{subresult/result}] |
\text{subresult?x.subresult?y.result!x+y})
\]

\[
((\text{fibber}(2-1)|\text{Nil})[\text{subresult/result}]
\]
subresult? y. result! 1 + y)

so by rule 5.1

\[ ((\text{fibber}(2-1)|\text{fibber}(2-2))[\text{subresult}/\text{result}] | \]
\[ \text{subresult}? x. \text{subresult}? y. \text{result}! x + y \]
\] subresult

\[ -1 > ((\text{fibber}(2-1)|\Nil)[\text{subresult}/\text{result}] | \]
\[ \text{subresult}? y. \text{result}! 1 + y \]
\] subresult

so by rule 6.2

if 2 < 2
then result! 1
else ((\text{fibber}(2-1)|\text{fibber}(2-2))[\text{subresult}/\text{result}] | \]
\[ \text{subresult}? x. \text{subresult}? y. \text{result}! x + y \]
\] subresult

\[ -1 > ((\text{fibber}(2-1)|\Nil)[\text{subresult}/\text{result}] | \]
\[ \text{subresult}? y. \text{result}! 1 + y \]
\] subresult

whence, finally, by rule 7.1

\[ \text{fibber}(2) -1 > ((\text{fibber}(2-1)|\Nil)[\text{subresult}/\text{result}] | \]
\[ \text{subresult}? y. \text{result}! 1 + y \]
\] subresult

The remaining steps of the computation are

\[ ((\text{fibber}(2-1)|\Nil)[\text{subresult}/\text{result}] | \]
\[ \text{subresult}? y. \text{result}! 1 + y \]
\] subresult

\[ -1 > \]
If now we use the transition relation $\rightarrow$ to ignore $\rightarrow^{-1}$ moves resulting from the internal communications of subresults we get

\[
\text{fibber}(2) \rightarrow_{\text{result}12} p
\]

and $p$ can perform no further actions.
2. A bisimulation theory of translation correctness

The aim of this chapter is to describe a theory of translation correctness for programming languages whose semantics is given in the abstract operational manner i.e. by a labelled transition system. Any proposed theory of translation correctness will be judged on many criteria, so perhaps it is best to start with a brief examination of the more obvious ones.

Firstly, of course, the theory must be general enough to apply to a large number of interesting translations while being sufficiently powerful to say something interesting about the properties of those translations. In general this must be a trade-off, the more powerful the statement of correctness given by the theory the fewer the number of translations likely to satisfy those conditions. Conversely a surfeit of generality can easily lead to a rather anaemic set of correctness criteria.

Secondly, for the purposes of clarity and modularity in the translations to which the theory is applied it would be greatly advantageous to have a theory which is compositional in the sense that the correctness of a translation which incorporates other sub-translations can be proved assuming only that the sub-translations have been proved correct in the sense of the theory. Given that the theory supports such a proof method the writing of translations themselves can be made easier by segmenting the problem into pieces, a large translation can be written in terms of other translations for subcomponents, and the specified correctness of the whole can be proved given any subtranslations which satisfy the correctness criteria.
Thirdly, a simple intuitive statement of correctness is to be desired so that it is easily seen what properties the translation has; criteria couched in terms of obscure mathematical concepts about the theory of computation will not be readily acceptable to most computer scientists.

Lastly, given the seemingly inherent complexity of most translations and their correctness proofs it would be useful if there were some hope of automating at least a proof checker, and preferably a proof generator.

The basic approach of this chapter will be to generalise a notion of behavioural equivalence which has been previously used in proving the equivalence of process descriptions in CCS [Milner 82]. In the previous work the process descriptions were configurations in the same transition system. Here we will depart from this approach in that we will need to extend the notion of equivalence to objects in different transition systems. This will require some refinement of the definitions but the major difference will lie in the addition of a non-behavioural component to the equivalence in order to cope simply with the notion of final states. We believe that the theory to be outlined goes some way to achieving each of the above criteria.

2.1.1. Review

As we saw in the introduction there have been a number of previous attempts to outline a theory of translation for programming languages, the most readily accepted of which is the denotational theory of Morris [Morris 73] which is described in terms of
commuting diagrams between algebras. This is a very elegant and intuitive theory but for our purposes it fails at an important point. Our attempt is aimed mainly for programming languages with non-determinism and parallelism built in as primitives; as yet there is no satisfactory treatment of such features within the algebraic approach.

Instead we turn to work already performed in the area of operational semantics by essentially two groups; in order to focus the discussion we will examine a single piece of work from each with particular reference to the criteria above rather than the less technical criteria discussed in the introduction.

The first, and most comprehensive, piece is the thesis of Wei Li [Li 83]. Chapter 5 of his thesis details an operational translation theory with much the same intent as this work. Roughly speaking the basic ideas of his approach were:

1. A syntactic translation between languages \( \text{Lang}_1 \) and \( \text{Lang}_2 \) is a function \( f: \text{Lang}_1 \rightarrow \text{Lang}_2 \) translating programs in \( \text{Lang}_1 \) to programs in \( \text{Lang}_2 \). Any syntactic translation induces a semantic translation between the transition systems which define, in the structural axiomatic form, the operational semantics of \( \text{Lang}_1 \) and \( \text{Lang}_2 \). Configurations in the respective transition systems will typically be composed of programs and some semantic elements such as states or environments. A semantic translation is a function from configurations to configurations which translates each component separately i.e. programs to programs and semantic components to semantic components. This is the view we will
take, and in the remainder of this thesis we will assume this distinction.

2. The execution of a program together with an input state can be represented by a finite transition sequence ending in a terminal configuration (successful computation), by a finite transition sequence ending in a nonterminal configuration (deadlocked computation), or by an infinite transition sequence (divergent computation). Saying that a translation is correct amounts to saying that all these three kinds of computations for a program in the object system and its translation in the target system correspond to each other. In particular, the possible final configurations of the translation of a program should be just the translations of the final configurations of the possible computations of the program for any given initial input state.

3. Having formalised the idea of the correctness of a translation, sufficient conditions, called adequacy conditions, which guarantee the correctness of a translation are proposed. It is adequacy which is established in the body of Li's proofs, correctness follows as a corollary.

This theory possesses substantial generality and power but falls down in some of its details. For instance,

a. The requirement that termination and deadlock be distinguished within and between transition systems is just too stringent for some translations; in his example translation from CSP to CCS, Li needed to doctor the CCS
transition system definition to introduce an appropriate
distinction between deadlock and termination. By the nature
(cleanliness) of his translation he was able to determine the
appropriate choice of terminals in a way which may not be
possible in general.

b. The compositionality of his theory is very weak as it is
double layered; a correctness criterion together with an
underlying set of conditions sufficient to guarantee
correctness. A composition theorem can be proved for
correctness [Li 83 Theorem 5.2] but when trying to prove a
translation with sub-translations correct the correctness of
the sub-translations is unlikely to enable proof of the
adequacy conditions to support correctness of the complete
translation. A solution would be to prove a composition
theorem for adequacy but as yet this has not been achieved.
It should be noted that adequacy is a good deal stronger than
correctness, dealing as it does with the observable actions
of the processes as well as their resulting states, and thus
is not a set of conditions necessary for correctness.

c. The statement of correctness is quite clear but says little
about behaviour while the statement of adequacy says a good
deal about behaviour but is rather opaque.

d. The proof technique of establishing adequacy leads to a very
large detailed proof holding little hope for machine
checkability. While the former charge may be laid against
almost any translation proof, including those to follow
unfortunately, the level of detail and subtlety is sufficient
to make the proofs very difficult to follow.

The second theory advanced is contained in a paper by Astesiano and Zucca [Asteniano and Zucca 82], wherein again the translation example is CSP to CCS. The approach taken is quite similar to that proposed in this chapter in that an existing equivalence for CCS is extended to a theory of translation. To introduce such non-behavioural factors as final state equality they resort to the introduction of processes that write out the contents of the final state so that the states can be compared in a behavioural way. This introduces processes into the translation which are there just for the theory to work rather than for the translation to work. The theory proposed here takes into account factors other than just behavioural properties of the processes and thus avoids the introduction of unnecessary terms.

The equivalence generalised in that work was the observational equivalence proposed by Milner [Milner 80]; the equivalence generalised in this chapter is the bisimulation equivalence of Park [Park 81b],[Milner 82]. This more recent equivalence has the advantage of an elegant proof technique while delivering a slightly finer equivalence than observational equivalence. This proof technique holds out some hope of machine implementation, programs to check simple bisimulations may easily be written in a number of programming languages. However a great deal more work must be done before bisimulations in the form used in the proof of chapter 3 can be generated by machines.
2.1.2. Bisimulations

In this subsection we review the notion of bisimulation equivalence between processes and the proof technique that the theory affords. The essence of the approach is that we do not wish to distinguish agents which have the same derivation tree, this is made precise by the notion of bisimulation between agents.

Definition 2.1

Let $L = \langle C, \rightarrow, A, T \rangle$, then for any $R \subseteq C \times C$ define $F(R)$ to be the set of pairs $(c_1,c_2)$ satisfying

1. $c_1 \xrightarrow{a} c_1'$ implies $c_2 \xrightarrow{a} c_2'$ & $c_1' R c_2'$
2. $c_2 \xrightarrow{a} c_2'$ implies $c_1 \xrightarrow{a} c_1'$ & $c_1' R c_2'$

whenever $a \in A$. A bisimulation is a relation $R \subseteq C \times C$ such that $R \subseteq F(R)$, thus $R$ is a bisimulation iff 1 and 2 above hold for $R$.

Two configurations $c_1,c_2 \in C$ are (bisimulation) equivalent, written $c_1 \sim c_2$, if there exists a bisimulation $R$ with $c_1 R c_2$.

It should be remembered that $F$ depends on the particular transition relation involved and so should most properly be written $F_{\rightarrow}$. However, we will generally omit the subscripts for convenience.

The motivation behind this definition may be made clearer by a simple paraphrase of the definition (given already in the introduction):

"Two agents are equivalent if whenever one of them can perform an observable action so can the other in such a way
that the resulting agents are equivalent".

Now we have an elegant proof technique; to show \( c_1 = c_2 \) it is necessary and sufficient to find a bisimulation containing the pair \((c_1, c_2)\). To demonstrate the proof technique we will consider an example from CCS.

Proposition 2.1

\( (1.\text{wave})|\text{shout} = 1.(\text{shout}|\text{wave}) \)

Proof: It is sufficient to provide a bisimulation containing the above pair of agents; the following set \( R \) of pairs is just such a relation.

\[
\begin{align*}
(1.\text{wave}|\text{shout} , 1.(\text{shout}|\text{wave})) , & \quad [1] \\
(\text{wave}|\text{shout} , \text{shout}|\text{wave}) , & \quad [2] \\
(\text{Nil}|\text{shout} , \text{shout}|\text{Nil}) , & \quad [3] \\
(1.\text{wave}|\text{Nil} , \text{Nil}|\text{wave}) , & \quad [4] \\
(\text{wave}|\text{Nil} , \text{Nil}|\text{wave}) , & \quad [5] \\
(\text{Nil}|\text{Nil} , \text{Nil}|\text{Nil}) & \quad [6]
\end{align*}
\]

To prove \( R \) is a bisimulation we will apply the following Lemma which helps reduce the number of cases which must be considered.

Lemma 2.1

Suppose \( \text{Rel} \subseteq \text{Terms} \times \text{Terms} \), then \( \text{Rel} \subseteq \text{F}^{\rightarrow}(\text{Rel}) \) is implied by the following closure condition: Whenever \( p \text{ Rel} q \) all the following hold,

1.1 If \( p \xrightarrow{1} p' \) then \( \exists q'. q \xrightarrow{e} q', p' \text{ Rel } q' \)

1.2 If \( p \xrightarrow{a} p' \), \( a = 1 \) then \( \exists q'. q \xrightarrow{a} q', p' \text{ Rel } q' \)
2.1 If $q \rightarrow^1 q'$ then $\exists p'. p \rightarrow^5 p', p' \text{ Rel } q'$
2.2 If $q \rightarrow^a q'$, $a = 1$ then $\exists p'. p \rightarrow^a p', p' \text{ Rel } q'$
Proof by induction on the length of the derivation $p \rightarrow^a p'$ or $q \rightarrow^a q'$.

To apply this lemma we take each pair $(p, q)$ in $\text{Rel}$ and establish conditions 1.1 ... 2.2, first by considering 1.1 and 1.2, and then considering what amounts to the converse, i.e. 2.1 and 2.2. Consider establishing condition 1.1 for pairs in [1], this will be written

$$[1] \rightarrow^1 [2]$$

This is intended to express the statement

If $(p, q) \in [1]$ and $p \rightarrow^a p'$ then

either $a = 1, \exists q'. q \rightarrow^5 q', (p', q') \in [2]$  
or $a = \text{shout}, \exists q'. q \rightarrow^\text{shout} q', (p', q') \in [4]$

Using this notation the proof may proceed by cases on $[n]$ for each $n \in [1, \ldots , 6]$, each case being considered first for condition 1 and then condition 2:

Then 2: $[1] \rightarrow^1 [2]$
Then 2: $[2] \rightarrow^\text{wavel} [3]$
2.1.3. Problems with bisimulations

The definition of bisimulation equivalence is based on a completely observational notion of comparison by visible action; properties of the agents which do not show up as observational differences will not be distinguished. This is not to say that the theory is not extendable by the introduction of more powerful observers who can distinguish hitherto equivalent agents, rather that having selected a form of observation it may also be found useful to examine agents without completely specifying how the examination is to be performed. We will return to this approach in defining P-bisimulations but for now we will consider a few examples of agents which are bisimulationally equivalent but differ in certain interesting properties.

Example 1: The equivalence of non-movers.

In a concurrent setting such as CCS, configurations may have no derivatives (i.e. be unable to proceed) for one of two reasons. Firstly they may be configurations representing a successful conclusion to some computation, an example of this might be Nil in
CCS. The second reason involves the concept of deadlocked configurations wherein, in a simple case, two agents are attempting to proceed with some activity yet are unable to synchronise their efforts. For example if two overly polite people reach a door at the same time each may wait for the other to pass first, with the result that neither can pass. This is called a deadlock and can be modelled in CCS by the configuration

\[ \text{fred_first}! \cdot \text{bill_first} ?? | | \text{bill_first}! \cdot \text{fred_first} ? \]

In this analogy \( \text{Nil} | | \text{Nil} \) might model a state where neither agent wants to pass through the door, both are content where they are. However, we have

\[ \text{fred_first}! \cdot \text{bill_first} ?? | | \text{bill_first}! \cdot \text{fred_first} ? = \text{Nil} | | \text{Nil} \]

which fails to recognise that one term (the left side) is 'frustrated' and the other 'content' in our interpretation. More formally may we say one is deadlocked and the other terminal.

**Example 2: Silent spinning**

The statement of bisimulation equivalence is given in terms of the relation \( \rightarrow \) which is ternary; \( c_1 \rightarrow a c_2 \) is to be interpreted as asserting that the agent \( c_2 \) is a result of agent \( c_1 \) performing action \( a \). This fails to capture the notion of an infinite sequence of 'silent' actions (say \( 1 \) moves in CCS with \( \rightarrow \) as the transition relation) wherein there is no result agent; infinite silent action is simply not accounted for by the definitions. For example if we assume the definition

\[ \text{spin} \triangleq 1 \cdot \text{spin} \]

then, with respect to the \( \rightarrow \) transition relation, \( \text{Nil} \) and \( \text{spin} \) have (up to renaming) identical derivations, ie

\[ \text{Nil} \rightarrow^* \text{Nil} \quad \text{only}, \quad [1] \]
and \( \text{spin} \rightarrow \text{spin only} \). \[2\]

Thus \( \rightarrow \) fails to distinguish between transitions wherein no action is performed, such as \([1]\), and transitions, such as \([2]\), wherein a silent action may be performed. This leads to

\[
\text{spin} = \text{Nil}
\]

and \( \text{spin}|a! = a! \) \[3\]

which is intuitively unpleasant since in \([3]\) the agent on the left need never perform the action \(a!\).

2.1.4. Translation correctness

We can now define a (semantic) translation as a function from configurations to configurations; such a function may well be induced by a syntactic translation in the manner discussed earlier but this is not necessary to the definition. In fact the definition could allow partial functions as translations but for simplicity we adopt the stronger form. In what follows let \( L_1 = \langle C_1, A, \rightarrow_1, T_1 \rangle, L_2 = \langle C_2, A, \rightarrow_2, T_2 \rangle \) be two transition systems over the same set of actions. Where no confusion can occur we will simply drop the subscripts from the transition relations.

**Definition 2.2**

A translation from \( L_1 \) to \( L_2 \) is a total function \( \text{tr}: C_1 \rightarrow C_2 \) and we may refer to \( L_1 \) as the source system and \( L_2 \) as the target system.

To handle translations the theory of bisimulations will be extended to the case where the agents related are configurations in different transition systems. For simplicity we will assume
that the two transition systems have the same set of actions, though it would be quite simple to relax this requirement it would also be notationally disastrous.

Definition 2.3

We first extend $F$ as given previously in Definition 2.1: for any $R \subseteq C_1 \times C_2$ define $F(R)$ to be the set of pairs $(c_1, c_2)$ satisfying

1. $c_1 \xrightarrow{a} c_1'$ implies $c_2 \xrightarrow{a} c_2'$, $c_1' R c_2'$
2. $c_2 \xrightarrow{a} c_2'$ implies $c_1 \xrightarrow{a} c_1'$, $c_1' R c_2'$

whenever $a \in A$.

A bisimulation is a relation $R \subseteq C_1 \times C_2$ such that $R \subseteq F(R)$. Thus the case when $L_1 = L_2$ is just the usual definition.

Turning to the criteria for translation correctness we introduce the notion of good translations as translations which preserve the behaviour of the source configuration as the behaviour of its image, simply by demanding there exists a bisimulation containing the pair $(c, \text{tr}(c))$ for each source configuration $c$.

Definition 2.4

A translation $\text{tr}$ from $L_1$ to $L_2$ is good if there exists a bisimulation $R \subseteq C_1 \times C_2$ such that $\forall c \in C_1. c R \text{tr}(c)$

Having generalised bisimulations to pairs of transition systems we now add an extra discriminatory mechanism, a $P$-bisimulation $R$ is a bisimulation such that $c_1 R c_2$ implies $c_1 P c_2$.

Definition 2.5

Let $P \subseteq C_1 \times C_2$ be an arbitrary binary predicate. A $P$-bisimulation is a bisimulation $R \subseteq P$. 

Of course we can now augment Definition 2.4:

**Definition 2.6**

A translation $\text{tr}$ is **P-good** if there exists a P-bisimulation $R$ such that

$$\forall c \in C, c R \text{tr}(c)$$

To prove a translation $\text{tr}$ is P-good we first demonstrate a bisimulation between configurations of the object system and their translations, then we show that the bisimulation is wholly contained in P. Thus with this definition of translation correctness we can retain the normal bisimulation proof technique in a slightly augmented form.

One of the properties we would most like to have is compositionality,

**Proposition 2.3**

Suppose

$$L_1 = \langle C_1, \rightarrow, A, T_1 \rangle \quad i = 1, 2, 3,$$

$$P_i = C_i \times C_{i+1} \quad i = 1, 2,$$

$$P_3 = C_1 \times C_3,$$

$$\text{tr}_i : L_i \rightarrow L_{i+1} \text{ is } P_i \text{-good } i = 1, 2,$$

$$c_1 \text{ P}_1 c_2, c_2 \text{ P}_2 c_3 \text{ implies } c_1 \text{ P}_3 c_3$$

then

$$\text{tr}_2 \circ \text{tr}_1 \text{ is } P_3 \text{-good}$$

**Proof** we have two bisimulations

$$R_i = P_i, \forall c_1 \in C_i \cdot c_1 R_i \text{tr}_1(c_i), i = 1, 2$$

the take
To justify the introduction of the predicate $P$ we now show its utility in a few simple examples.

**Examples of non-behavioural predicates**

We have already seen in the review of bisimulations that we would like a few 'standard' ways of extending the notion of bisimulation; by varying $P$ in the definition of $P$-goodness we can alter the requirements for translation correctness to capture these extensions. Note it is not being suggested that the following examples for $P$ should always be used, for instance the main example in the next chapter is not terminal-good, where 'terminal' is as defined in example 1 to follow. The intent is to allow a degree of flexibility in the translation correctness requirements while retaining a general framework within which many examples will fit.

**Example 1**

$$c_1 \text{ terminal } c_2 \iff (c_1 \in T_1 \iff c_2 \in T_2)$$

Thus a terminal-bisimulation cannot confuse deadlocked and terminal states; terminals may only be paired with terminals.

**Example 2**

There are many ways in which divergence might be defined to indicate properties of agents such as the ability to 'spin' silently or some degree of syntactic 'under-definedness'. Exactly how it is defined will depend on the transition system and the
potentially undesirable properties of certain configurations in
that system, however suppose we have predicates \( \text{div}_1 \subseteq C_1 \) and \( \text{div}_2 \subseteq C_2 \) which indicate divergence of their arguments. Then we can
define
\[
c_1 \text{ converge } c_2 \iff (\text{div}_1(c_1) \iff \text{div}_2(c_2))
\]
A converge-bisimulation existing between two agents implies that
each can diverge after a sequence of actions only if the other
can.

Example 3
Suppose we have two total valuation functions \( f_1 : C_1 \rightarrow V \) and \( f_2 : C_2 \rightarrow V \) for some set \( V \) of values, then define
\[
c_1 \text{ valuation } c_2 \iff f_1(c_1) = f_2(c_2)
\]
This sort of relation allows the comparison of non-behavioural
properties of configurations; for example a comparison of states
in CSP or imperative languages in general.

2.1.5. Remarks

We will now examine a few simple consequences of the definition of
P-goodness and point out one of the peculiarities of this
approach.

Proposition 2.2
If \( R \) is a P-bisimulation and \( c_1 R c_2 \) it does not follow that every
bisimulation containing the pair \( (c_1, c_2) \) is a P-bisimulation.
Proof: Recalling the graphical notation for transition systems
introduced in Chapter 1, take
Then P is a P-bisimulation with what is probably the intended matching of terminals. However the following is also a bisimulation but not a P-bisimulation:

\[ P = \{<c,c>, <\text{done, done}>, <\text{abort, abort}>\} \]

At first sight this looks rather serious, implying essentially that in some cases the P in P-goodness seems to have no restrictive power over the correspondence between source and target configurations. The intent in introducing P was to establish 'allowable' correspondence between source and target configurations, the idea being that after any sequence of actions performed by both a source configuration and its translation the resulting configurations should be related by P. For instance, if P was a valuation comparison of states then we would expect that the final configurations have the same state component. The example above shows this is not necessarily true and we now examine what, if anything, has been lost.

The following proposition asserts that for every finite computation of source (or target) there is a corresponding finite computation of target (or source). Further, if the source (target) computation has a single outcome then all outcomes of the target
(source) are P-related to that outcome.

**Proposition 2.4**

Suppose \( R \) is a \( P \)-bisimulation, \( c_1 \ R \ c_2 \), then \( \forall w \in A^* \)

1. If \( c_1 \xrightarrow{w} c_1' \) then \( \exists c_2' . c_2 \xrightarrow{w} c_2' , c_1' \ R \ c_2' \)

2. If \( c_2 \xrightarrow{w} c_2' \) then \( \exists c_1' . c_1 \xrightarrow{w} c_1' , c_1' \ R \ c_2' \)

3. If \( c_1 \xrightarrow{w} c_1' \) and \( (c_1 \xrightarrow{w} c) \implies c = c_1' \) then

\[ \forall c_2' . c_2 \xrightarrow{w} c_2' \implies c_1' \ P \ c_2' \]

**Proof** by induction on the length of the derivation of \( \xrightarrow{w} \).

Thus if both source and target run through a sequence \( w \) of actions then the resulting configurations are not necessarily related in \( P \) unless \( w \) "specifies to within \( P \)" those configurations. For instance in the example above a can lead to either done or abort, it does not specify which. However, if we changed the system to

\[
\begin{array}{c}
\text{c} \\
\text{a} \quad \text{b} \\
\text{done} \quad \text{abort}
\end{array}
\]

where \( a \neq b \) then \( P \) is the only bisimulation.

Having now outlined the theory it seems appropriate to consider some of its features and faults. The first point to note is that no restriction has been placed on the form that \( P \) may take. Thus in transition systems with a rather impoverished set of actions little of the work of comparison may be done by the bisimulation part of the definition and much more by the \( P \) part. In the extreme case where no externally observable actions are performed all the discriminatory power lies with \( P \). Since this theory was
not really aimed at such systems this is not a cause of great concern, but it does lead to certain theoretical difficulties as to the exact power of the theory. The theory is at least as strong as a pure bisimulation theory but beyond that it seems to have few limits. This gives it more the flavour of a theory schema, which at least can be defended on the grounds of flexibility, a framework within which many different sorts of translations can be proved correct. In the next chapter we articulate the theory on an example translation in which subtranslations carry in their P predicates properties that are necessary for their correct functioning within their intended environment. These extra properties are highly dependent on the nature of the particular translation which requires them and thus have no place in the statement of a general theory.

Secondly, the efficacy of the theory relies a great deal on an appropriate choice of transition systems. The intended interpretation of the bisimulation relation is that of some observer performing tests on the external observable actions of two systems and comparing the results. If the ternary $\rightarrow$ relation does not match the intuitions of observability then some degree of intuitive acceptability may be lost.

Lastly, as we have noted, this theory is at least as strong as a purely bisimulation theory. It may be that for some commonly occurring translations this is rather too strong already, there may be intuitively acceptable translations which are not P-good for any P. This is a point which we will return to in later chapters but for the moment we will just remark on it in passing.
3. An example translation in the bisimulation theory

The aim of this chapter is to present an improvement of the translation given in [Li 83 chapter 6]. The improvement takes two forms; firstly, the translation is extended to cover a larger range of syntactic forms. Specifically, the restriction of [Li 83 chapter 6 section 6.2.2] on the allowable forms of guarded commands is dropped so that the form of guarding is the more general 'strong' form for all commands. Secondly, an improved proof technique is obtained by using the bisimulation theory of the previous chapter.

The description of CSP employed is essentially that of [Li 83 chapter 2] with some minor changes to simplify the statement of correctness; these changes will be pointed out and explained as they occur. Due to the great bulk of the description of CSP and the large degree of case analysis inherent in the bisimulation approach, only the most difficult and exemplary cases are dealt with in the proof in order that this chapter maintain a reasonable size. Section 3.1 constitutes a formal syntax and semantics for the variant of CSP we will employ, section 3.2 introduces the translation, section 3.3 provides the correctness criteria, and section 3.4 establishes the claimed correctness.

3.1. A variant of CSP

In common with all languages defined in this thesis our variant of CSP shares the definitions, given in chapter 1, of expressions, values, states etc. and in what follows we take these as given. We will closely follow the description given in [Li 83 chapter 2]
since our interest is not, primarily, in providing a new semantics for CSP; we take that as having been achieved and refer the interested reader to [Li 83 chapter 2] whenever additional motivation or explanation is sought. As usual we begin with the syntax;

3.1.1. Syntax

The syntax of CSP is parameterised on the defined categories of section 1.2.1 and the following syntactic categories:

Plab - a given countably infinite set of process labels, ranged over by P, Q, R.

Pten - a given countably infinite set of pattern symbols, ranged over by W.

The categories of guarded commands and commands of CSP are defined using a BNF-like notation.

Gcom - the set of guarded commands, ranged over by gc

\[ gc ::= b \rightarrow c \mid gc_1 \circ gc_2 \]

Com - the set of commands, ranged over by c, \( \xi \)

\[ c ::= \text{SKIP} \mid \text{ABORT} \mid x := e \mid c_1 ; c_2 \mid \text{if } gc \text{ fi } \mid \text{do } gc \text{ od } \mid P ? W(x) \mid P ! W(e) \mid c_1 || c_2 \mid P :: c \mid \text{process } P :: c \]

where \( gc_1 \circ gc_2 \) is called an alternative (guarded command), if \( gc \) fi is called a conditional (command), do gc od is called a repetitive (command) and \( P ? W(x) \), \( P ! W(e) \) are called input and output commands respectively. Finally, \( P :: c \) denotes a process.
named P with body c and process P; c is the declaration of the label P so that a process named P within c cannot be directly referenced since the scope of its name is the command c.

3.1.2. Static Semantics

Before giving a formal operational semantics for CSP, it is necessary to guarantee that all syntactic clauses considered are valid. This notion of syntactic validity is a device intended to capture some of the context-sensitive aspects of programming language grammars, dealing with such things as non-interference of processes. In order to capture a notion of non-interference between processes accessing a shared state we need to introduce and define the following sets.

Definition 3.1

Syn - the union of Gcom and Com, ranged over by ω.
RV(ω) - the set of variables in ω that may be read.
WV(ω) - the set of variables in ω that may be written to.
FPL(ω) - the set of free (agent) process labels contained in ω.

For a guarded command gc we also need the following predicate when giving the dynamic semantics; we give its definition here for convenience.

Bool(gc) - the disjunction of the boolean guards occurring in gc.

For each piece of syntax ω we define RV(ω), WV(ω), and FPL(ω) in a tabular fashion, the tables are to be read by cross-indexing the function required (on the vertical axis) with its argument (on the horizontal axis); thus eg. WV(P?W(x))={x} from the intersection of
second row (WV) and first column (P?W(x)) in the third table.

<table>
<thead>
<tr>
<th>b -&gt; c</th>
<th>gc_1 &amp; gc_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RV</td>
<td>FV(b)+RV(c)</td>
</tr>
<tr>
<td>WV</td>
<td>WV(c)</td>
</tr>
<tr>
<td>FPL</td>
<td>FPL(c)</td>
</tr>
<tr>
<td>Bool</td>
<td>b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SKIP</th>
<th>ABORT</th>
<th>x:=e</th>
<th>c_1;</th>
<th>c_2</th>
<th>if gc</th>
<th>fi</th>
<th>do gc</th>
<th>od</th>
</tr>
</thead>
<tbody>
<tr>
<td>RV</td>
<td>φ</td>
<td>φ</td>
<td>FV(e)</td>
<td>RV(c_1)+RV(c_2)</td>
<td>RV(gc)</td>
<td>RV(gc)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WV</td>
<td>φ</td>
<td>φ</td>
<td>{x}</td>
<td>WV(c_1)+WV(c_2)</td>
<td>WV(gc)</td>
<td>WV(gc)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPL</td>
<td>φ</td>
<td>φ</td>
<td>φ</td>
<td>FPL(c_1)+FPL(c_2)</td>
<td>FPL(gc)</td>
<td>FPL(gc)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| P?W(x) | P!W(e) | c_1 || c_2 | R::c | process R;c |
|--------|--------|-------|-------|------|-------------|
| RV     | φ     | FV(e) | RV(c_1)+RV(c_2) | RV(c) | RV(c)      |
| WV     | {x}   | φ    | WV(c_1)+WV(c_2) | WV(c) | WV(c)      |
| FPL    | φ     | φ    | FPL(c_1)+FPL(c_2) | {R}  | FPL(c)-{R} |

Finally, the set of free variables contained in a command or a guarded command is defined by:

\[
\text{FV}(c) = \text{RV}(c) + \text{WV}(c),
\]
\[
\text{FV}(gc) = \text{RV}(gc) + \text{WV}(gc).
\]

The property of being a valid syntactic form is written as ⊨ and may be written in subscripted form such as \(\neg_{\text{Gcom}} gc\) to assert that gc is valid as a guarded command, and \(\neg_{\text{Com}} c\) to assert that c is a valid command.
Now we can define what we mean by non-interference through shared variables in parallel processes. The idea is that each command in a parallel construct should act as if it had the (shared) state completely to itself; this makes it possible to reason about the behaviour of a parallel construct \( c_1 || c_2 \) in terms of the behaviour of its constituents \( c_1 \) and \( c_2 \) each alone. Consider, if \( c_2 \) can alter the value of a variable we do not want \( c_1 \) to be able to 'see' this change (by reading the value of this variable from the store) since then \( c_1 \) would be interfered with. Thus we require

\[
RV(c_1) \land WV(c_2) = \phi
\]

Similarly we also require

\[
WV(c_1) \land WV(c_2) = \phi
\]

since otherwise \( c_2 \) might interfere with the final result state produced by \( c_1 \). Thus we will require

\[
FV(c_1) \land WV(c_2) = \phi
\]

and symmetrically

\[
FV(c_2) \land WV(c_1) = \phi
\]

Definition 3.2

Let \( c_1, c_2 \in \text{Com} \), then

separated\( (c_1, c_2) \) if \( FV(c_1) \land WV(c_2) = FV(c_2) \land WV(c_1) = \phi \)

The rules for deriving syntactic validity are as follows:

Guarded Commands

1. \( \text{Com} \ c \quad \text{Com} \ b \rightarrow c \)

2. \( \text{Gcom} \ g_{c_1}, \text{Gcom} \ g_{c_2} \quad \text{Gcom} \ g_{c_1 \cdot c_2} \)
Commands

1. $\vdash_{\text{Comm}} \text{SKIP}$
2. $\vdash_{\text{Comm}} \text{ABORT}$

3. $\vdash_{\text{Comm}} x := e$
4. $\vdash_{\text{Comm}} c_1, \vdash_{\text{Comm}} c_2$
   $\vdash_{\text{Comm}} c_1; c_2$

5. $\vdash_{\text{Geom}} gc$
6. $\vdash_{\text{Geom}} gc$
   $\vdash_{\text{Comm}} \text{if} \ gc \ \text{fi}$
   $\vdash_{\text{Comm}} \text{do} \ gc \ \text{od}$

7. $\vdash_{\text{Comm}} P?W(x)$
8. $\vdash_{\text{Comm}} P!W(e)$

9. $\vdash_{\text{Comm}} c, \text{FPL}(c) = \phi$
   $\vdash_{\text{Comm}} R; c$

10. $\vdash_{\text{Comm}} c$
    $\vdash_{\text{Comm}} \text{process} \ R; c$

11. $\vdash_{\text{Comm}} c_1, \vdash_{\text{Comm}} c_2, \text{separated}(c_1, c_2), \text{FPL}(c_1) \land \text{FPL}(c_2) = \phi$
    $\vdash_{\text{Comm}} c_1 || c_2$

Requirements of non-interference and disjoint process names are imposed in rule 11.

The condition $\text{FPL}(c) = \phi$ in rule 9 demands that subprocesses of a named process must be declared local to that process, so for example

process $R; \ \text{process} \ Q; \ \text{process} \ P;$

$R: (P: \text{SKIP} || Q: \text{SKIP})$  

is not a syntactically valid form whereas

process $R;$

$R: (\text{process} \ P; \ \text{process} \ Q;$

$P: \text{SKIP} || Q: \text{SKIP})$

is a valid form.
3.1.3. Dynamic semantics

Having defined the syntax of CSP and its static semantics, thereby defining a language of CSP programs, we now give the dynamic semantics for commands through their associated transition system.

Definition 3.3

\[ \text{Alab} = \{(N,P)\text{W?v: } N \in \text{Plab}, P \in \text{Plab}, W \in \text{Pten}, v \in \text{Val}\} + \{(N,P)\text{W!v: } N \in \text{Plab}, P \in \text{Plab}, W \in \text{Pten}, v \in \text{Val}\} + \{\epsilon\} \]

the set of action labels, ranged over by \( a \) and \( \alpha \), where \( * \) is introduced as standing for an unknown process name. The intended meaning of these labels is as follows:

\( \epsilon \) represents a 'silent' internal action.

\( (N,P)\text{W!v} \) - the agent named \( N \) outputs value \( v \) together with pattern \( W \) to the agent named \( P \). Patterns allow both senders and receivers to select only communications with that pattern.

\( (N,P)\text{W?v} \) - the agent named \( N \) receives value \( v \) from the agent named \( P \), both using pattern \( W \).

Note in each case \( N \) is the initiator and source of the action.

The complement of an action label \( a \), written \( \overline{a} \), is the action label describing the 'other half' of a handshake communication.
Definition 3.4

\[ a = \begin{cases} 
(P, Q) \text{W?V} & \text{if } a = (Q, P) \text{W?V} \\
(P, Q) \text{W?V} & \text{if } a = (Q, P) \text{W?V} 
\end{cases} \]

Definition 3.5

CSP is the transition system defined by:

\[ \text{config}(CSP) = \{ <c, s>, [-\text{Com} c] + \{ \text{done, abort} s \} \}, \]

ranged over by \( y \). We also allow \( t \) to range over \( \{ c: [-\text{Com} c] + \{ \text{done, abort} \} \}. \) The configuration \( <\text{done, s}> \) corresponds to successful termination with state \( s \), while \( <\text{abortion, s}> \) corresponds to unsuccessful termination through some run-time error (Li did not give a state component to the abortion configuration, using 'abortion' instead, but argued for this new form with the state component [Li 83 page 236]).

\[ \text{labels}(CSP) = \text{Alab} \]

\[ \text{trans}(CSP) = \Rightarrow \]

derived from the transition relation \( \Rightarrow \) to be defined below, by

\[ \Rightarrow \epsilon \]

so \( \epsilon \) is regarded as 'silent' action.

\[ \text{term}(CSP) = \phi \]

since \( y \Rightarrow \epsilon y \) for every \( y, \epsilon \) \( \text{config}(CSP) \). However, we would like to distinguish \( <\text{done, s}>, <\text{abort, s}> \) as somehow different so we introduce the set \( \text{aterm} \) of 'almost terminals':

\[ \text{aterm} = \{ <\text{done, s}>, <\text{abort, s}> \} \]
which we will use as a predicate. Note $\text{atrm}(y)$ implies $y \xrightarrow{a} \theta$ for any $a \in A_{\text{lab}}$.

In defining the transition relation $\rightarrow$ of CSP we will require an auxiliary transition relation

$$
\rightarrow_{gc} c \mid \langle gc, s \rangle : \frac{\rightarrow_{\text{com}} gc}{\text{labels(CSP)}} \times \text{config(CSP)}
$$

which will be given first. The following notation is introduced for brevity;

**Notation 3.1**

$$
\begin{align*}
\frac{r_1 \xrightarrow{a_1} \cdots \frac{r_i \xrightarrow{a_i} \cdots r_n \xrightarrow{a_n}}{r_n}}{r \xrightarrow{a} r_1 | r_2 | \cdots | r_n} \\
\frac{r'_1 \xrightarrow{a'_1} \cdots \frac{r'_i \xrightarrow{a'_i} \cdots r'_n \xrightarrow{a'_n}}{r'_n}}{r' \xrightarrow{a'} r'_1 | r'_2 | \cdots | r'_n}
\end{align*}
$$

is an abbreviation for the $n$ rules:

$$
\begin{align*}
\frac{r \xrightarrow{a} r_i}{r' \xrightarrow{a'} r'_i}
\end{align*}
$$

where $i = 1, \ldots, n$

Before we give the transition rules we briefly indicate the differences with Li's rules [Li 83 pages 67-73].

1. Errors occurring during evaluation of expressions are disallowed, this is introduced by Li in [Li 83 page 231] when proving correctness. The primary reason for this simplification is that CCS does not consider such a possibility in the semantics of the CCS construct 'if $b$ then $p$ else $q$'.

2. The configurations $s$ and abortion employed by Li are replaced by $\langle \text{done}, s \rangle$ and $\langle \text{abort}, s \rangle$ respectively for reasons explained
on page 236 of Li's thesis; essentially that it makes the proof easier for the parallel construct by demanding (via the transition rules) that unsuccessful termination has no deleterious effect on the state. In Li's theory it would also make the translation easier, obviating the need for a separate process to detect abortion and reset the memory (see process R [Li 83 page 235]).

3. In addition to the replacement above we will frequently replace occurrences of s and abortion by <SKIP,s> and <ABORT,s> respectively. This will give us the important useful property that for any action a,

\[ y ^{\geq 2} y' , \text{ aterm}(y') \implies y \epsilon \{<\text{SKIP},s>,<\text{ABORT},s>\} \]

Further, we change one of Li's rules for the repetitive construct from

\[
\begin{align*}
&<\text{gc},s> \xrightarrow{a} \text{do gc od, s} \quad | \quad <\text{abort},s'> \\
&\text{--------------------------} \\
&<\text{do gc od, s} > \xrightarrow{a} <\text{do gc od, s'} > \quad | \quad <\text{ABORT, s}'>
\end{align*}
\]

to

\[
\begin{align*}
&<\text{gc},s> \xrightarrow{a} \text{do gc od, s} \quad | \quad <\text{abort},s'> \\
&\text{--------------------------} \\
&<\text{do gc od, s} > \xrightarrow{a} <\text{SKIP;do gc od, s'} > \quad | \quad <\text{ABORT;do gc od, s'}>
\end{align*}
\]

merely to reduce the number of terms necessary in the bisimulation. This is acceptable since it only introduces more \( \epsilon \) moves into the computations.

4. Finally, we modify the rules for parallel composition to the form used by Li for his translation [Li 83 page 231].

To sum up the changes:

Changes 1 and 4 were made by Li in chapter 6 to make
the translation work.

Change 2 was proposed by Li.

Change 3 is new but virtually trivial, just providing a useful property.

Li made one other change in chapter 6 [Li 83 section 6.2.2], restricting the allowable syntactic forms of guarded commands. The main improvement of our translation is that we show how this restriction may be dropped.

Guarded Commands

guards

\[
\text{eval}(b,s) = \text{tt} , <c,s> \rightarrow y \\
\frac{}{<b \rightarrow c,s> \rightarrow_{gc} y}
\]

alternative

\[
<gc_1,s> \rightarrow_{gc} <c,s'> | <\text{done},s'> | <\text{abort},s'> , i \in \{1,2\} \\
\frac{}{<gc_1 \cdots gc_2,s> \rightarrow_{gc} <c,s'> | <\text{done},s'> | <\text{abort},s'>}
\]

Commands

assign

\[
\text{eval}(e,s) = v \\
\frac{}{<x:=e,s> \rightarrow \langle \text{SKIP},s[v/x]\rangle}
\]

\text{SKIP}

\[
\langle \text{SKIP},s \rangle \rightarrow \langle \text{done},s \rangle
\]

\text{ABORT}

\[
\langle \text{ABORT},s \rangle \rightarrow \langle \text{abort},s \rangle
\]
input
\[ P ? W(x), s \rightarrow [P ? W(x)] \rightarrow s \rightarrow \text{SKIP}, s[v/x] \]

output
\[ \text{eval}(e, s) = v \]
\[ P ? W(e), s \rightarrow [P ? W(e)] \rightarrow s \rightarrow \text{SKIP}, s \]

composition
\[ \begin{array}{c}
\langle c_1, s \rangle \rightarrow \langle c_1', s' \rangle \mid \langle \text{done}, s' \rangle \mid \langle \text{abort}, s' \rangle \\
\langle c_1; c_2, s \rangle \rightarrow \langle c_1', c_2, s' \rangle \mid \langle c_2, s' \rangle \mid \langle \text{ABORT}, s' \rangle
\end{array} \]

conditional
\[ \begin{array}{c}
\langle \text{go}, s \rangle \rightarrow \langle c, s' \rangle \mid \langle \text{done}, s' \rangle \mid \langle \text{abort}, s' \rangle \\
\text{eval} (\text{bool}(\text{go}), s) = \text{ff} \\
\langle \text{if go fi}, s \rangle \rightarrow \langle c, s' \rangle \mid \langle \text{SKIP}, s' \rangle \mid \langle \text{ABORT}, s' \rangle
\end{array} \]

repetition
\[ \begin{array}{c}
\langle \text{go}, s \rangle \rightarrow \langle c, s' \rangle \\
\text{go} \\
\langle \text{do go od}, s \rangle \rightarrow \langle c; \text{do go od}, s' \rangle \\
\langle \text{go}, s \rangle \rightarrow \langle \text{done}, s' \rangle \mid \langle \text{abort}, s' \rangle \\
\text{go} \\
\langle \text{do go od}, s \rangle \rightarrow \langle \text{SKIP}; \text{do go od}, s' \rangle \mid \langle \text{ABORT}; \text{do go od}, s' \rangle \\
\text{eval} (\text{bool}(\text{go}), s) = \text{ff} \\
\langle \text{do go od}, s \rangle \rightarrow \langle \text{SKIP}, s \rangle
\end{array} \]
parallel

1.
\[ <c_1, s> \xrightarrow{a} <c_1', s'> | <\text{done}, s'> | <\text{abort}, s'> \]
\[ <c_1 || c_2, s> \xrightarrow{a} <c_1', || c_2, s'> | <c_2; \text{SKIP}, s'> | <c_2; \text{ABORT}, s'> \]

2.
\[ <c_2, s> \xrightarrow{a} <c_2', s'> | <\text{done}, s'> | <\text{abort}, s'> \]
\[ <c_1 || c_2, s> \xrightarrow{a} <c_1', || c_2', s'> | <c_1; \text{SKIP}, s'> | <c_1; \text{ABORT}, s'> \]

3.
\[ <c_1 || c_2, s> \xleftarrow{\xi} <c_1', || c_2', s'> \]

4.
\[ <c_2, s> \xrightarrow{a} <c_2', s'> , <c_1, s> \xrightarrow{a} <c_1', s'> \]
\[ <c_1 || c_2, s> \xleftarrow{\xi} <c_1', || c_2', s'> \]

Definition 3.6

\[ \text{rename}_R(a) = \begin{cases} 
\epsilon & \text{if } a = \epsilon \\
(R, P)W?v & \text{if } a = (\ast, P)W?v, R = P \\
(R, Q)W?v & \text{if } a = (\ast, Q)W?v, R = Q 
\end{cases} \]

The intent of \text{rename}_R is to specify actions emanating from some command \(c\) as actions of a process \(R::c\). The conditions \(R = P\) and \(R = Q\) ensure that any process trying to communicate with itself will deadlock, see \[Li 83\] page 70 for details.

\[ <c, s> \xrightarrow{a} <c', s'> | <\text{done}, s'> | <\text{abort}, s'> , \text{rename}_R(a) = \alpha \]
\[ <R::c, s> \xrightarrow{a} <R::c', s'> | <\text{SKIP}, s'> | <\text{ABORT}, s'> \]
Definition 3.7

\[
\text{scope}_{R,L}(a) = \begin{cases} 
  a & \text{if } a = c \\
  a & \text{if } a = (N,P)W?v \text{ and } R = N, R \neq P \\
  (*,P)W(v) & \text{if } a = (N,Q)W?v \text{ and } R = N, R = Q \\
  (*,Q)W(v) & \text{if } a = (R,P)W?v \text{ and } P \notin L \\
  (*,Q)W(v) & \text{if } a = (R,Q)W?v \text{ and } Q \notin L 
\end{cases}
\]

where \( N \in \text{Plab}\{c\} \)

The function \( \text{scope}_{R,L} \) is used to implement scoping rules for the process labelled \( R \) by renaming actions of subprocesses also labelled \( R \). The motivation behind the scoping rules is quite subtle and complex, see [Li 83 page 71], but for our purposes in investigating translation correctness the details are not important; their translation is virtually trivial. Note that \( \text{scope}_{R,L}(a) \) is a partial function.

\[
\text{scope} \quad \langle c,s \rangle \xrightarrow{a} \langle c',s' \rangle \mid \langle \text{done},s' \rangle \mid \langle \text{abort},s' \rangle \mid \langle \text{skip},s' \rangle \mid \langle \text{abort},s' \rangle
\]

The following proposition will help to reduce the number of cases we need consider when constructing and verifying the bisimulation,

Proposition 3.1

\( y \xrightarrow{a} y' \), \( \text{aterm}(y') \) implies \( y = \langle \text{skip},s \rangle \) or \( y = \langle \text{abort},s \rangle \)

\[
\text{scope}_{R,L}(a) \quad a \\
\]

3.2. The translation

In this section we describe a translation from the CSP defined in
the previous section to the CCS of chapter 1. The translation is essentially that of Li given in [Li 83 chapter 6] but with one major difference: we make no restriction on the allowable forms of guarded commands. The restriction introduced by Li [Li 83 page 229] constrained guarded commands to one of 4 forms

\[ b \rightarrow \text{SKIP}; c \]
\[ b \rightarrow \text{P?W(x)}; c \]
\[ b \rightarrow \text{PIW(e)}; c \]
\[ \text{gc}_1 \circ \text{gc}_2 \]

This is essentially Hoares original form of guarding ([Hoare 78]) and is usually written, with a single arrow, as (respectively)

\[ b \rightarrow c \]
\[ b, \text{P?W(x)} \rightarrow c \]
\[ b, \text{PIW(e)} \rightarrow c \]
\[ \text{gc}_1 \circ \text{gc}_2 \]

In contrast, the form introduced by Plotkin [Plotkin 82] given in the previous section has no such restriction. The merit of this form is that it is simpler and more general than the original. We will henceforth refer to Plotkins form as strong guarding and Hoares as weak guarding. The problem of translating strong guards lies in the properties of the CCS operator + with respect to 'silent' action, i.e. \( \perp \) moves in CCS. The translation of an alternative guarded command \( \text{gc}_1 \circ \text{gc}_2 \) will be as Li gave it, \( p+q \) where \( p \) is the translation of \( \text{gc}_1 \) and \( q \) that of \( \text{gc}_2 \). Now, if it is the case that \( p \), say, must read from the (simulated) store to see if \( \text{gc}_1 \) could proceed then, in performing the read, a
derivative of \( p \) will replace \( p+q \) i.e. \( gc_1 \) will appear to have been selected. However, it may be found subsequently in the computation from \( p \) that \( gc_1 \) could not in fact perform an action (if, say, it was trying to send to a non-existent receiver) and so should not have been selected since deadlock ensues. Li almost completely solved this problem by moving all the necessary read actions, for his restricted syntactic forms, to the surrounding context of the guarded command, i.e. to the conditional or repetitive command which contains it. This was achieved by defining a set \( GV \) of 'guard variables' of a guarded command whose values must be read before making a selection from the alternative guarded commands (see [Li 83 page 230]). In this section we show how \( GV \), renamed \( TV \) for 'troublesome variables' can be extended to unrestricted guarded commands and used to guide the translation. We will make this all clear with an example when the translation has been given, but we begin with a definition of the troublesome variables of a guarded command.

### 3.2.1. The Troublesome Variables

For each command, and guarded command, we define a set of "troublesome variables" which includes all the variables whose value may need to be known in order to determine the first action(s) of the (guarded) command. This is essentially an extension of Li's set \( GV \) [Li 83 page 230] to cover all commands rather than just \( \text{SKIP}, \ P!W(e) \) and \( P?W(x) \). We define \( TV \) in a tabular form,
A few of the definitions may be unclear in the light of the (loose) prose definition of TV so we examine the tabular definition more closely:

**TV**\(_{b \to c}\) = TV\(b\) + TV\(c\): in order to proceed a guarded command must know the values of all variables in its boolean part \(b\) (to determine its truth value) and all variables necessary to determine the first move of its constituent command (see the rule guards of sec 3.1).

TV\(x:=e\) = FV\(e\): this is a useful fiction since in fact the free variables of \(e\) are not troublesome; the translation of \(x:=e\) cannot deadlock after reading them. However, using this definition allows us to reduce the number of terms in the bisimulation while still being able to prove condition 5.2 of Theorem 3.2. The alternative is to use TV\(x:=e\) = \(\phi\) and add

\[
[1.1] \begin{cases} 
  \langle \text{SKIP}, s[v/x] \rangle, & \text{RS}([T-X]-Y)[\text{store}_x\text{,e,\text{DONE}!}] \text{ with } s : Y:X) \\
  Y \subseteq T-X, & \text{eval}(e,s) = v 
\end{cases}
\]
to $R_{x}(x=\text{e})$ as defined later in section 4.1.

$TV(c_1;c_2) = TV(c_1)$: this is because we are interested only in the first action, which must be performed by $c_1$.

In order to succinctly state the translation we will find it useful to define some useful shorthand descriptions of some CCS agents.

### 3.2.2. Auxiliary CCS definitions

To aid in the specification of the translation we introduce a number of notational identities as 'syntactic sugar' and a single agent definition, $cell_{x}$, for constructing a simulated state.

**Simulating states:** As part of the semantic translation we will need to translate the state component of a CSP configuration to a CCS term. A state is a function from $Vars$ to $Val$ where $Vars = \{x_n: n \in \text{Nat}\}$ is an infinite set of variables. It will be convenient to assume that all the CSP programs under consideration use variables in some initial subset $\{x_n: n \leq k\}$ of $Vars$ for some $k \in \text{Nat}$; by making $k$ large enough we can include any finite set of programs. CSP states will be modelled in CCS by a (finite) product of individual register cells each subscripted by the subscript number of the variable whose value they hold. Each cell can perform update and retrieval functions for its variable, so for every $n \in \text{Nat}$, define

$$cell_n(y) \leftarrow get_{n,y}.cell_n(y)$$

$$store_{n,v'.cell_n(v')$$
Note the actions too are subscripted by the number of the variable whose value the cell holds. Taking the parallel product of all cells with subscripts less than or equal to $k$, each holding the value associated with its variable in the state $s$, we define for each state $s$,

$$[s] = \Pi_{n \leq k} \text{cell}_n (s(x_n))$$

This definition delivers the following desirable properties,

1. $[s] - \text{get}_n \mid s(x_n) > [s]$ for all $n \neq k$.
2. $[s] - \text{store}_n ?v > [s[v/x_n]]$ for all $n \leq k$.
3. $[s]$ has no transitions but 1 and 2.

To save writing a lot of subscripts we will henceforth write $\text{cell}_x$ for $\text{cell}_n$ if $x=x_n$, and similarly for the actions $\text{get}_n$ and $\text{store}_n$.

Using simulated states: to read the values of sets of variables from the store we introduce a shorthand notation $\text{RS}(X)[p]$ for CCS terms defined as follows for any CCS term $p$ and $X \subseteq \text{Vars}$,

$$\text{RS}(X)[p] = \left\{ \begin{array}{ll}
p & \text{if } X=\emptyset \\
\text{get}_n ?x.\text{RS}(X\setminus\{x\})[p] & \text{if } x=x_n \text{ and } x_m \in X\setminus\{x\} \text{ implies } m \leq n.
\end{array} \right.$$  

The idea is that CCS and CSP share the set Vars so $\text{RS}(X)[p]$ will read all the current values of the variables in $X$ from the simulated state and bind them in $p$ to those same variables, and then become this substituted version of $p$. This sort of process clearly needs a simulated state to interact with, so define for each CCS term $p$ and state $s$.  


\[ p \text{ with } s = (p[s]) \backslash \{ \text{Stateact} + \{ \text{DONE, ABORT} \} \} \]

where Stateact = \{ get_{x, \text{store}_{x}} : x \in \text{Vars} \}. Thus in 'p with s', p has exclusive access to the simulation of s and cannot signal termination (it will be part of the function of the non-behavioural relation to detect termination in other ways). As a notational convenience define for \( X \subseteq \text{Vars} \)

\[ p \text{ with } s : X = p[sub(X, s)] \text{ with } s \]

where sub\((X, s)\) is, as defined in chapter 1 section 2.1, the substitution of s\((x)\) for x, if \( x \in X \). Thus for example

\[ RS([x])[p] \text{ with } s : \emptyset \xrightarrow{} p \text{ with } s : \{ x \} \]

Detecting termination: in the translation, successful or abortive termination will be signalled by the actions DONE! and ABORT! respectively. Since the translation is recursive and syntax directed we will need to compose the recursive subtranslations in a manner which takes account of this convention. Firstly we will need to guarantee exclusive access to these signals from the sub-translations by having some process q exclusively monitor the sub-pieces for occurrences of these actions; so define "p monitored by q" by

\[ p \{ q = (p[\text{done,abort/DONE,ABORT}])q) \backslash \{ \text{done,abort} \} \]

To model a sequential composition \( c_1; c_2 \) we would like q to monitor the translation of \( c_1 \) for termination; if it terminates successfully q should begin execution of the translation of \( c_2 \), otherwise abort. As a syntactic sugaring we will write

\[ a \rightarrow p \quad \text{for} \quad a.p \]

So define
\[ p_1 \text{ bef } p_2 = p_1 \{ \text{ done? } \rightarrow p_2 \]
\[ \quad + \]
\[ \quad \text{ abort? } \rightarrow \text{ ABORT!} \} \]

For the more complex parallel case 'par' performs a similar function,

\[ p_1 \text{ par } p_2 = p_1 | p_2 \{ \text{ done? } \rightarrow (\text{ done? } \rightarrow \text{ DONE!} \]
\[ \quad + \]
\[ \quad \text{ abort? } \rightarrow \text{ ABORT!} \} \]
\[ \quad + \]
\[ \quad \text{ abort? } \rightarrow \text{ ABORT!} \} \]

When only one of \( p_1 \) and \( p_2 \) has terminated we get an intermediate configuration from \( p_1 \text{ par } p_2 \); 'parD' if the termination was successful and 'parA' otherwise:

\[ p_1 \text{ parD } p_2 = p_1 | p_2 \{ \text{ done? } \rightarrow \text{ DONE!} \]
\[ \quad + \]
\[ \quad \text{ abort? } \rightarrow \text{ ABORT!} \} \]
\[ p_1 \text{ parA } p_2 = p_1 | p_2 \{ \text{ done? } \rightarrow \text{ ABORT!} \]
\[ \quad + \]
\[ \quad \text{ abort? } \rightarrow \text{ ABORT!} \} \]

At first glance it might seem that a simpler definition of \( p_1 \text{ parA } p_2 \) might be

\[ p_1 \text{ parA } p_2 = p_1 | p_2 \{ \text{ ABORT!} \]

but this would not accord with the definition of CSP parallel composition wherein termination of the whole depends on the termination of both components; this latter definition would allow unsuccessful termination whenever just one component aborted.

3.2.3. The translation

We are now in a position to give the (syntactic) translation of commands, but first a few words on the use of subscripts and troublesome variables. The translation function for commands and
guarded commands is subscripted by an arbitrary set $X$ of variables. It is assumed that if the set $X$ is non-empty then the translation function is being employed to translate a sub-piece of some larger construct wherein, during execution, the surrounding context of this sub-piece will bind values for these variables before attempting its execution. Consider the translation of the output command:

$$[[P!W(e)]_X = RS(FV(e)-X)[(*, P)Wie.DONE1]$$

The use of '$FV(e)-X$' reflects that it will not be necessary to bind values to the free variables of $e$ occurring in $X$ since they can be assumed already bound with the correct values. The subscript sets $X$ are determined, recursively, during the translation process in the clauses for conditional and repetitive commands. Essentially, in translating, say, 'if go fi' the idea is to make sure all the troublesome variables of go will be bound before execution reaches the translation of go, then translate go under that assumption. This will ensure, by the definition of troublesome variables, no communications with the simulated store will be necessary to determine the first action of go. We return to reinforce this point in an example after the translation has been given.

**Definition 3.9**

Suppose $X \subseteq \text{Vars}$, then

$$[[.]_X : \text{Syn} \rightarrow \text{config}(L[CCS])$$

are defined as follows; (We write $[.]$ for $[[.]_\phi$)
GUARDED COMMANDS

\[[b \rightarrow c]_X\] - if \( b \) then \([c]_X\)

\[[gc_1, gc_2]_X\] - \([gc_1]_X + [gc_2]_X\)

COMMANDS

\[[\text{SKIP}]_X\] - 1.DONE!

\[[\text{ABORT}]_X\] - 1.ABORT!

\[[x:=e]_X\] - RS(FV(e)-X)[store_x \{e\}.DONE!]

\[[c_1; c_2]_X\] - \([c_1]_X\) bef \([c_2]_X\)

\[[P?W(x)]_X\] - \([\ast, P]W(x).\text{store}_x \{x\}.\text{DONE!}\]

\[[P!W(e)]_X\] - RS(FV(e)-X)[\([\ast, P]W(e).\text{DONE!}\]

\[[\text{if gc fl}]_X\] - RS(TV(gc)-X)[if \( \text{Bool}(gc) \)

then \([gc]_{TV(gc)+X}\]

else 1.ABORT!]

To define the translation of an iterative command we introduce

for each \( gc \in \text{Gcom} \) the following definition;

\( D_{gc} \leq RS(TV(gc))[if \text{Bool}(gc) \)

then \([gc]_{TV(gc)+X}\) bef \( D_{gc} \)

else 1.DONE!\]

Then,

\[[\text{do gc od}]_X\] - RS(TV(gc)-X)[if \( \text{Bool}(gc) \)

then \([gc]_{TV(gc)+X}\) bef \( D_{gc} \)

else 1.DONE!\]

\[[c_1 | c_2]_X\] = \([c_1]_X\) par \([c_2]_X\)

\[[R::c]_X\] = \([c]_X[\text{ren}_R]\)

where \( \text{ren}(R) \) is the morphism defined by
\[ \text{ren}(R)(a) = \begin{cases} \text{rename}_R(a) & \text{if } a \in \text{Alab} \\ a & \text{otherwise} \end{cases} \]

\[ [[\text{process } R; c]]_X = [[c]]_X[[\text{sco}(R, c)]] \]

where \( \text{sco}(R, c) \) is the morphism defined by

\[ \text{sco}(R, c)(a) = \begin{cases} \text{scope}_{R, FPL}(c)(a) & \text{if } a \in \text{Alab} \\ a & \text{otherwise} \end{cases} \]

An important corollary of this definition is:

**Corollary 3.1**

\[ \forall c \in \text{Com. } \forall X \subseteq \text{Vars. } \text{FV}([[c]]_X) \subseteq X \]

Proof by structural induction on \( c \).

The semantic translation follows quite simply from the syntactic translation:

**Definition 3.10**

\[ [[\cdot]]_{\text{sem}} : \text{config}(\text{CSP}) \to \text{config}(L[\text{CCS}]) \]

is defined by

\[ [[c, s]]_{\text{sem}} = [[c]] \text{ with } s \]
\[ [[\text{done}, s]]_{\text{sem}} = \text{DONE! with } s \]
\[ [[\text{abort}, s]]_{\text{sem}} = \text{ABORT! with } s \]

As usual we drop the subscript 'sem' since no confusion can arise.

**3.2.4. An example command translated**

As an example of the functioning of the subscript sets we examine the translation of a simple command 'example_c' which is
disallowed in Li's CSP:

\[
\text{example}_c = \begin{cases}
\text{if true} & \rightarrow \text{if true} \rightarrow \text{PIW}(x) \ \text{fi} \\
\text{true} & \rightarrow \text{SKIP} \\
\text{fi}
\end{cases}
\]

The translation of \text{example}_c is then

\[
\text{RS}(\{x\})[ \begin{cases}
\text{if true or true} \\
\text{then} \begin{cases}
\text{false} & \rightarrow \text{if true} \rightarrow \text{PIW}(x) \ \text{fi} \\
\text{true} & \rightarrow \text{SKIP} \\
\text{fi} \\
\text{else} 1. \text{ABORT} \\
\end{cases}
\end{cases}
\]

Clearly there is some room available for optimisation here but the point can still be discerned if we consider an equivalent (in fact congruent) agent \text{eq}_c derived from \text{example}_c by replacing 'if true then p else q' by 'p' and equating 'true or true' with...
I true':

eq_c = RS([x])[(*, P)Wlx.DONE! + 1.DONE!]

The agent that would be produced by Li's translation method, if the syntactic restrictions he imposed were lifted to allow example_c as an argument, is, writing $[.]_{L_1}$ for his translation function,

\[
\begin{align*}
\text{if } & \text{ true} \\
\text{then } & [ \text{true } \rightarrow \text{if true } \rightarrow P!W(x) ]_{L_1} \\
\text{else } & 1. \text{ABORT!}
\end{align*}
\]

- if true or true
  then if true
    then $[ if \text{true } \rightarrow P!W(x) ]_{L_1} +$
    if true
      then $[ \text{SKIP} ]_{L_1}$
    else 1. \text{ABORT!}
  else 1. \text{ABORT!}

- if true or true
  then if true
    then RS([x])[\text{if true }
      then $[ \text{true } \rightarrow P!W(x) ]_{L_1}$
      else 1. \text{ABORT!}]
    +
    if true
      then 1. \text{DONE!}
    else 1. \text{ABORT!}
  else 1. \text{ABORT!}

which has an equivalent (congruent) agent,

eq_{bad} = RS([x])([* , P)Wlx.DONE!]
  +
  1.DONE!
Comparing eq_c and eq_bad we see that 'eq_bad with s' can, in reading the value of x, select the output command for execution before executing it, i.e.

$$\text{eq_bad with } s \xrightarrow{5} ((*, P) \text{W!x.DONE! with s})[s(x)/x]$$

and the derivative cannot be paired in a bisimulation with any derivative of \( \langle c, s \rangle \). In contrast 'eq_c with s' reads the value of x before attempting a choice, avoiding the trap wherein reading x makes the choice.

3.3. The statement of correctness

To give a simple statement of correctness we first identify a class of contexts, which help to characterise agents which have run their course. We begin by considering the following question: if

$$\langle <c, s> \rangle \xrightarrow{w} p \text{ with } \neg \nu$$

for some \( w \in \text{Alab}^* \), what are the forms \( p \) may take given \( c \) (but not \( s \))? Intuitively, for each \( c \) we want to know when 'p with s' has terminated (rather than deadlocked) and in what way. To do this we define two sets for each (guarded) command \( \omega \),

- \( tf(\omega) \) the terminated forms of the translation of \( c \).
- \( atf(\omega) \) the almost-terminated forms of the translation of \( \omega \); they are willing to signal termination but have not yet done so. These sets are defined from consideration of the translation;
<table>
<thead>
<tr>
<th></th>
<th>$\text{atf}$</th>
<th>$\text{tf}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b \rightarrow c$</td>
<td>$\text{atf}(c)$</td>
<td>$\text{tf}(c)$</td>
</tr>
<tr>
<td>$g_{c_1} \circ g_{c_2}$</td>
<td>$\text{tf}(g_{c_1}) + \text{tf}(g_{c_2})$</td>
<td>$\text{atf}(g_{c_1}) + \text{atf}(g_{c_2})$</td>
</tr>
<tr>
<td>$\text{SKIP}$</td>
<td>${\text{DONE!}}$</td>
<td>${\text{N11}}$</td>
</tr>
<tr>
<td>$\text{ABORT}$</td>
<td>${\text{ABORT!}}$</td>
<td>${\text{N11}}$</td>
</tr>
<tr>
<td>$x := e$</td>
<td>${\text{DONE!}}$</td>
<td>${\text{N11}}$</td>
</tr>
<tr>
<td>$\text{P?W(x)}$</td>
<td>${\text{DONE!}}$</td>
<td>${\text{N11}}$</td>
</tr>
<tr>
<td>$\text{P!W(e)}$</td>
<td>${\text{DONE!}}$</td>
<td>${\text{N11}}$</td>
</tr>
<tr>
<td>$\text{if } g \text{ go } ri$</td>
<td>$\text{atf}(g_0) + {\text{ABORT!}}$</td>
<td>$\text{tf}(g_0) + {\text{N11}}$</td>
</tr>
<tr>
<td>$c_1 ; c_2$</td>
<td>${p_1, p_2, p_1 {\text{N11}}$</td>
<td>${p_1, p_2, p_1 {\text{ABORT!}}$</td>
</tr>
<tr>
<td></td>
<td>$p_1 \in \text{tf}(c_1)$,</td>
<td>$p_1 \in \text{tf}(c_1)$,</td>
</tr>
<tr>
<td></td>
<td>$p_2 \in \text{atf}(c_2)$</td>
<td>$p_2 \in \text{tf}(c_2)$</td>
</tr>
<tr>
<td>$\text{do } g_{c_0} \text{ od}$</td>
<td>${p; q, p {\text{ABORT!}}, \text{DONE!}}$</td>
<td>${p; q, p {\text{N11}, \text{N11}}$</td>
</tr>
<tr>
<td></td>
<td>$p \in \text{tf}(g_{c_0})$,</td>
<td>$p \in \text{tf}(g_{c_0})$,</td>
</tr>
<tr>
<td></td>
<td>$q \in \text{atf}(\text{do } g_{c_0} \text{ od})$</td>
<td>$q \in \text{tf}(\text{do } g_{c_0} \text{ od})$</td>
</tr>
<tr>
<td>$c_1</td>
<td></td>
<td>c_2$</td>
</tr>
<tr>
<td></td>
<td>$p \in \text{tf}(c_1)$,</td>
<td>$p \in \text{tf}(c_1)$,</td>
</tr>
<tr>
<td></td>
<td>$q \in \text{tf}(c_2)$</td>
<td>$q \in \text{tf}(c_2)$</td>
</tr>
<tr>
<td>$R := c$</td>
<td>${p[\text{Ren}(R)]}$</td>
<td>${p[\text{Ren}(R)]}$</td>
</tr>
<tr>
<td></td>
<td>$p \in \text{atf}(c)$</td>
<td>$p \in \text{tf}(c)$</td>
</tr>
<tr>
<td>$\text{process } R; c$</td>
<td>${p[\text{SCO}(R,c)]}$</td>
<td>${p[\text{SCO}(R,c)]}$</td>
</tr>
<tr>
<td></td>
<td>$p \in \text{atf}(c)$</td>
<td>$p \in \text{tf}(c)$</td>
</tr>
</tbody>
</table>
There are two points to note, firstly there are no action operators in $t_f(w)$ for any $w$, and secondly for each $p \in atf(w)$ there is exactly one action label, either DONE! or ABORT!. Thus we can define a function 'lastact' from almost-terminated forms, delivering this action label and we can introduce predicates for recognising terminal and almost-terminal forms.

**Definition 3.12**

- **null(p)** if $\exists \omega. \ p \in tf(\omega)$
- **nulld(p)** if $\exists \omega. \ p \in atf(\omega), \ lastact(p) = DONE!$
- **nula(p)** if $\exists \omega. \ p \in atf(\omega), \ lastact(p) = ABORT!$

The following proposition supports the intent of the previous definitions.

**Proposition 3.2**

1. **nulld(p)** implies
   - a) There is a unique $q. \ p \xrightarrow{DONE!} q$
   - b) $p \xrightarrow{\beta} \ implies \ a = DONE!$

2. **nula(p)** implies
   - a) There is a unique $q. \ p \xrightarrow{ABORT!} q$
   - b) $p \xrightarrow{\beta} \ implies \ a = ABORT!$

We are now in a position to state the relation 'result' which constitutes the non-behavioural component of the correctness
criteria. Basically it demands that if an almost-terminal configuration of CSP is paired with some CCS agent then that agent is of the form 'p with s' where s is the state component of the CSP configuration. Further, p has only a single action left; DONE! if the CSP configuration is a successful termination, ABORT! if it is unsuccessful. Conversely, any pairing (y, p with s) such that p is willing to signal termination (by DONE! or ABORT!) must guarantee that y is a configuration very close to the appropriate termination (with the same state) and p is 'dead' after its signal.

For simplicity we make no demands in result for deadlocked CSP configurations, nor demands about convergence, though the translation would support interesting results for these notions.

Definition 3.13

Define result C config(CSP) x config(L[CCS]) by

\[ y \text{ result } p \text{ with } $ \]

iff

1. \( y = <\text{done}, s> \) implies \( s=\$, nuld(p) \)
2. \( y = <\text{abort}, s> \) implies \( s=\$, nula(p) \)
3. \( p \xrightarrow{\text{DONE!}} \) implies \( y \in \{<\text{done}, \$>, <\text{SKIP}, \$>\}, \ nuld(p) \)
4. \( p \xrightarrow{\text{ABORT!}} \) implies \( y \in \{<\text{abort}, \$>, <\text{ABORT}, \$>\}, \ nula(p) \)

\[ \square \]

Theorem 3.1

The translation \([.]): \text{config(CSP)} \rightarrow \text{config(L[CCS])}\) is result-good.

Proof: The proof is the subject of the next section.

\[ \square \]
3.4. The proof of correctness

The purpose of this section is to construct a result-bisimulation \( R \subseteq \text{config}(\text{CSP}) \times \text{config}(\text{L}[\text{CCS}]) \) such that \( y R [y] \) for every \( y \in \text{config}(\text{CSP}) \). The construction will proceed by structural induction on the possible forms of CSP commands: the bisimulation we construct for configurations of the form \( y = \langle c_1, c_2, s \rangle \), say, will assume constructed bisimulations for the configurations of the form \( \langle c_1, s \rangle \) and \( \langle c_2, s \rangle \). The required bisimulation \( R \) is then the union of the bisimulations for each command \( c \).

Before presenting the bisimulations we need a definition necessary for synthesising and analysing computations of \( c_1 || c_2 \) in terms of computations of \( c_1 \) and computations of \( c_2 \). The notion of the merging of states, given here as a partial function \( * \), is intimately connected with the notion of separation of processes. Reviewing Definition 3.2 we recall two commands \( c_1, c_2 \) altering a common store, for each \( c_i \) (\( i \in \{1, 2\} \)) the set of variables whose value may be altered was \( \text{WV}(c_i) \), those referred to at all was \( \text{FV}(c_i) \), and the definition of separation stated

\[
\text{WV}(c_i) \cap \text{FV}(c_j) = \emptyset, \quad \{i,j\} = \{1, 2\}.
\]

This can be represented pictorially with a Venn diagram of the set \( \text{Vars} \).

![Venn Diagram](attachment://venn_diagram.png)
Now suppose $c_1, c_2$ proceed in parallel, altering a common initial state 'init-s' to $s_1$ and $s_2$ respectively. Then the changes made by $c_1$ in $s_1$ are invisible to $c_2$, and vice versa. We can now define for such states $s_1, s_2$ a composite state $s$ which is the same as $s_1$ and $s_2$ where they both agree (i.e. on Vars-WV($c_1$)-WV($c_2$)) but like $s_1$ on WV($c_1$) and $s_2$ on WV($c_2$). Thus $s$ collects the changes made in $s_1$ and $s_2$ and is well-defined since the changes are on disjoint sets.

**Definition 3.14**

Suppose $W_1 \subseteq \text{Vars } 1 \cup 2$,

\[
W_1 \cap W_2 = \phi,
\]

\[s_1(x) = s_2(x) \text{ if } x \in W_1 \cup W_2.\]

Define:

\[s_1 \ast s_2(x) = \begin{cases} 
  s_1(x) \text{ if } x \in W_1 \\
  s_1(x) \text{ otherwise}
\end{cases}
\]

A notational point: in all instances of its use at least one argument will be subscripted to disambiguate $s_1, s_2$. For example,

\[**s_j\]

means \[**s_2\] if $j = 2$

or \[s_1**\] if $j = 1$

i.e. the argument order is reversed if $j = 1$. Further, \[\ast\] should more accurately be written \[\ast_{W_1,W_2}\], but in all uses of \[\ast W_1\] and \[W_2\] will be defined in the immediate context (In fact only one use of \[\ast\] is made in the bisimulation, and therein \[W_1\] and \[W_2\] are the written variables of $c_1$ and $c_2$ respectively in a parallel command $c_1 | c_2$).
To extract the constituent commands of a guarded command we define a set 'commands(gc)',

**Definition 3.15**

<table>
<thead>
<tr>
<th>commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>b -&gt; c</td>
</tr>
<tr>
<td>gc_1^\rightarrow gc_2</td>
</tr>
</tbody>
</table>

Before we can give \( R_x(\text{do gc od}) \) we need to define the set of processes resulting from the recursive execution of the translation of 'do gc od'. The idea is that during any simulated iteration of gc in do gc od there may be extant terms from previous iterations. We wish to concentrate on the currently active term 'p', say, so we define for each CCS term p the set 'ITER(p)' of possible forms for p with its environment of debris from previous iterations.

**Definition 3.16**

Let p be any CCS term and gc \( \in \text{Gcom} \), then define

\[
\text{ITER}(p) = \{ r_1 \{ r_2 \{ \ldots r_n \{ p \} \ldots \} : \text{null}(r_i)_{i=1,\ldots,n,\neq 1}\}
\]

and let iter(p) range over ITER(p).

\[ \square \]

Relating this definition to that of terminated forms we see,

**Corollary 3.2**

\( \forall \text{gc} \in \text{Gcom}. \text{tf}(\text{do gc od}) \subseteq \text{ITER}(<1>) \)

\[ \square \]
3.4.1. The bisimulation

We now define a bisimulation \( R_X(c) \) for every \( X \subseteq \text{Vars} \) and \( c \in \text{Com} \), the union over \( X \) and \( c \) of these sets will be the required result-bisimulation. As a notational convenience we allow \( R_X(S) \) where \( S \) is a subset of \( \text{Com} \) to stand for the union over \( c \in S \) of \( R_X(c) \). Also we allow \( R(c) \) to stand for \( R_\emptyset(c) \). Note in each set of pairs numbered \([n]\), for some \( n \), there is an implicit universal quantification over unbound variables.

Definition 3.17

END is the following set of pairs;

\[
\{(\text{SKIP}, s), (\text{done}, s)\}, \quad \{\text{nuld}(p)\} \\
+ \{(\text{ABORT}, s), (\text{abort}, s)\}, \quad \{\text{nula}(p)\}
\]

Definition 3.18

\[ R_X(\text{SKIP}) \]

[1] \( \{(\text{SKIP}, s), \text{1.DONE! with s}\} \)
[2] END

\[ R_X(\text{ABORT}) \]

[1] \( \{(\text{ABORT}, s), \text{1.ABORT! with s}\} \)
[2] END

\[ R_X(x:=e) \]

[1] \( \{(x:=e, s), \text{RS((FV(e)-X)-Y)[store}_{x \text{e.DONE!}} \text{with s : Y:X}}\), \quad Y \subseteq \text{FV(e)-X}\} \)
[2] END
Here the difference between the two pair forms [1.1] and [1.2] arises because on the first iteration (only) we can assume values will be bound for variables in X.
[1.1] \{<\text{do gc od, s}>, \text{RS}((T-X)-Y)\text{if Bool(gc)}
\quad \text{then } [g_c]_{T+X} \text{ bef D}_{gc}
\quad \text{else 1.DONE! } \}
\quad \text{with } s : Y : X:
\quad Y \subseteq T-X \}

[1.2] \{<\text{do gc od, s}>, \text{iter}(D_{gc}) \text{ with s}>,
\quad \{<\text{do gc od, s}>, \text{iter}(\text{RS}(T-Z)\text{if Bool(gc)}
\quad \text{then } [g_c]_T \text{ bef D}_{gc}
\quad \text{else 1.DONE! })\}
\quad \text{with } s : Z:
\quad Z \subseteq T \}

[2.1] \{<\xi;\text{do gc od, s}>, p \text{ bef } D_{gc} \text{ with } $>:
\quad <\xi,s> R_{T+X}(\text{commands(gc)}) p \text{ with } $ \}

[2.2] \{<\xi;\text{do gc od, s}>, \text{iter}(p \text{ bef } D_{gc}) \text{ with } $>:
\quad <\xi,s> R_{T}(\text{commands(gc)}) p \text{ with } $ \}

[3] END

R_{X}(c_1||c_2) \text{ Let } * = ^{w_1,w_2,}$

[1] \{<\xi_1||\xi_2,s>, p_1 \text{ par } p_2 \text{ with } $>:
\quad <\xi_1,s_1> R_{X}(c_1) p_1 \text{ with } $_1._i=1,2
\quad s_1*s_2 = s, $_1*$_2=$ \}

[2] \{<\xi;\text{SKIP,s}>, p_1 \text{ parD } p_2 \text{ with } $>:
\quad \text{null}(p_1),
\quad <\xi,s_j> R_{X}(c_j) p_j \text{ with } $_j,
\quad [i,j]=\{1,2\},
\quad s*s_j = s, s*$_j=$ \}
Before we attempt a proof that the sets presented above constitute a bisimulation we note a few simple properties of those sets which will be useful later on, beginning with some definitions. Firstly, in implementing CSP in CCS we have required extra action labels for communication with the simulated store, detection of termination and silent CCS action. Clearly we could add these actions, and others if need be, to labels(CSP) so that labels(CSP) = labels(L[CCS]) as the simple theory required, but its only a minor point so we won't bother.
Definition 3.19
The CCS action labels we have employed in translating CSP are given by

\[ \text{Actimp} = \{ \text{store}_x, \text{get}_x : x \in \text{Vars} \} + \]
\[ 1, \text{DONE}, \text{ABORT}, \text{done}, \text{abort} \} + \]
\[ \{(N,P)W : N \in \text{Plab}^+\{ \text{\#} \}, P \in \text{Plab}, W \in \text{Pten} \} \]

It will be useful to extend the notions of free, written, and read variables, as given in Definition 3.1 for CSP commands, to their translations (and thereby to their derivatives) by consideration of the translation, wherein for any X,

\[ \text{get}_x \in \text{sort}(\mathcal{L}_X) \implies x \in \text{RV}(c) \]
\[ \text{store}_x \in \text{sort}(\mathcal{L}_X) \implies x \in \text{WV}(c) \]

Definition 3.20
Suppose \( \text{sort}(p) \subseteq \text{Actimp} \), define the sets of (simulated) written, read, and free variables by

\[ \text{swv}(p) = \{ x : \text{store}_x \in \text{sort}(p) \} \]
\[ \text{srv}(p) = \{ x : \text{get}_x \in \text{sort}(p) \} \]
\[ \text{sfv}(p) = \text{swv}(p) + \text{srv}(p) \]

Now Definition 3.2 can be accordingly extended with a useful corollary,
Definition 3.21
Suppose $p, q$ are CCS terms, $\text{sort}(p) \subseteq \text{Actimp}$, $\text{sort}(q) \subseteq \text{Actimp}$, then
\[
\text{separated}(p, q) \iff \text{swv}(p) \land \text{sfv}(q) = \text{swv}(q) \land \text{sfv}(p) = \emptyset
\]
\[
\Box
\]
Corollary 3.3
Let $c_1, c_2 \in \text{Com}$, then for any $X \subseteq \text{Vars}$
\[
\text{separated}(c_1, c_2) \implies \text{separated}(\llbracket c_1 \rrbracket_x, \llbracket c_2 \rrbracket_x)
\]
\[
\Box
\]
The following proposition states some useful properties of $R_x(c)$ for any $c \in \text{Com}$,

Proposition 3.3
Suppose $X \subseteq \text{Vars}$, $c \in \text{Com}$, then
1. END $\subseteq R_x(c)$
2. Suppose for some $c \in \text{Com}$ we have $y R_x(c) p$ with $s$, then
   1. $\text{FV}(p) \subseteq X$
   2. $\text{sort}(p) \subseteq \text{Actimp}$

Proof by structural induction on $c$.
\[
\Box
\]

3.4.2. The proof
In this subsection we prove a single theorem, the upshot of which is that $R(c)$ is a result-bisimulation containing the pair $\langle c, s \rangle, \llbracket \langle c, s \rangle \rrbracket$ for every state $s$. That $\llbracket \cdot \rrbracket : \text{config}(\text{CSP}) \rightarrow \text{config}(\text{L[CCS]})$ is result-good is then immediate.
We start with a lemma establishing that the relation END, which by Proposition 3.3 is a subset of every $R_X(c)$, is a result-bisimulation in itself.

Lemma 3.1

END is a result-bisimulation.

\[ \square \]

Theorem 3.2

Suppose $X \subseteq \text{Vars}$, $c \in \text{Com}$, $s \in \text{States}$, then

1. $\langle c, s \rangle R_X(c) \left[ c \right]_X$ with $s : X$

2. $R_X(c) \subseteq \text{result}$

3. Suppose $y R_X(c) p$, then
   1. If $p \xrightarrow{a} q$ then either
      (a) $a = 1$, $y \xrightarrow{c} y' R_X(c) q$
      (b) $a = 1$, $y R_X(c) q$
      (c) $a = 1$, $y' R_X(c) q$
   2. If $y \xrightarrow{a} y'$ then $y \xrightarrow{a} q$, $p \xrightarrow{a} q$, $y' R_X(c) q$

4. Suppose $c = \text{if } gc \text{ fi or } c = \text{do } gc \text{ od}$, $X \subseteq \text{Vars}$, and $TV(gc) = T$ then
   1. $\left[ gc \right]_{T+X} \text{ with } s : T+X \xrightarrow{a} r$, $a = 1$
      implies
      $\exists y. \langle gc, s \rangle \xrightarrow{a} y R_X(\text{commands(gc)}) r$
   2. $\left[ gc \right]_{T+X} \text{ with } s : T+X \xrightarrow{a} r$
      implies
      $\exists y. \langle gc, s \rangle \xrightarrow{a} y R_X(\text{commands(gc)}) r$
   3. $\langle gc, s \rangle \xrightarrow{a} y$
      implies
      $\exists r. \left[ gc \right]_{T+X} \text{ with } s : T+X \xrightarrow{a} r$, $y R_X(\text{commands(gc)}) r$
5. Suppose \( X \subseteq \text{Vars} \) and \( \text{TV}(c) = T \), then

1. \( \llbracket c \rrbracket_{T+X} \) with \( s : T+X \xrightarrow{a} r \), \( a \neq 1 \)
   implies
   \[ \exists y. \ \langle c, s \rangle \xrightarrow{a} y \ R_{T+X}(c) r \]

2. \( \llbracket c \rrbracket_{T+X} \) with \( s : T+X \xrightarrow{1} r \)
   implies
   \[ \exists y. \ \langle c, s \rangle \xrightarrow{E} y \ R_{T+X}(c) r \]

Before launching into the proof of this theorem it is instructive to pause and consider exactly what it says, considering each consequent in turn:

1. This will guarantee that for every configuration of CSP which is not of the form \( \langle \text{done}, s \rangle \) or \( \langle \text{abort}, s \rangle \) we have a bisimulation containing it paired with its translation, relative to any set \( X \) of variables, substituted by the appropriate values for variables in \( X \). Taking \( X = \emptyset \) we get \( \langle c, s \rangle \ R(c) [Kc, s] \) as required by result-goodness. This part is easily proved by looking through the bisimulations for the appropriate pair.

2. This is the requirement that the bisimulation be a result-bisimulation. In establishing this condition we concentrate on the cases involving successful termination, the corresponding unsuccessful cases are proved similarly.

3. A simple inductive argument shows this condition implies the bisimulation closure condition \( R \subseteq F(R) \) for the transition relations \( \rightarrow_{\text{CCS}} \) and \( \rightarrow_{\text{CSP}} \) (i.e. \( \text{trans}(L[\text{CCS}]) \) and
trans(CSP)). We use this stronger form here simply to reduce
the number of cases and definitions required in the proof.
The proof will concentrate on condition 3.1 for exemplary
cases since condition 3.2 is more easily accepted by simple
inspection and the remaining cases introduce no new
techniques or definitions.

We shall employ the abbreviated notation introduced in
chapter 2 for describing the cases in the bisimulation proof,
with the added feature of decorations when proving 3.1 to say
which subcase a, b or c was satisfied.

4. This is where the earlier discussion of troublesome variables
reaches formal fruition; these results relate the first
actions of a guarded command to those of its translation and
are used in proving the conditional and iterative
translations correct.

5. To carry the proof of 4 through an inductive step we will
need a property similar to 4 for commands since the first
step, if any, of 'b -> c' is the first step of c.

Proof: We proceed by structural induction on c, in each case of c
we refer to the set of pairs \( R_x(c)[n] \) as \([n]\);

\( \text{case } c = \text{SKIP} : \)

1. \([1]\) contains the required pairs.
2. Only pairs in END need examination, with immediate confirmation.
3. Obviously there are very few transitions to be considered;

   For 3.1 :
   \([1] \overset{1}{\rightarrow} \text{END} \quad (a) \)
For 3.2:

\[ [1] \xrightarrow{\varepsilon} \text{END} \]

4. Clearly \( c \) is not a conditional or repetitive command.

5. \( \text{TV} (\text{SKIP}) = \emptyset \), so

\[ [\text{SKIP}]_X \text{ with } s : X \rightarrow r \]

implies \( a = 1 \) and \( r = \text{DONE!} \) with \( s \), but

\[ <\text{SKIP}, s> \xrightarrow{\varepsilon} <\text{done}, s> \text{ END } r \]

\textbf{case } c = \text{ABORT} : \text{ the proof is essentially as for SKIP.}

\textbf{case } c = x:=e : 

1. The required pairs are contained in [1].
2. Only END need be examined, with immediate confirmation.
3. The transitions may be examined as follows;

For 3.1:

\[ [1] \xrightarrow{1} [1] \quad (b) \quad \text{Reading variables} \]

\[ [1] \xrightarrow{1} \text{END} \quad (a) \quad \text{The assignment} \]

For 3.2:

\[ [1] \xrightarrow{\varepsilon} \text{END} \]

4. Clearly \( c \) is neither a conditional nor an iterative command.

5. \( \text{TV}(x:=e) = \text{FV}(e) \) so

\[ [x := e]_{T+X} = \text{store}_x ! e \cdot \text{DONE!} \]

thus \( [x := e]_{T+X} \text{ with } s : T+X \rightarrow r \) implies

\( a = 1 \), \( r = \text{DONE!} \) with \( s[v/x] \), \( \text{eval}(e, s) = v \)

Hence we establish 5.2 since

\[ <x := e, s> \xrightarrow{\varepsilon} <\text{SKIP}, s[v/x]> \text{ END } r \]

\textbf{case } c = P?W(x) :

1. Those required pairs are in [1].
2. Only END need be examined, with immediate confirmation.
3. The transitions may be analysed as follows;

For 3.1:

\[\begin{align*}
[1] & \rightarrow \text{ }(\#P)^{W^?V} \rightarrow [2] \quad (c) \\
[2] & \rightarrow \text{ END} \quad (a)
\end{align*}\]

For 3.2:

\[\begin{align*}
[2] & \rightarrow \text{ END}
\end{align*}\]

4. Clearly c is not of the supposed form.

5. TV(P?W(x)) = \emptyset \text{ (so T+X = X)} and

\[\begin{align*}
[P?W(x)]_X \text{ with } s : X \rightarrow r
\end{align*}\]

implies \(v \in \text{Val such that}

\[a = (\#P)V^?V, r = \text{store}_x v.\text{DONE! with } s : X\]

but we can establish 5.1 by

\[\langle P?W(x), s \rangle \rightarrow \text{ (\#P)^{W^?Y} <SKIP, s[v/x]\rangle R_X(c)[2] r}
\]

case c = \#P!W(e):

1. Those pairs required are in [1].

2. Only END need be examined, with immediate confirmation.

3. The transitions may be analysed as follows;

For 3.1:

\[\begin{align*}
[1] & \rightarrow [1] \quad (b) \quad \text{Reading variables} \\
[1] & \rightarrow \text{ (\#P)^{WIY} END} \quad (c)
\end{align*}\]

For 3.2:

\[\begin{align*}
[1] & \rightarrow \text{ (\#P)^{WIY} END}
\end{align*}\]

4. Clearly c is neither a conditional nor an iterative command.

5. TV(P!W(e)) = FV(e), so

\[\begin{align*}
[P!W(e)]_{T+X} = (\#P)!e.\text{DONE!}
\end{align*}\]

and we only need consider case 5.1

5.1 Suppose
\[ [P!W(e)]_{T+X} \text{ with } s : T+X \rightarrow r \]

then \( a = (*)P!W(e), \text{eval}(e,s) = v, r = \text{DONE! with } s \)

Accordingly,
\[ <P!W(e),s> \rightarrow <\text{SKIP},s> \text{ END } r \]

**case** \( c = c_1 ; c_2 \):

1. The required pairs are contained in [1] because by the

   inductive hypothesis

   \[ <c_1,s> \ 	ext{RX}(c_1) \ [c_1]_X \text{ with } s : X \]

   so

   \[ <c_1;c_2,s> \ 	ext{RX}(c_1;c_2)[1] \ [c_1]_X : X \text{ bef } [c_2] \text{ with } s \]

   but

   \[ [c_1]_X : X \text{ bef } [c_2] \text{ with } s = [c_1]_X \text{ bef } [c_2] \text{ with } s : X \]

   since by Corollary 3.1

   \[ \text{FV}([c_2]) = \phi \]

2. We proceed by cases on the pair forms:
   
   [1]. Clearly \( y = <\text{done},s> \), and \( p \text{ bef } q \rightarrow \text{DONE!} \) for any \( p,q \).
   
   [2]. Clearly \( y = <\text{done},s> \), and \( p \notd q \rightarrow \text{DONE!} \) implies

   \( q \rightarrow \text{DONE!} \) (since \( \text{null}(p) \)) so then \( \xi = \text{SKIP} \)

   and \( \text{nuld}(q) \) (by the inductive hypothesis) and hence

   also \( \text{nuld}(p \notd q) \) since \( \text{null}(p) \).

3. The transitions may be analysed as follows;

   For 3.1 :
   
   [1] \( \rightarrow \) [1] (a,b,c) Executing command \( \xi \).
   
   [1] \( \rightarrow \) [2] (a) \( \xi = \text{SKIP} \).
   
   [1] \( \rightarrow \) END (a) \( \xi = \text{ABORT} \).
   
   [2] \( \rightarrow \) [2] (a,b,c) Executing command \( \xi \).
   
   [2] \( \rightarrow \) END (a) \( \xi = \text{SKIP} \) or \( \xi = \text{ABORT} \).

For 3.2 :
We just take [1] as an exemplar, so suppose
\[ p \xrightarrow{a} [c_2] \text{ with } $ -a \rightarrow r \]
and let \( R_1 = R_x(c_1), R_2 = R(c_2). \)

We proceed by cases on the derivation,

1. \( p \) communicates with the translated state, i.e.

\[ a, p', $' \text{ such that } p \xrightarrow{a} p', [($) \xrightarrow{a} [($)']. ] \]
\[ a = 1, r = p' \text{ bef } [c_2] \text{ with } $' \]

then
\[ p \text{ with } $ -1 \rightarrow p' \text{ with } $' \]

so by the inductive hypothesis

either \( <\xi, s> R_1 p' \text{ with } $' \)

or \[ y. <\xi, s> \xrightarrow{E} y R_1 p' \text{ with } $' \]

In the former case we have immediately
\[ <\xi; c_2, s> R_x(c_1; c_2)[1] r \]

(establishing (b)), and in the latter we must proceed

by cases on \( y; \)

1. \( y = \text{<done, s>}: \) then

\[ \text{nul}(p'), $'=s, \xi = \text{SKIP} \]

from which we see
\[ <\xi, s> \text{ END } p' \text{ with } $' \]

so \( <\xi, s> R_1 p' \text{ with } $' \)

whence we can establish (b) since
\[ <\xi;c_2,s> \xrightarrow{R_x(c_1;c_2)[1]} \]

2. \( y = <\text{abort},s> \): is treated very like the above case.

3. \[ \xi', s'. y = <\xi',s'> \] : then immediately
\[ <\xi;c_2,s> \xrightarrow{E} <\xi';c_2,s'> \xrightarrow{R_x(c_1;c_2)[1]} \]

establishing (a).

2. \( p \) acts on its own, i.e. \( \exists p' \) such that
\[ p \xrightarrow{a} p', r = p' \text{ bef } [c_2] \text{ with } \$

then
\[ p \text{ with } \$ \xrightarrow{a} p' \text{ with } \$

and the proof proceeds as in case 1.

3. \( p \) terminates successfully, i.e. \( \exists p' \) such that
\[ p \xrightarrow{\text{DONE}} p', r = p' \{ [c_2] \text{ with } \$, a = 1 \]

then
\[ \xi = \text{SKIP}, s = \$, \text{null}(p') \]

so we establish (a) since
\[ <\xi;c_2,s> \xrightarrow{E} <c_2,s> \xrightarrow{R_x(c_1;c_2)[2]} \]

4. \( p \) terminates unsuccessfully: this case is very similar to that in 3.

case \( c = \text{if } gc \text{ fi} \):

1. Those required pairs are contained in \( [1] \).

2. For pairs in \( [2] \) we can apply the inductive hypothesis, for pairs in \text{END} the result is immediate; this just leaves pairs in \( [1] \):

[1]. Clearly \( y = <\text{done},s> \) so suppose
\[ RS((T-X)-Y)[if \text{ Bool}(gc) \]

then \[ [gc]_{T,X} \]

else \[ 1. \text{ABORT!} \] \text{ with } s : T+X \xrightarrow{\text{DONE}}

then
\[ [gc]_{T+X} : \text{DONE} \]

so \( \xi \in \text{commands}(gc) \). \( [\xi]_{T+X} : \text{DONE} \)

hence by the inductive hypothesis \((2) \xi = \text{SKIP}, \) but

\[ [[\text{SKIP}]_{T+X} : \text{DONE}] \]

whence our initial supposition was false.

3. The transitions may be analysed as follows;

For 3.1 :

\[ [1] \xrightarrow{1} [1] \quad (b) \quad \text{Reading troublesome variables.} \]

\[ [1] \xrightarrow{a} [2] \quad (a, c) \quad \text{First action of } [[gc]]. \]

\[ [1] \xrightarrow{-} \text{END} \quad (a) \quad \text{All booleans false, so abort.} \]

For 3.2 :

\[ [1] \xrightarrow{a} [2] \]

\[ [1] \xrightarrow{-} \text{END} \]

The only non-trivial case is pairs in [1]; suppose

\( Y \subseteq T-X, \text{RS}((T-X)-Y)[p] \) with \( s \xrightarrow{a} r, \)

\( p = \text{if } \text{Bool}(gc) \) then \( [[gc]]_{T+X} \) else 1.ABORT!

We proceed by cases on the derivation

1. Reading troublesome variables, i.e. \( \dot{y} \in (T-X)-Y \) such that

\( Y' = Y+[x], \ a = 1, \ r = \text{RS}((T-X)-Y')[p] \) with \( s : Y':X \)

then immediately, verifying \((b)\), we have

\( <\text{if } gc \ f1, s> \ R_{Y}(c)[1] \ r \)

2. The boolean parts of all the guards are false and the command aborts, i.e.

\( Y = T-X, \ a = 1, \ \text{eval}(\text{Bool}(gc)[\text{sub}(T+X, s)]) = \text{ff}, \)

\( r = \text{ABORT}! \) with \( s \)

then, verifying \((a)\), we have

\( \text{eval}(\text{Bool}(gc), s) = \text{ff}, \)

\( <\text{if } gc \ f1, s> \xrightarrow{-} <\text{ABORT}, s> \text{ END} \ r \)
3. This is the first action of the translated guarded command, and it isn't a 1, i.e.

\[ [gc]_{T+X} : T+X \text{ with } s \rightarrow r, \ a=1, \]

\[ \text{eval}(\text{Bool}(gc)\text{[sub}(T+X, s)]) = \text{tt} \]

then by 4.1 \( y \) such that

\[ <gc, s> \rightarrow y R_{T+X}(\text{commands}(gc)) r \]

whence we have established (c) since

\[ <\text{if } gc \text{ fi}, s> \rightarrow y R_X(c)[2] r \]

4. This is the first action of the translated guarded command, and it is a 1; this case is very like 3 except that we invoke 4.2 to establish (a).

4. Let \( T = TV(gc) \), then

4.1 We proceed by structural induction on \( gc \);

BASE \( gc = b \rightarrow \xi \): Then let \( R = R_{T+X}(\xi) \) (so \( R \subset R_X(c)[2] \)) and suppose

if \( b \) then \( [\xi]_{T+X} \text{ with } s : T+X \rightarrow r, \ a=1, \)

then by CCS semantics

\[ [\xi]_{T+X} \text{ with } s : T+X \rightarrow r, \ \text{eval}(b\text{[sub}(T+X, s)])=\text{tt}, \]

hence by the inductive hypothesis (5.1) on \( \xi \) and Definition 1.4

\([y, <\xi, s> \rightarrow R r, \text{eval}(b,s)=\text{tt}\]

so by CSP semantics (guards)

\[ <b \rightarrow \xi , s> \rightarrow y R r \]

STEP \( gc = gc_1gc_2 \): Suppose

\[ [gc_1]_{T+X} + [gc_2]_{T+X} \text{ with } s : T+X \rightarrow r, \ a=1, \]

then for some \( i \in \{1,2\}, \)

\[ [gc_i]_{T+X} \text{ with } s : T+X \rightarrow r \]

so by the inductive hypothesis on \( gc_i, \)

\([y, <gc_i, s> \rightarrow y R_{T+X}(c) r\]

and hence by CSP semantics (alternative)
4.2 and 4.3 follow similarly.

5. Let $T = TV(c)$, then $T = TV(gc)$ and $Bool(gc)(sub(T+X,s))$ is closed

5.1. Since $a = 1$ we need only consider the case

$$[[gc]]_{T+X} \text{ with } s : T+X \xrightarrow{a} r$$

then by 4.1

$$\{ y. <gc,s> \xrightarrow{a} y \text{ } R_X(c)[2] r \}

and by semantics of CSP (conditional 1) and Definition 1.4

$$<\text{if } gc \text{ } fi, s> \xrightarrow{a} y$$

5.2. Suppose

$$\text{if } Bool(gc)$$

then $$[[gc]]_{T+X}$$

else 1. ABORT! with $s : T+X \xrightarrow{1} r$

then we have two cases;

1. eval($Bool(gc)(sub(T+X,s))$)=tt, $$[[gc]]_{T+X} \text{ with } s : T+X \xrightarrow{1} r$$:

   By 4.2 we have

   $$\{ y. <gc,s> \xrightarrow{t} y \text{ } R_X(c)[2] r \}

   so by CSP semantics (conditional 1)

   $$<\text{if } gc \text{ } fi, s> \xrightarrow{t} y.$$

2. eval($Bool(gc)(sub(T+X,s))$)=ff, r=ABORT! with $s$:

   Apply CSP semantics, Definition 1.4 and use END.

\[\text{case } c = \text{do } gc \text{ od} :\]

1. Those required pairs are in [1.1].

2. The proof here mirrors those for the conditional and
   sequential commands, the approach is the same.

3. The transitions may be analysed as follows;

   For 3.1:
Reading troublesome variables.

First action of [gc].

No booleans true.

Executing command \( \xi \).

\( \xi = \text{SKIP} \).

\( \xi = \text{ABORT} \).

For 3.2:

\[ [1.n] \rightarrow [2.n] \]

\[ [1.n] \rightarrow [2.n] \]

\[ [2.n] \rightarrow [2.n] \]

\[ [2.n] \rightarrow [1.2] \]

\[ [2.n] \rightarrow [1.2] \]

The proof is very like that given for the alternative construct except for the accretion of debris.

4. The proof is the same as that given for the alternative construct.

5. Let \( T = TV(\text{do } gc \text{ od}) = TV(gc) \);

5.1. Suppose

\[
\begin{align*}
\llbracket gc \rrbracket_{T+X} & \text{ bef } D_{gc} \text{ with } s : T+X \rightarrow r, a = 1 \\
\text{then since } & a = 1 \exists p \text{ such that} \\
& r = p \text{ bef } D_{gc} \text{ with } $,
\end{align*}
\]

\[
\text{eval}(\text{Bool}(gc)[\text{sub}(T+X, s)]) = \text{tt},
\]

\[
\llbracket gc \rrbracket_{T+X} \text{ with } s : T+X \rightarrow p \text{ with } $.
\]

Hence by 4.1,

\[
\exists \xi, s'. <gc, s> \rightarrow <\xi, s'> R_{T+X}(\text{commands}(gc)) p \text{ with } $
\]

since a straightforward inductive proof shows that almost-terminal configurations \( <\text{done},$> \) and \( <\text{abort},$> \) cannot result since \( a = 1 \); so
5.2 Suppose
\begin{align*}
&([g;c]_{T+X} \text{ before } D_g; c \text{ with } s : T+X \rightarrow \rightarrow r) \\
&\text{then, at first sight, we have 3 cases;}
\end{align*}

1. \begin{align*}
&([g;c]_{T+X}[\text{sub}(T+X, s)] \rightarrow \text{DONE} \rightarrow p, r = p \{ D_g; c \text{ with } s \} \text{ such that} \\
&\text{which implies } \xi \in \text{commands}(g;c) \text{ which contradicts the inductive hypothesis (2)}
\end{align*}

2. \begin{align*}
&([g;c]_{T+X}[\text{sub}(T+X, s)] \rightarrow \text{ABORT} \rightarrow \text{is treated similarly.}
\end{align*}

3. \begin{align*}
&([g;c]_{T+X} \text{ with } s : T+X \rightarrow \rightarrow p \text{ with } s, \text{ with } r = p \text{ before } D_g; c \text{ with } s, \text{ then by } 4.2 \text{ } y \text{ such that} \\
&\langle g;c, s \rangle \rightarrow y \Rightarrow R_{T+X}[\text{commands}(g;c)] \text{ p with } s
\end{align*}

We proceed by cases on \( y \);

1. \( y = \langle \text{done}, s \rangle \): then
\begin{align*}
\text{nul}(p'), s = & \$
\end{align*}
so \( \langle \text{SKIP}, s \rangle \text{ END } p \text{ with } \$ \)
whence
\begin{align*}
\langle \text{SKIP}, s \rangle R_{T+X}[\text{commands}(g;c)] \text{ p with } s, \\
\langle \text{do gc od}, s \rangle \rightarrow & \langle \text{SKIP; do gc od}, s \rangle, R_{X}(c)[2.1] \text{ r}
\end{align*}

2. \( y = \langle \text{abort}, s \rangle \text{ : similarly.}

3. \( y = \xi', s'. \ y = \langle \xi', s' \rangle \text{ : then immediately} \\
\langle \text{do gc od}, s \rangle \rightarrow & \langle \xi'; \text{do gc od}, s \rangle, R_{X}(c)[2.1] \text{ r}
\end{align*}
ocase \( c = c_1 | c_2 \)

1. Those required pairs are contained in [1].

2. The proof is very like that for the sequential command and will not be given here.

3. The transitions may be analysed as follows;
For 3.1:

1. \( \rightarrow_{\alpha} \) (a, b, c) Action other than termination.
2. \( \rightarrow_{\alpha} \) One command is SKIP.
3. \( \rightarrow_{\alpha} \) One command is ABORT.

For 3.2:

1. \( \rightarrow_{\alpha} \)
2. \( \rightarrow_{\alpha} \)
3. \( \rightarrow_{\alpha} \)
4. \( \rightarrow_{\alpha} \)

We take as an exemplary case pairs in [1], so suppose

\( p_1 \) par \( p_2 \) with \$ \( \rightarrow_{\alpha} \) \( r \)

and let \( R_1 = R_x(c_1), R_2 = R_x(c_2) \).

We proceed by cases on the derivation:

1. \( p_1 \) communicates with the simulated state, i.e.

\( \alpha, p_1', \$' \) such that

\( p_1 \rightarrow_{\alpha} p_1', [\$] \rightarrow_{\alpha} [\$'] \),

\( a = 1, r = p_1' \) par \( p_2 \) with \$

then by sorts \( \alpha, p_1', \$' \) such that

\( [\$] \rightarrow_{\alpha} [\$'], \$_1 * \$_2 = \$' \)

so

\( p_1 \) with \( \rightarrow_{\alpha} p_1' \) with \$'}
and hence by the inductive hypothesis either

\[ <\xi_1, s_1> \xrightarrow{\text{R}_1} \text{p}_1' \text{ with } $1' \]

or

\[ \downarrow \text{y. } <\xi_1, s_1> \xrightarrow{-\xi} \text{y} \xrightarrow{\text{R}_1} \text{p}_1' \text{ with } $1' \]

In the former case

\[ <\xi_1 | | \xi_2, s> \xrightarrow{\text{R}_X(c)[1]} r \]

and in the latter case we proceed by cases on y;

1. \text{y = <done, }s_1> : then

\[ \text{nuld}(p'), $1' = s_1, \xi_1 = \text{SKIP} \]

from which

\[ <\xi_1, s_1> \xrightarrow{\text{END}} \text{p}_1' \text{ with } $1' \]

so we establish \((b)\) since

\[ <\xi_1 | | \xi_2, s> \xrightarrow{\text{R}_X(c)[1]} r \]

2. \text{y = <abort, }s_1> : is handled similarly.

3. \text{y = <\&, ', }s_1> : then immediately

\[ <\xi_1 | | \xi_2, s'> \xrightarrow{\text{R}_X(c)[1]} r, s_1'*s_2 = s' \]

2. \text{p}_1 \text{ sends a value to } \text{p}_2', \text{i.e. } \text{A, v, p}_1', \text{p}_2' \text{ such that}

\[ \text{p}_1 \xrightarrow{\text{A}\text{v}} \text{p}_1', \text{p}_2 \xrightarrow{\text{A}\text{v}} \text{p}_2', a = 1, \]

\[ r = \text{p}_1' \text{ par } \text{p}_2' \text{ with } $ \]

then

\[ \text{p}_1 \text{ with } $1, \xrightarrow{\text{A}\text{v}} \text{p}_1' \text{ with } $1', \]

\[ \text{p}_2 \text{ with } $2, \xrightarrow{\text{A}\text{v}} \text{p}_2' \text{ with } $2 \]

and hence by the inductive hypothesis \[ <\xi_1', s_1'> \xrightarrow{\text{A}\text{v}} <\xi_1', s_1> \xrightarrow{\text{R}_1} \text{p}_1' \text{ with } $1, \]

\[ <\xi_2, s_2'> \xrightarrow{\text{A}\text{v}} <\xi_2, s_2> \xrightarrow{\text{R}_2} \text{p}_2' \text{ with } $2 \]

whence

\[ <\xi_1 | | \xi_2, s> \xrightarrow{-\xi} <\xi_1' | | \xi_2', s'> \xrightarrow{\text{R}_X(c)[1]} r, \]

\[ s_1'*s_2' = s' \]
3. \( p_1 \) acts on its own, i.e., \( \downarrow p_1' \) such that

\[ p_1 \xrightarrow{a} p_1', \ r = p_1', \ \text{par} \ p_2 \ \text{with} \ s \]

then

\[ p_1 \ \text{with} \ s_1 \xrightarrow{a} p_1' \ \text{with} \ s_1 \]

and from here on we proceed as in case 1.

4. \( p_1 \) terminates successfully, i.e., \( \downarrow p_1' \) such that

\[ p_1 \xrightarrow{\text{DONE!}} p_1', \ r = p_1', \ \text{parD}, \ p_2 \ \text{with} \ s, \ a=1 \]

then

\[ \text{null}(p_1'), \ s_1 = s_1, \ \xi_1 = \text{SKIP} \]

whence we establish (a) since

\[ \langle \xi_1 | \xi_2, s \rangle \xrightarrow{\xi} \langle \xi_2; \text{SKIP}, s \rangle \ \text{R}_X(c)[2] \ r \]

and

\[ s = s_1 * s_2 = s_1 * s_2 = s * s_2 \ (\text{since} \ s_1 * s_2 = s) \]

5. \( p_1 \) terminates unsuccessfully; rather like case 4.

The (symmetric) cases for \( p_2 \) are addressed in exactly the same fashion.

4. Since \( c = c_1 | c_2 \) is neither an alternative nor an iterative command the result is immediate.

5. Let \( T=TV(c_1) + TV(c_2) \);

5.1 Suppose

\[ \llbracket c_1 \rrbracket_{T+X} \ \text{par} \ \llbracket c_2 \rrbracket_{T+X} \ \text{with} \ s : T+X \xrightarrow{a} r, \ a \neq 1 \]

then since \( a \neq 1 \) we must have \( \downarrow p_1, p_2 \) such that

\[ r = p_1 \ \text{par} \ p_2 \ \text{with} \ s \]

\[ \llbracket c_1 \rrbracket_{T+X} \ \text{with} \ s : T+X \xrightarrow{a} p_1 \ \text{with} \ s, \]

\[ p_j = \llbracket c_j \rrbracket_{T+X} \ [\text{sub}(T+X, s)], \]

\[ \{i,j\} = \{1,2\} \]

Now we apply the inductive hypothesis to get

\[ \langle c_1, s \rangle \xrightarrow{a} y \ \text{R}_{T+X}(c_1) \ p_1 \ \text{with} \ s \]
The proof proceeds routinely by cases on $y$.

5.2 Suppose

$$[[c_1]]_{T+X} \text{ par } [[c_2]]_{T+X} \text{ with } s : T+X \xrightarrow{1} r$$

then by the inductive hypothesis (2) neither translated command can have signalled termination, so

$$r = p_1 \text{ par } p_2 \text{ with }$$

Now we have two cases,

1. $[[c_i]]_{T+X} \text{ with } s : T+X \xrightarrow{a} p_i \text{ with }$

   $$[[c_j]]_{T+X} \text{ with } s : T+X \xrightarrow{a} p_j \text{ with } s, a=1$$

   $\{i,j\} = \{1,2\}$:

   Then by the inductive hypothesis $\xi_1, \xi_j$.

   $$\prec c_i,s \succ \xrightarrow{a} \prec \xi_1,s' \succ R_{T+X}(c_i) p_i \text{ with }$$

   $$\prec c_j,s \succ \xrightarrow{a} \prec \xi_j,s \succ R_{T+X}(c_j) p_j \text{ with } s$$

   so

   $$\prec c_1 || c_2,s \succ \xrightarrow{a} \prec \xi_1 || \xi_2,s' \succ R_X(c)[1] r$$

   and we can use [1].

2. $[[c_1]]_{T+X} \text{ with } s : T+X \xrightarrow{1} p_1 \text{ with }$

   $$p_j = [[c_j]]_{T+X} \left[ \text{sub}(T+X, s) \right]$$

   Similarly.

$\square$

This concludes the (fragment of) the proof for Theorem 3.2. It includes the main concepts and techniques required to deal with the remaining cases.

**Corollary 3.4**

$$[[.]] : \text{config(CSP)} \rightarrow \text{config(L[CCS])} \text{ is result-good.}$$

$\square$
4. A testing theory of translation correctness

A basic problem of translation theory is to determine the weakest properties required of a translation in order that there is never any conclusive disagreement over the properties of an object and its translation. To formalise this notion requires a more careful examination of the notions of 'conclusive disagreement' and 'properties'.

As with the bisimulation theory we shall attempt to generalise an existing equivalence relation over CCS; this time we employ the 'testing' equivalence of [de Nicola and Hennessy 84].

4.1. Motivations

Given any system it is a minimal requirement that we have some means of testing, i.e. observing, its properties. Typically this observation may extend over a number of interactions and constitute an interactive dialogue with the system, or it may be a simple static test, or it may be a combination of the two. However, it seems neither necessary nor desirable to demand that the tests performed on the target be identical with those performed on the source; in fact we just need a 'corresponding' test. For example you may simulate the action of an electronic digital computer on a piece of paper; tests on the source will involve button pushing, reading VDU's etc. while those on the target involve rifling through bits of paper. Correctness of a translation will depend on a preservation of certain relationships between objects and the tests performed upon them.
In this section we aim to determine very weak requirements for translation correctness by analysing what is required of a translation in order that we are assured of complete and lasting agreement over the properties of the source objects and their translations. Essentially there are two key points on which the theory will rest; firstly we will not require the existence of a single 'super-observer' to compare the behaviour of source and target objects. Such an approach, which is essentially that taken by the bisimulation theory, would, for example, be difficult to apply in cases where a single observable step of the source object is translated to an action involving many observable steps of the target. Instead it is much simpler to postulate the existence of a 'degenerate' observer monitoring the outcomes (only) of experiments, where an experiment is the application of a relevant test to some object. Relevant tests of course will vary according to the object. For example if the source objects are variously coloured and flavoured gob-stoppers while the target objects are a set of similarly coloured billiard balls then tests for flavour are relevant in the source but not the target. (However, it may be that tests for flavour can be 'coded up' as tests for colour, e.g. 'is-it-strawberry' becomes 'is-it-red'). Thus relevance is determined by applicability and the ability to discriminate between objects presented for examination.

All we will require of observers is that they can distinguish whether the result of some experiment was a success or a failure. We will not distinguish modes of failure in experiments, such as the usual notions of deadlock and divergence. If an experiment does not succeed it just does not succeed; we do not say that it
is deadlocked or divergent or anything else. Note this applies to experiments but not necessarily the objects undergoing testing, for given acute enough tests we may be able to distinguish deadlock and divergence within process objects.

4.1.1. Experiment systems

The theory of experimentation outlined above intentionally does not determine the allowable tests, notions of success, or how tests are to be administered. Thus any particular set of experiments will be parameterised on particular instantiations for these variables.

Definition 4.1

An experiment system Ex is a triple \( \langle E, \rightarrow, \text{success} \rangle \) where

- \( E \) is a set of 'experiments', ranged over by e.
- \( \rightarrow \subseteq E \times E \) is a binary relation (i.e. an unlabelled transition relation).
- \( \text{success} \subseteq E \) is a set of 'successful' experiments.

In defining or analysing an experiment system Ex we may write

\[
\begin{align*}
\text{exp}(Ex) & \quad \text{for } E \\
\text{trans}(Ex) & \quad \text{for } \rightarrow \\
\text{succ}(Ex) & \quad \text{for } \text{success}.
\end{align*}
\]

Thus an experiment system is very similar to a labelled transition system with just one label used all the time, no set of terminals, but with an added notion of success. Given the similarity between
<E, ->, success> and a labelled transition system <E, ->, [ε], φ> we will import many of the notions (such as computations, derivatives etc.) from labelled transition systems without further comment.

Though for much of the theory to be developed it is not necessary to distinguish test from object in an experiment, it is intuitively expected that some test is being performed on some object in order to answer some question. Thus in many cases E will be described in the form

\[ E = T[P] \]

where

- \( T \) is a set of tests, ranged over by \( t \).
- \( P \) is a set of objects, ranged over by \( p \).

\[ T[P] = \{ t[p] : \text{applicable}(t, p) \} \]

\( t[p] \) is the experiment formed by interfacing test \( t \) with object \( p \) in some way to be defined.

applicable is a binary relation governing whether or not some test \( t \) and object \( p \) can be interfaced.

If all tests are applicable to all objects then its definition may be omitted.

As an example construction of an experiment system we show how, in the spirit (if not the letter) of [de Nicola and Hennessy 84], CCS may be adapted to this paradigm.

**Example EXP(CCS)**

We proceed by giving the definitions of experiments, transitions between experiments and success, thereafter turning to their motivation. An experiment is a CCS term, that is we take

\[ \text{exp}(\text{EXP(CCS)}) = \{ t[p] : t, p \in \text{Terms}, \bot t, \bot p \} \]
assuming as in chapter 1 a set of action names Act, a set of procedure names Proc, and a definition set D. We may write $t[p]$ for the experiment $t[p]$.

Recalling the definition of the transition relation $\rightarrow^C$ Terms $\times$ $\{[1]+\text{labels}(\text{L(CCS)})\} \times$ Terms from chapter 1 we introduce the successful experiments by assuming a distinguished action $\omega \in$ Act and defining

$$\text{succ}(\text{EXP}(\text{CCS})) = \{e: e \rightarrow^{\omega} \}$$

An experiment proceeds by the (externally) silent transitions of the CCS term,

$$\text{trans}(\text{EXP}(\text{CCS})) = \rightarrow^1$$

Given that we have made a distinction between test and object in an experiment $t[p]$ what is being tested for? The idea is that $t$ is designed to test (or use) some aspect of $p$'s behaviour, and the experiment is constructed to answer the question "will $p$ satisfy $t$?". For example take


The question then becomes "will $p$ do a receive action on channel A before a receive action on channel B", and the answer is "maybe". How answers are derived and used to compare objects is the subject of the remainder of this chapter.

Note, if the test was 'wrongly' applied, say as

$$t[p] = t + p$$

then you have an answer to a different question. The art of constructing interesting experiments lies in asking the right questions; for CCS the most interesting questions address the communication behaviour of a process so experiments of the form
t|p (in the above system) will provide the most information, t having free access to p. In a sequential language like Pascal processes don't communicate except by altering the state for a subsequent process; thus you might expect experiments of the form p;t where t tests the state resulting from the execution of p. Thus, in summary, experiments frame questions about the properties of their constituent objects. Questions thus posed have answers in the set true, false and maybe.

Returning to the question of where we draw the line between test and object we can see that in the design of most experiments some distinction is intended. However, the progress of an experiment may make no use of that distinction.

For example an experiment to measure the temperature of a phial of ice water by immersing a thermometer must consider the possibility that the object whose temperature is measured is not the body of water alone but rather the water plus a warm thermometer. In CCS t|(p|q) behaves just like (t|p)|q, so is the test t or t|p? We will find this fuzzy distinction between test and object (in EXP(CCS)) very useful in chapter 6.

We end this example by noting that part of the art of designing experiments lies in ensuring that everything determining the outcome is described in the experiment; an experiment should represent a "closed world". This is reflected in our definition of trans(EXP(CCS)) and is crucial to the definition of implementation to follow.
4.1.2. Translations

The position we adopt now is that a translation specifies a mapping between experiments rather than the more usual mapping between objects to be experimented upon, so a translation now contains information not only on the correspondence between objects but also on that between tests on those objects.

Definition 4.2

Let \( \text{EX}_1 = \langle E, \rightarrow_1, \text{success}_1 \rangle \) , \( \text{EX}_2 = \langle F, \rightarrow_2, \text{success}_2 \rangle \) be experiment systems. A translation is a total function \( [.] : E \rightarrow F \).

Note that in this general definition a translation is not required to give any information on what objects are being experimented upon or a correspondence between them.

For the remainder of this chapter we assume the pair \( \text{EX}_1 \) and \( \text{EX}_2 \) of experiment systems, \( e \) ranging over \( E \) and \( f \) over \( F \), together with a translation \( [.] : E \rightarrow F \). It is expected that most translations between experiment systems will be generated in a context-free form, that is if \( E = T_1[P_1] \) and \( F = T_2[P_2] \), then

\[
[ t[p] ] = [t]_T [p]_P
\]

where \( [.]_T : T_1 \rightarrow T_2 \),

\( [.]_P : P_1 \rightarrow P_2 \)

are total functions and applicable\(1(t,p) \) implies applicable\(2([t]_T,[p]_P) \)

However, this is not necessary and indeed the translation of chapter 5 is not of this form.
4.2. Translation correctness

Our intention is that a correct translation should preserve certain properties of experiments or, equivalently in many cases, certain relationships between objects and tests. However, we must appreciate there is some measure of asymmetry between the two experiment systems when judging a translation. This is because the target object is intended as a "faithful transcription" of the source object, but the converse is not in general necessary. Thus we assume the observer has expectations of the properties of the target object discernable by testing; how may he be disappointed?

4.2.1. Games observers play

Translation correctness can be viewed as a two player game in the everyday informal sense. The two players are called the implementer and the observer and the aim of the game (for the observer) is to find mistakes in a translation supplied by the implementer. The aim of the game for the implementer is to avoid losing to the observer.

The field of play is composed of the following entities, provided by the implementer:

1. A set of source experiments, ranged over by e.

2. A set of target experiments, ranged over by f.

3. A translation \([\cdot]\) from source experiments to target experiments so that \([e]\) is claimed to 'implement' e
satisfactorily (i.e. without mistakes).

4. A specification which associates with each source experiment e a set of "computation descriptions", one for each possible computation of e (there being at least one such). A computation description may take many forms, for example "the red light will flash and then it stops", or "successful" etc.

5. A relation corresponds between descriptions of source experiment computations and descriptions of target experiment computations, e.g.

"the red light will flash and then it stops"

corresponds

"the green light goes out and then it explodes".

The implementer having provided all these things the game proceeds as follows:

A. The observer picks a source experiment e.

B. The observer then performs the target experiment [e] to get a target computation (description) and tries to produce a "corresponding" source computation, using corresponds to get a source computation (description) and looking for that in the specification of the possible computations of e.

If no such computation can be found the observer wins.

C. Goto A.

So the game played is essentially
Given a computation of the target experiment produce a "corresponding" source experiment computation from the specification.

Thus the observer must be able to justify the observed computation as corresponding in some way to a particular source computation, saying "This computation I have observed is really that computation" while pointing out the correspondent in the source specification.

Now, crucial to the notion of implementation is the assumption (stated here for the first time) that the observer has no control over the particular computation exhibited during the performance of a target experiment. That is, if an experiment has more than one possible computation then any factors which may determine a particular computation (e.g. the phase of the moon, ambient temperature etc.) are assumed to be unknown to the observer, so to the observer the experiment may appear to be non-deterministic.

Hence we need not require

Given a computation from the specification of the source experiment produce a "corresponding" target experiment computation.

because although we have a specification for the source to which the observer points while justifying the target computation there is no specification for the target which can be pointed to. The only way to produce a target computation to point at is to 'force' the target experiment to produce it. But the observer cannot force it, whenever an experiment runs whatever choices it
has cannot be influenced by the observer.

Perhaps a simple, all too familiar, example will clarify the point:

Your TV set is not working properly; sometimes when you turn it on it works fine, but sometimes it just sits there and fizzes. You seem to have no way of guessing which it will do when you go to turn it on and you cannot divine any way of influencing the outcome. You phone the TV repairman who dutifully rushes to your aid.

"What's the problem?" he says, asking for a specification of what the TV set does.

"Sometimes when I switch it on it works just fine, sometimes it just fizzes" you say, giving him the specification in terms of the experiment to be performed and the, two, possible computations thereof.

"OK mate, lets give it a go" he says, adopting the role of observer now he has the experiment and the specification. He will now attempt to verify that the specification and the implementation (your TV set) correspond.

He switches it on, it works fine. Try again: he switches it off and then on again. Still works fine. Again. Again.... He performs the same experiment many times without observing the computation which fizzes; he is unable to force its occurrence.
Is he justified in denouncing you as a liar? Clearly not, you said it might work and it just so happens that it did so every time. His inability to produce the fizzle is no evidence against the validity of the specification, everything that happened was specified as a possibility. (Incidentally, you might try employing him to switch your TV set on every night; he seems, inadvertently, able to determine your desired computation).

The field of play for the TV set game is:

1. There is only one source experiment, namely the hypothetical switching on of your TV set in a conceptual world.

2. For the single source experiment the description of its allowable computations is:

   "Sometimes it works just fine, sometimes it just fizzes"

3. The single target experiment is the actual switching on of the actual TV set in the real world.

Note we will make the normal simplifying assumption that switching the TV set on and then off has no lasting effect on the TV set, so that each time it is switched on again it is a new experiment rather than a continuation of the old one. This assumption seems to be made by the repairman also.

4. With only one source experiment and one target the translation is clear.

5. The relationship corresponds is that between the description of a phenomenon and the phenomenon itself.
Returning to the general game of translation correctness we now spare a thought for the unhappy lot of the implementer; he cannot win. However, we will define translation correctness in such a way as to guarantee he doesn't lose. Firstly we need to fix ourselves more firmly in the domain of experiment systems by defining the notion of 'correspondence' to be used for computations, and we begin that by extending the notion of success from experiments to experiment computations.

Definition 4.3
Suppose EX = <E,->,success> is an experiment system and <e_k>, K ε comp(EX) is a computation, finite or infinite, in this system. Then define

\[ \text{success}(<e_k>) \iff \exists k \in K. \text{success}(e_k) \]

This seems to be the 'least' definition of success for computations in that we would expect it to imply any other notion.

A specification was said to describe, for each experiment e, its allowable computations in some way. The way we have chosen to describe experiment computations is by their success or failure; so a specification is a pairing of experiments with their possible results.

Definition 4.4
Let EX = <E,->,success> be an experiment system, e ε E, c ε comp(e). Then define

\[ \text{result}(c) = \begin{cases} \top & \text{if success}(c) \\ \bot & \text{otherwise} \end{cases} \]
and define the result-set \( \text{Res}(e) \) by

\[
\text{Res}(e) = \sum_{c \in \text{comp}(e)} \text{result}(c)
\]

Thus a result-set is the set of all possible outcomes (success or failure, \( \top \) or \( \bot \)) of an experiment. Finally, define

\[
\text{Ressets} = \{\{\top\}, \{\bot\}, \{\top, \bot\}\}
\]

the set of all possible result-sets, ranged over by \( R \).

It may appear at first sight that we have chosen to largely ignore the role computation descriptions might play in the general theory by restricting them to one bit (success or failure) of information. However, for our purposes (and probably most others) it will suffice when the tests employed are sufficiently rich to constitute descriptions of object computations. Then an experiment succeeds when the object performs a computation from a set described by the test.

**Proposition 4.1**

Let \( \text{EX} = \langle E, \rightarrow, \text{success} \rangle \), then \( \forall e \in E. \text{Res}(e) \in \text{Ressets} \)

**Proof** It is merely necessary to prove \( R = \emptyset \), which follows immediately since the set \( \text{comp}(e) \) is always non-empty.

The form of specifications is now clear; they define for each experiment \( e \) its result-set, a description of its possible computations.

**Definition 4.5**

Let \( \text{EX} = \langle E, \rightarrow, \text{success} \rangle \), then a specification \( \text{spec} \in \text{SPEC(EX)} \) is a pairing of experiments with their possible results:
\[
\text{SPEC}(\text{EX}) = E \rightarrow \text{Resets}
\]

Note that since specifications always describe hypothetical experiments we can take the specification to be the result-set function \(\text{Res}\) for these experiments.

Having fixed the descriptions of computations in a vocabulary of success and failure common to all experiment systems the notion of correspondence becomes trivial, we can take it to be equality over \(\{T, \bot\}\). Thus \(T\) corresponds \(T\), \(\bot\) corresponds \(\bot\) and no other correspondences hold. In what follows explicit reference to the correspondence relation will usually be dropped since it has been reduced to equality.

Returning to the plight of the implementer we recall that he loses when the observer produces a target computation description which has no correspondent in the specification, that is

\[
\exists e \in E, c \in \text{comp}(\llbracket e \rrbracket). \text{result}(c) \not\in \text{Res}(e)
\]

To avoid this possibility we introduce the notion of 'implementation' in the following definition:

**Definition 4.6**

\[f \text{ implements } e \quad \text{iff} \quad \text{Res}(f) \supset \text{Res}(e)\]

This is the major definition of this chapter.
4.2.2. Comparing results

It will be helpful, both technically and intuitively, to restate the definition of 'implements' in terms of two predicates, \textit{must} and \textit{may}, over experiments corresponding to the necessity and possibility (respectively) of a successful outcome to the experiment. We also employ a predicate \textit{mustnot} to express necessity of failure, but it will become redundant later. All predicates are written postfix.

Definition 4.7

Let \( e \in E \), then

\[
\begin{align*}
\text{e must} & \quad \text{iff} \quad \text{Res}(e) = \{\top\} \\
\text{e mustnot} & \quad \text{iff} \quad \text{Res}(e) = \{\bot\}
\end{align*}
\]

We can express implementation in terms of \textit{must} and \textit{mustnot} as the following proposition shows,

Proposition 4.2

\( f \) \textit{implements} \( e \)

iff

1. \( e \text{ must} \) implies \( f \text{ must} \)
2. \( e \text{ mustnot} \) implies \( f \text{ mustnot} \)

Proof Suppose \( f \textit{ implements} e \), then

\[ \text{Res}(f) \subseteq \text{Res}(e) \]

1. By the definition of \textit{must}

\( e \text{ must} \) implies \( \text{Res}(e) = \{\top\} \)

and, by Proposition 4.1

\[ \text{Res}(f) \subseteq \text{Ressets} \]
so \( \text{Res}(f) = \{T\} \).

2. Similarly.

Now suppose 1 and 2; clearly if \( e \) \text{must} or \( e \) \text{mustnot} then \( \text{Res}(e) = \text{Res}(f) \). Otherwise the result is trivially necessary since then \( \text{Res}(e) = \{T, \bot\} \).

It is easy to see that at least the forward implication is well-founded in intuition; returning to our earlier TV set example if you say the set \text{must} fizzle and it doesn't then the set is not how you described it. Similarly if you say it \text{mustnot} work then it better hadn't. (This should bring us to the conclusion that when constructing specifications implementers would like to avoid claims about certainty as much as possible).

To help with the intuitions of the second implication we introduce the predicate \text{may} which asserts the possibility of a successful computation,

\textbf{Definition 4.8}

\[ e \text{ may} \iff \top \in \text{Res}(e) \]

\[ \Box \]

\textbf{Proposition 4.3}

1. \( \forall e. e \text{ must} \text{ implies} e \text{ may} \)

2. \( \forall e. e \text{ may} \iff e \text{ must/not} \)

\[ \Box \]

Now we can restate Proposition 4.2 as
Proposition 4.4

\[ f \text{ implements } e \quad \text{iff} \quad 1. e \text{ must implies } f \text{ must} \]
\[ 2. f \text{ may implies } e \text{ may} \]

Thus, essentially, implementation promises two things

1. If success is guaranteed by the specification then it is guaranteed by the implementation.

2. If the implementation succeeds then the specification can too.

4.2.3. Complete implementation

Returning once more to our TV set example we notice that a fully functional TV set which always works implements the specification given to the repairman (as does one which always fizzes). Thus we see that the repairman believes the set is more than just an implementation in our defined sense, he believes that every result possible in the specification is possible in the implementation. Hence we would like to reassure the repairman by telling him not just that it implements the specification, but rather that it completely implements the specification.

Definition 4.9

\[ f \text{ completely-implements } e \quad \text{iff} \quad \text{Res}(f) = \text{Res}(e) \]

Thus the TV set which always works is not a complete-implementation of the specification "sometimes it works, sometimes it just fizzes" given to the repairman.
We can recast this definition in a number of ways,

**Proposition 4.5**

The following statements are equivalent:

1. \( f \) completely-implements \( e \)
2. \( f \) implements \( e \), \( e \) implements \( f \)
3. \( e \) must iff \( f \) must, \( e \) may iff \( f \) may

Two sorts of contract are now possible between implementer and observer, viz. "it is an implementation" and "it is a complete-implementation", but it should be noted here that the observer will never (within the rules and conditions of the game as given) be able to show an implementation is not complete. An extended version of the game might involve playing the game the other way around by taking computations of the specification and demanding the implementation produce them; inability to do so would show the implementation was not complete.

**4.2.4. Translation correctness**

With the two available forms of contract between observer and implementer come two notions of translation correctness, one of which subsumes the other.

**Definition 4.10**

Let \([\cdot]: E \rightarrow F\) be a translation, then

1. \([\cdot]\) is an implementation iff \( \forall e \in E. \ [e] \) implements \( e \)
2. \([\cdot]\) is a complete-implementation iff \( \forall e \in E. \ [e] \) completely-implements \( e \)
Proposition 4.6
Any complete-implementation is an implementation.

Probably one of the most important properties we would like for any notion of translation correctness is compositionality, that is if we translate A to B and B to C both correctly then the translation A to C is correct.

Proposition 4.7
Suppose \([\cdot]_1: E \rightarrow F\) and \([\cdot]_2: F \rightarrow G\) are translations, then
1. \([\cdot]_1, [\cdot]_2\) both implementations implies
\([\cdot]_2 \circ [\cdot]_1: E \rightarrow G\) is an implementation.
2. \([\cdot]_1, [\cdot]_2\) both complete-implmentations implies
\([\cdot]_2 \circ [\cdot]_1: E \rightarrow G\) is a complete-implementation.

4.2.5. Implementations in CCS

The notion of implementation has, so far, been concerned solely with experiments but we now extend it so that we may say one object implements another rather than one experiment implements another. Clearly we are now going to distinguish between tests and objects in experiments so suppose
\[\text{Ex} = \langle T[P], \rightarrow, \text{success} \rangle\]
and assume (for convenience) all tests \(t \in T\) are applicable to all objects \(p \in P\).
An object $p$ implements another object $q$ if every application of a test to $p$ implements the experiment applying the same test to $q$.

**Definition 4.11**

1. $p$ implements $q$ iff $\forall t \in T. \ t[p] \ implements \ t[q]$.  
2. $p$ completely-implements $q$ iff  
   
   $\forall t \in T. \ t[p] \ completely-implements \ t[q]$.

**Proposition 4.8**

1. `implements` is a preorder over $P$.
2. `completely-implements` is an equivalence over $P$.

**Examples**

We can now present a few example implementations of processes together with an experiment system $\text{EXPJ}$ similar to $\text{EXP(CCS)}$ but restricted in a way appropriate to the examples. The examples concern the implementation of a simple sorting machine defined in CCS by

$$\text{sort}(M) \gets \text{user?x.sort}(M ++ [x])$$

$$+ \text{ if } M = \emptyset \text{ then user!min(M).sort}(M -- [\text{min}(M)])$$

where $M$ ranges over multisets of natural numbers, and $\text{min}(M)$ is the smallest number occurring in $M$.

Thus a sorting machine is always willing to receive from a user another value to add to its collection, and if its collection is non-empty then it is prepared to deliver to the user the smallest value therein, revising its collection appropriately in both cases.
The experiment system EXPI is designed to allow only very restricted testing of sort(M) and its implementations, for reasons discussed in example 1 which follows the definition of EXP1.

Definition EXPI

Suppose L[CCS] is the transition system for CCS containing all the action names, procedure names and definitions used in the following examples.

The objects to be tested are CCS process descriptions without the distinguished action name ω:

\[ P = \{ p : p \in \text{config}(L[CCS]), \omega \notin \text{FL}(p) \} \]

A test follows a 3 step cycle:

1. As a user offer a value to the machine.
2. As a user accept a value from the machine.
3. Decide, on the basis of some boolean test, whether or not to continue.

We allow a test to succeed by offering ω, so

\[ T \subseteq \text{config}(L[CCS]), \text{ranged over by } t \text{ and defined by} \]

\[ t ::= \text{userIn.user?x.if } b \text{ then } t \mid 1.\omega \]

where n ranges over numerals for natural numbers.

A test is interfaced by putting it in parallel with the object and both test and object must be well-defined terms:

\[ t[p] = t[p] \in \text{config}(L[CCS]) \text{ if } \vdash t \text{ and } \vdash p \]

An experiment proceeds by the usual CCS semantics:

\[ t[p] \rightarrow t'[p'] \text{ iff } t[p] \rightarrow t'[p'] \]

Success, as in EXP(CCS) is an offer of ω.
success(t[p]) \iff t[p] \not\rightarrow

Now, EXPI = \langle T[P], \rightarrow, success \rangle.

Example 1

Whereas sort(\varnothing) is a completely general sorter for multisets of arbitrary size we may only be able to implement a machine of some fixed finite size. For example we can define a degenerate sorting machine for multisets of no more than one element; basically just an "echo machine":

\text{sort1} \leftarrow \text{user}x.\text{user}!x.\text{sort1}

so sort1 can only receive a value and then immediately output it.

Now, since \text{implements} is defined relative to a set of tests (or more precisely relative to an experiment system, see Definition 4.11) we see that in EXP(CCS) sort1 does not implement sort(\varnothing).

For example (in EXP(CCS)) take the test

\[ t = \text{user}!1.\text{user}!2.1.\omega \]

then \[ t[\text{sort}(\varnothing)] \text{ must} \]

but \[ t[\text{sort1}] \text{ mustnot} \]

However, restricting the applicable tests as in EXPI corresponds to "conditions of use" under which the object will function as an implementation, and in EXPI sort1 \text{implements} sort(\varnothing) since the "conditions of use", enshrined in the definition of tests, state that the machine will never be asked to hold more than one value at a time.

In general then, \text{implements} is defined relative to an experiment system EX in which the conditions of use are implicit in the
structure and composition of the experiments allowed. For the statement 'p implements q' to be of use it must be shown that p will be used in an experiment of EX.

Example 2

In this example we will see that if a specification allows a choice to be made in an 'uncontrollable' way then an implementer is perfectly within his rights to make an arbitrary choice once and for all. Consider the, rather odd, sorting machine

\[
sort2(M) \leftarrow \text{user?}x.\text{sort2}(M++[x]) \\
+ \text{user?}x.\text{user!m.sort2}(M++[x]--[m]) \\
+ \text{if } M = \emptyset \text{ then user!min}(M).\text{sort2}(M--[\text{min}(M)])
\]

where \( m = \text{min}(M++[x]) \). This is very similar to 'sort' except that when receiving a new value it can arbitrarily decide to make the user's next interaction a receipt of the smallest value \( m = \text{min}(M++[x]) \) currently held. The somewhat churlish behaviour of \( \text{sort2}(\emptyset) \) is implemented (but not completely implemented) by sort1 in both EXP1 and EXP(CCS) since sort1 always makes its (potentially arbitrary) choice in favour of the second summand above. This is closely analogous to the TV set example given earlier in that a TV set that always works when switched on implements one that also has the (uncontrollable) option of fizzling. This sort of implementation will be addressed more fully in chapter 6.

Example 3

It is possible, in the appropriate conditions, for certain forms of divergence to prevent one process implementing another. The
appropriate conditions will depend on the form of the divergence and would typically constitute conditions on the form of experiment systems. Rather than trying to pursue a general approach we will merely give an example of two processes which are bisimulationally equivalent but only one of which implements the other in EXP(CCS).

\[
\text{sort3}(M) \leftarrow \text{user?x. sort3}(M++[x]) \\
+ \text{1.sort3}(M) \\
+ \text{if } M \neq \emptyset \text{ then user?min(M). sort3}(M--[\text{min}(M)])
\]

then, recalling from chapter 2,

\[
\text{sort}(\emptyset) = \text{sort3}(\emptyset)
\]

but sort3(\emptyset) does not implement sort(\emptyset) in either EXP1 or EXP(CCS) since for example in EXP1,

\[1.\omega! | \text{sort}(\emptyset) \text{ must}\]

but

\[1.\omega! | \text{sort3}(\emptyset) \text{ must}\]

since

\[1.\omega! | \text{sort3}(\emptyset) \rightarrow 1.\omega! | \text{sort3}(\emptyset) \rightarrow ...\]

forms an infinite unsuccessful computation.

4.3. Summary and comparison

In this chapter we have motivated and described an approach to translation correctness based on the testing equivalence for CCS of [de Nicola and Hennessy 84]. Translations were defined as functions between experiments. Thus rather than just translating objects we also require the "conditions of use" to be translated also. A careful analysis of the aims of correctness criteria
resulted in two forms of correctness being defined: implementation and complete-implementation.

It now seems appropriate to compare the theories of translation correctness so far described. Each theory takes a very different approach to defining translation correctness, making a technical comparison both very difficult and of doubtful value. Rather we seek here to illuminate various points where the testing theory differs substantially in approach. We begin with the bisimulation theory:

4.3.1. A comparison with the bisimulation theory

We divide the comparison into two parts: what you gain and what you lose.

Gains by using the testing theory

1. Freedom from fixed actions: the bisimulation theory presupposes a common set, or at best two sets with a bijective mapping between them, of action labels. In the definition of bisimulations (chapter 2 section 1) we find a rigid demanding form of comparison which is far too strict for many reasonable translations; an example of such a translation (and an analysis of why the bisimulation approach won't work) is the subject of chapter 5 where the aim of the translation is the eradication of action labels.

2. Freedom from fixed choice structure: the bisimulation mode of comparison essentially demands that any 'processes' paired in a bisimulation have exactly the same capabilities. As was
shown in [de Nicola and Hennessy 84] this can lead to processes with intuitively the same behaviour having no bisimulation. For example, suppose

\[ p = A_1.B_1 + A_1.C_1, \quad q = A_1.(1.B_1 + 1.C_1), \quad p R q \]

for some R C Terms x Terms a bisimulation, then we can derive a contradiction since

\[ A_1.(1.B_1 + 1.C_1) \xrightarrow{A_1} 1.B_1 + 1.C_1 \]

but

\[ A_1.B_1 + A_1.C_1 \xrightarrow{A_1} r \text{ implies } r = B_1 \text{ or } r = C_1 \]

and clearly in either case \((r, 1.B_1 + 1.C_1)\) cannot be a member of a bisimulation since \(r\) will lack some capability of \(1.B_1 + 1.C_1\) \((r \xrightarrow{B_1} \text{ or } r \xrightarrow{C_1} \) but not both). The point is that \(p\) has made an (uncontrollable) choice sooner than \(q\), the choice being which of \(B_1, C_1\) to offer after \(A_1\). This 'motion' of choice points will occur in the translation of chapter 5 and will be carefully analysed therein.

3. Freedom to implement: the relation \textit{implements} over process objects (Definition 4.11) is a pre-order rather than an equivalence like = over CCS terms. This can make life easier for the implementer in two ways; firstly given a specification he need not \textbf{fully} implement it, as was seen in example 2. Secondly, as we will see in chapter 6, given a particular machine he can construct a specification which abstracts away some particular uninteresting property of the machines behaviour while still retaining the relationship of \textit{implements} between machine and specification.

\textit{Losses by using the testing theory}
The most significant loss is the bisimulation proof technique, for each of the three gains above are nails in the coffin of any attempt to construct a bisimulation. Gains 2 and 3 cannot possibly allow the construction of a bisimulation but gain 1 can be retained in very special circumstances, of doubtful value, as shown in chapter 5 section 2.1 (where the attempt fails by gain 2 at least). The approach in chapter 6 works only where a direct substitution of specification for implementation in an experiment is possible (so it's not much use for proving translations). A more general approach is pursued in chapter 5 section 2 but it doesn't constitute a proof technique, rather a set of proof 'guidelines' or 'targets' are defined as conditions sufficient to guarantee (complete-) implementation under certain circumstances.

4.3.2. A comparison with Li's theory

Li's approach is very different from that of the testing theory, the notion of observation is crucial to the testing theory but virtually irrelevant to a 'correct' translation in Li's sense. All that matters is that any (and all) tips of the translated configurations computation set correspond to tips in the original configurations computation set (see chapter 2) and each has an infinite computation if the other has. In chapter 5 we will see how the testing theory can take on a very similar appearance, inspecting only the tips of computations for success. However, there are crucial differences, most notably that the testing theory is inspecting computation trees of experiments rather than just object configurations; the observation and influence of the test part has no counterpart in Li's theory.
Observation is introduced in the adequacy conditions [Li chapter 5 section 5.3 Definition 5.10] by comparison of action labels, but the example of chapter 5 is not even correct in Li's sense so it cannot satisfy these conditions. A more detailed explanation is given in chapter 5 section 2.
5. A example in the testing theory

The aim of this chapter is to demonstrate a translation which is not 'correct' in any of the operational theories of translation except the testing theory. This is achieved by changing the way communications are performed from the simultaneous matching of complementary labels in the source language to the extended manipulation of a 'communication state' in the target language.

In section 5.1 we first define the source language, going on to define the target language in the light of the desired translation. This defines two labelled transition systems, Source and Target. We then formally present a translation \([.]_1\) from configurations of Source to configurations of Target. This translation is of the form employed in all but the testing theory where translations are functions between experiments. In order to apply the testing theory we define two experiment systems, SEXP and TEXP, derived from Source and Target, and extend the translation to these experiment systems as \([.]_2\).

In section 5.2 we address the correctness problem, first by establishing that \([.]_1\) is not correct for any of the theories given (the testing theory is not directly applicable to \([.]_1\)). Further, we show, with particular reference to the bisimulation theory, why these theories could not be simply extended to cover \([.]_1\). It then remains to articulate the testing theory, proving \([.]_2\) is a complete-implementation.
5.1.1. The Source language

The language chosen as the source for the translation is a version of CSP wherein only the very core is retained so we are not concerned with many details largely irrelevant to the aims of this chapter. In particular the following features have been omitted:

1. Nested parallelism - no processes within a parallel construct may contain a parallel construct. This is reflected in the absence of the BNF rule

\[ c ::= c_1 || c_2 \]

which was present in the CSP of chapter 3. This also removes the need for the 'scoping' construct and, to a lesser extent, the 'renaming' construct.

2. Guarded commands - in anything other than a 'toy' language, which is what we are constructing, some form of conditional construct is necessary. However, it is not necessary to the construction of a translation with the properties we seek and is thus omitted, together with the constructs 'if gc fi' and 'do gc od' employing it.

It should be noted that the removal of the 'do gc od' construct from CSP will have the rather drastic effect of restricting all processes to having only finite computations. This is a serious omission in any language and is tolerated only to simplify presentation and allow the introduction of a set of conditions which, under the finite computations assumption and an assumption of the finality of successful configurations, are sufficient to prove a translation is a complete-implementation.
At the end of this chapter we will see how this technique may be extended to the infinite case.

3. Disjoint naming of processes - in the static semantics for the CSP of chapter 3 the following rule applied a condition 'disjoint(c₁,c₂)' that no two processes in the same parallel construct may have the same process label:

\[ [\vdash c₁, \vdash c₂, \text{separated}(c₁,c₂), \text{disjoint}(c₁,c₂)] \]

We will drop the condition 'disjoint(c₁,c₂)' but retain the others. The reasoning is as follows; given the changes in 1. and 2. the language is now rather too deterministic for our purposes. Removing the disjointness condition introduces nondeterminism by making it possible for there to be many processes competing to make a single communication. For instance the CSP process

\[ <R::S!x || S::R?x || S::R?y, s> \]

is not syntactically valid for the CSP of chapter 3, but it is upon consideration of such processes that our dismissal of the bisimulation approach will, at least partly, rest.

Note this change is different from 1. and 2. in that it introduces processes that could not be defined in the CSP of chapter 3.

In all other (remaining!) aspects the source language is almost identical with the earlier CSP, any differences will be noted and justified as they occur. We proceed as usual in defining a language, starting with the syntax.
The Source language syntax

As usual we assume the provision of a set \( \text{Var} \) of variables, a set \( \text{Proc} \) of process names (ranged over by \( R, S \) in this case) and other entities such as states and evaluation functions all as defined in chapter 1 section 1.1.

There are three major syntactic categories:

**Acom** - a set of 'atomic commands', ranged over by \( \text{ac} \) and defined by
\[
\text{ac} ::= x:=e \mid S!e \mid R?x
\]

**Com** - a set of 'commands', ranged over by \( \text{c} \) and defined by
\[
\text{c} ::= \text{ac;}\text{c} \mid \text{done}
\]

**Prog** - a set of 'programs', ranged over by \( \text{p} \) and defined by
\[
\text{p} ::= R::\text{c} \mid \text{p}_1 \mid \text{p}_2
\]

In what follows we will make an heroic attempt to have our cake and eat it with respect to this definition of Prog. Some descriptions of elements \( p \) of Prog will follow the above definition but others will proceed as if Prog was defined by
\[
\text{p} ::= R_1::\text{c}_1 \mid \ldots \mid R_n::\text{c}_n
\]

This flattened form is justified on the grounds that it facilitates many constructions and arguments wherein \( \mid \mid \) may be treated as an, in some appropriate sense, associative operator.

Thus a program describes a set (to most semantic notions \( \mid \mid \) will appear to be commutative as well as associative) of named commands executing, and perhaps communicating via 'instantaneous handshakes', in parallel. Each command is a linear (perhaps empty) sequence of atomic commands ending in 'done', an analogue of Nil in CCS. As with Nil in CCS we will, typically, drop
trailing occurrences of 'done' from commands; thus

\[ x := e; \text{R?y; done} \]

may be written as

\[ x := e; \text{R?y} \]

Each atomic command is either an assignment statement \((x := e)\) or an action to communicate the value of some expression \(e\) to any command labelled \(S\) \((S!e)\), or else an action to accept the transmission of such a value from any command labelled \(R\) and bind it to the variable \(x\) \((R?x)\).

**The source language static semantics**

In this language the static semantics is very simple, the only constraint applied is that of non-interference through state variables as described in chapter 3 section 1.2. We will restate here the relevant portions of said definition for ease of reference. First we need to restate some auxiliary definitions:

**Definition 5.1**

In order to characterise the subsets of \(V_{ars}\) which may influence (in being read) or be influenced by (in being written to) some command we introduce the following functions \(RV, WV, FV\) from \(Com\) to the powerset of \(V_{ars}\), and define them in the usual tabular fashion:

- \(RV(c)\) - the set of variables \(c\) may read from.
- \(WV(c)\) - the set of variables that \(c\) may write to.
- \(FV(c) = RV(c) + WV(c)\) - the set of free variables of \(c\).
\[ x := e; c \quad S_l e; c \quad R ? x; c \]

<table>
<thead>
<tr>
<th></th>
<th>FV(e)+RV(c)</th>
<th>FV(e)+RV(c)</th>
<th>RV(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RV</td>
<td>FV(e)+RV(c)</td>
<td>FV(e)+RV(c)</td>
<td>RV(c)</td>
</tr>
<tr>
<td>WV</td>
<td>{x}+WV(c)</td>
<td>WV(c)</td>
<td>{x}+WV(c)</td>
</tr>
</tbody>
</table>

and of course \( RV(\text{done}) = WV(\text{done}) = \emptyset \).

\( \Box \)

Now we can go about defining non-interference as before, i.e.

**Definition 5.2**

For \( c_1, c_2 \in \text{Com} \) define

\[ \text{non-inter}(c_1, c_2) \text{ if } FV(c_1) \land WV(c_2) = FV(c_2) \land WV(c_1) = \emptyset \]

Thus if \( c_2 \) can influence variable \( x \) (i.e. \( x \in WV(c_2) \)) then it can be neither read nor written to by \( c_1 \) (i.e. \( x \notin FV(c_1) \)), and vice versa. However, \( c_1 \) and \( c_2 \) may read from the same variable if neither can alter it.

\( \Box \)

Thus we arrive, finally, at the single simple restriction on programs, namely that no two labelled commands in a program may interfere with each other through the medium of the shared state.

Put more formally:

\[ \forall i,j \in \{1, \ldots, n\}, i \neq j. \text{non-inter}(c_i, c_j) \]

\[ \vdash R_1::c_1 \mid \ldots \mid R_n::c_n \]

where \( \vdash \) is the expression of syntactic validity.

**The source language operational semantics**

The next thing to do is introduce the semantic notions of configurations and transitions. Now seems as good a time as any to christen the language so let's call it ... 'Source'. As before
the Source language semantics is given by a labelled transition system. This system will more or less coincide with that given in chapter 2 on this restricted subclass of CSP programs.

Definition 5.3

As in chapter 3 a configuration is a piece of program text together with a binding of values to variables, i.e. a state

\[ \text{config(Source)} = \{<p,s>: | p | \} \]

We will define a non-empty set of terminals for use later in defining success

\[ \text{term(Source)} = \{<R_1::c_1|...|R_n::c_n,s>: c_1 = ... = c_n = \text{done}\} \]

Thus every command in a terminal configuration cannot, and does not wish to, perform any (more) actions.

Now we almost have a labelled transition system, the next step is to define the transition labels,

\[ \text{labels(Source)} = \{\epsilon, R::\text{!}v, S::R?v\} \]

All that remains now is to define \(\text{trans(Source)}\). The definition of \(\text{trans(Source)}\) is as usual by a form of structural induction on the program syntax. Thus, in this case, we are in fact defining a family of transition relations over a number of transition systems since, for instance, \(<x::e,s> \in \text{config(Source)}\). We will finesse this point, dropping all subscripts as usual, in the belief that the reader can make the necessary distinctions.

Atomic commands

\[ \text{eval(e,s)} = v \]

1. \[ <x::e,s> \rightarrow^E <\text{done},s[v/x]> \]
2. \quad \text{eval}(e,s) = v \\
\quad \frac{}{} \quad \langle S! e, s \rangle \rightarrow_{S!} \langle \text{done}, s \rangle \\

3. \quad \langle R? x, s \rangle \rightarrow_{R?} \langle \text{done}, s[v/x] \rangle \\

Commands \\

The configuration \langle \text{done}, s \rangle \text{ has no transitions.} \\

4. \quad \langle a, c, s \rangle \rightarrow_{a} \langle \text{done}, s' \rangle \\
\quad \frac{}{} \quad \langle ac, s \rangle \rightarrow_{a} \langle \text{done}, s' \rangle \\

Programs \\

5. \quad \langle c, s \rangle \rightarrow_{a} \langle c', s' \rangle \\
\quad \frac{}{} \quad \langle R:: c, s \rangle \rightarrow_{R::} \langle \text{done}, s' \rangle \\

6. \quad \langle p_1, s \rangle \rightarrow_{a} \langle p_1', s' \rangle \\
\quad \frac{}{} \quad \langle p_1 || p_2, s \rangle \rightarrow_{a} \langle p_1', || p_2', s' \rangle, \langle p_2 || p_1, s \rangle \rightarrow_{a} \langle p_2 || p_1', s' \rangle \\
\quad \frac{}{} \quad \langle p_1 || p_2, s \rangle \rightarrow_{S!} \langle p_1', || p_2', s' \rangle, \langle p_1 || p_2', s \rangle \rightarrow_{S!} \langle p_1', s' \rangle, \{i,j\} = \{1,2\} \\
\quad \frac{}{} \quad \langle p_1 || p_2, s \rangle \rightarrow_{E} \langle p_1', || p_2', s' \rangle, \langle p_2 || p_1, s \rangle \rightarrow_{E} \langle p_2 || p_1', s' \rangle \\

It should be noted that we can state equivalent forms for 5, 6, and 7 in the \( p_1 || \ldots || p_n \) notation.

5.1.2. Translation motivation \\

We are aiming to construct a translation from the language \( \text{Source} \) to some as yet undefined language \( \text{Target} \), wherein the paradigm of handshake communication is replaced by a mechanism of a rather different form. We will see that a natural and simple translation can be constructed, along with a language \( \text{Target} \) to support it,
that performs this function. Also it will be a complete-
implementation yet present severe difficulties to the other
theories. Thus we now proceed to motivate the translation and
Target language, and later describe why they have the properties
we require.

We start with the very simplest case of communication between just
two processes, that is there are only two processes and they both
know it. The basic paradigm of the translation is to model
handshake communication by placing/removing messages in/from a
common area, i.e. communication will be conducted through shared
variables. Consider a source configuration

\[
<R::S!e;c_1 \mid | S::R?x;c_2 , s>
\]

How do we model the handshake communication that the semantics of
Source calls for? Suppose in our common area we have two 'trays',
in analogy with the in/out trays on many people's desks. Call the
in tray 'send' and the out tray 'sent'. The communication
proceeds as follows in stages 1 to 5:

1. Both trays are empty.

2. R places the value of \( e \) (evaluated in the state \( s \)) into the
   send tray, and waits for something to appear in the sent tray.

3. Meanwhile, S has been waiting for a value to appear in the
   send tray. On the appearance of the value placed there by R
   it is seized by S and stored in the variable \( x \). Now both
   trays are empty again.

4. S now acknowledges its receipt of the value by placing some
token, it doesn't matter what, in the sent tray and is then
free to go about its other business, i.e. $c_2$.

5. Still vigilantly waiting, R spots the token in the sent tray, removes it and, taking that to signal completion of the communication, continues on its way. Both trays are empty yet again, ready for the next communication.

Before we extend this method there are two points which are worth noting and which will remain valid over subsequent developments,

1. Communications are no longer 'instantaneous', they are extended over a number of phases and the particular phase occurring in some configuration is determined by both the contents of the trays and the program text. However, there is an invariant relationship, which we will describe later, between these entities which will be crucial to the proof of complete-implementation.

2. In extending a communication event over a number of phases we have introduced a slight asymmetry between senders and receivers, namely that senders start first and finish last. This 'starting first' will, in the following refinement, be the undoing of any attempt at a bisimulation approach to correctness.

Having demonstrated the simplest case we go on to extend the technique to many communicating processes, some of which have the same name. To cope with many processes we index the trays by the names of the sender and receiver, thus

send becomes $(R,S,send)$

to contain the value $R$ wishes to send to $S$, and
sent becomes \((R,S,\text{sent})\) to contain the acknowledgement that \(S\) wishes to send to \(R\).

Hence we have a number of trays indexed by

\[
\text{Proc} \times \text{Proc} \times \{\text{send, sent}\}
\]

each triple being \((\text{sender}, \text{receiver}, \text{send})\) or \((\text{sender}, \text{receiver}, \text{sent})\).

If now we were to allow each tray to contain an unordered set of values then we would have two problems:

1. Values are associated with particular processes; consider the following configuration:

\[
<\text{R}: \text{S}!e_1; x:= e_3 || \text{R}: \text{S}!e_2 || \text{S}: \text{R}?y, s>
\]

A possible computation using trays as described is:

1. The left \(R\) puts the value of \(e_1\) in tray \((R, S, \text{send})\) and waits.

2. The right \(R\) puts the value of \(e_2\) in tray \((R, S, \text{send})\) and waits.

3. \(S\) takes the value of \(e_2\) from the tray \((R, S, \text{send})\) and puts it in variable \(y\).

4. \(S\) puts a token into the tray \((R, S, \text{sent})\) and finishes.

5. Now the left \(R\) takes the token, leaving \((R, S, \text{sent})\) empty, and finishes by assigning the value of \(e_3\) to \(x\).

This is unacceptable since we now have a configuration where \(y\) has the value of \(e_2\), implying \(S\) communicated with the right(most) \(R\), but \(x\) has the value of \(e_3\), implying that \(S\)
communicated with the left(most) R.

The solution is to have each R provide a unique identifying token with the value it places in the send tray, it is this token which is returned by S in the sent tray and each sending process can only retrieve the token it sent with its value.

2. If the set is unordered we would still be left with the problem of choosing among the alternatives. It would be nicer to have some deterministic mechanism make the choice and to this end we will demand that each tray forms a queue of its contents, i.e. a first-in-first-out (FIFO) list. (Unfortunately the analogy with trays falls down a bit here since an office tray functions as a stack!). Thus senders of the same name to any receiver of a particular name must queue up, and their acknowledgements are similarly queued out.

The translation would still be correct in the sense of the testing theory if we removed this determinism but besides being quite a natural addition it will turn out that this extra determinism will preclude any possibility of applying the bisimulation theory in an acceptable way.

Now that we have informally motivated the translation, and thereby the Target language, we next formalise the Target language and then go on to consider the notion of experimentation to be employed.
5.1.3. The Target language

As mentioned earlier the Target language will share parameters, both syntactic (Proc, Expr, Vars) and semantic (States, eval etc.), with the Source language. The essential difference is in the mode of communication, that is shared variables rather than handshaking. This difference is achieved in two changes to Source, the introduction of a communication state and the substitution (for the handshake commands Rx, Sl) of atomic commands to manipulate that state. In order to define communication states we need first to define queues properly.

**Definition 5.4**

A queue, q, over some set A of queue elements is an ordered finite submultiset of A

\[ q = \langle a_1, \ldots, a_n \rangle, \quad n \geq 0 \]

The set of all such queues is written

\[ \text{Queues}(A) \]

We need some notation for describing queues; suppose \( q = \langle a_1, \ldots, a_n \rangle \), then

1. \( \text{mset}(q) \) is the corresponding unordered multiset:

\[ \text{mset}(q) = [a_1, \ldots, a_n] \]

set(q) is the corresponding unordered set:

\[ \text{set}(q) = \{a_1, \ldots, a_n\} \]

If there are no repetitions in the multiset mset(q) then we write \( \text{norep}(q) \).

2. **empty** is the empty queue (in the case \( n = 0 \)):

\[ q = \text{empty} \quad \text{if} \quad \text{set}(q) = \emptyset \]

3. \( q = a_1 . q' \) if \( n \geq 1 \), \( q' = \langle a_2, \ldots, a_n \rangle \)
i.e. \( a_1 \) is at the front of the queue.

4. \( q = q'.a_n \) if \( n \geq 1 \), \( q' = \langle a_1, \ldots, a_n \rangle \)
   i.e. \( a_n \) is at the end of the queue.

As usual with queues it is expected that removals from a queue are done at the front, and additions at the rear. It is only under such conditions of use that they can (meaningfully) be called queues.

We now define communication states, as semantic entities like normal states, in accord with the earlier informal motivation of trays as queues.

**Definition 5.5**

A communication state \( Q \) is a triple of functions

\[
\begin{align*}
  f &: \text{Proc} \times \text{Proc} \times \text{send} \rightarrow \text{Queues(Nat} \times \text{Expr}) \\
  g &: \text{Proc} \times \text{Proc} \times \text{sent} \rightarrow \text{Queues(Nat)} \\
  h &: \text{Nat} \rightarrow \text{Nat}
\end{align*}
\]

The intent is that the tokens identifying processes uniquely are natural numbers (we will not bother to distinguish numbers from numerals, and let \( t \) (for token) range over \( \text{Nat} \)). Since the domains of \( f \), \( g \), and \( h \) are disjoint, and their elements easily recognised, we just write \( Q \) for each of \( f \), \( g \), and \( h \). Thus

\( Q(R,S,\text{send}) \) is a queue of token-expression pairs, the 'send' tray. We employ expressions here rather than values to facilitate certain aspects of the correctness proof. A translation which placed values rather than expressions on the queues would be correct also.
Q(R, S, sent)- is a queue of tokens (natural numbers), the 'sent' tray.

Q(t)- is a token t'. This is where a receiver whose token is t remembers that the corresponding sender is t' and may be thought of as representing a set of single element 'pending' trays, one for each process (see the dynamic semantics of the atomic command 'rec' later on).

For the purposes of the translation we will need a particular initial communication state which says that no communications are taking place.

Definition 5.6

The communication state initQ is defined by

∀ R, S. initQ(R, S, send) = empty, initQ(R, S, sent) = empty.

∀ t ∈ Nat. initQ(t) = 0.

(It is intended that no process will be assigned the identifying token 0).

The Target language is meant to be as close as possible to the Source language except for the change in communication mode. Thus its description will closely parallel that of Source and it will be fully explained only at points of departure.

The Target language syntax

There are three major syntactic categories as before

Acom2 - a set of 'atomic commands' ranged over by ar, defined by

ar ::= x:=e | off(qvd, <t, e>) | rec(qvd, <t', x>) |
\[ \text{akn}(qvt, <t'>) | \text{wait}(qvt, <t>) \]

where \( qvd \) ranges over \( \text{Proc} \times \text{Proc} \times \{\text{send}\} \) and \( qvt \) ranges over \( \text{Proc} \times \text{Proc} \times \{\text{sent}\} \)

For the intended interpretation of these syntactic entities the puzzled reader should skip to the section on dynamic semantics.

**Com2** — a set of 'commands' ranged over by \( r \) and defined by

\[ r ::= \text{ar};r \mid \text{done} \]

**Prog2** — a set of 'programs' ranged over by \( pr \) and defined by

\[ pr ::= R::r \mid pr_1 | pr_2 \]

As before, in Prog, some descriptions of elements of Prog2 may be written as if Prog2 was defined by

\[ pr ::= R_1::r_1 | \ldots | R_n::r_n \]

---

**The Target language static semantics**

The only syntactic restriction applied in the Target language is the direct analogue of that in Source, viz. non-interference. Thus, in the interests of brevity, we merely redefine the sets of variables written or read:

**Definition 5.7**

<table>
<thead>
<tr>
<th></th>
<th>( \text{off}(qvd, &lt;t, e&gt;) ; r )</th>
<th>( \text{rec}(qvd, &lt;t', x&gt;) ; r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{RV} )</td>
<td>( \text{FV}(e) + \text{RV}(r) )</td>
<td>( \text{RV}(r) )</td>
</tr>
<tr>
<td>( \text{WV} )</td>
<td>( \text{WV}(r) )</td>
<td>( [x] + \text{WV}(r) )</td>
</tr>
</tbody>
</table>
Finally $FV(r) = RV(r)+WV(r)$.

The Target language dynamic semantics

There are quite a few changes here so we'll go a bit slower; we first define the transition system Target

**Definition 5.8**

A configuration is now a triple: a piece of program text, a communication state, and a normal state:

$$\text{config}(\text{Target}) = \{<pr,Q,s>: |- pr\}$$

As in Source, the terminal configurations are those in which all component commands are exhausted (we make no conditions on the communication state, but for exhausted translations we will be able to prove it is initQ).

$$\text{term}(\text{Target}) = \{<R_1::r_1 | \ldots | R_n::r_n,Q,s>: r_1 = \ldots = r_n = \text{done}\}$$

The labels which can appear on arrows are, of course, heavily depleted; communications pass 'silently' through the communication state,

$$\text{labels}(\text{Target}) = \{\varepsilon\}$$

The definition of $\text{trans}(\text{Target})$ (i.e. $\rightarrow$) proceeds in the same muddled fashion as employed in Source, confusing different ternary relations in the same $\rightarrow$ symbol,
Atomic commands

\[ \text{eval}(e, s) = v \]

1. \[ <x := e, s> \rightarrow <\text{done}, s[v/x]> \]

2. \[ <\text{off}(\text{qvd}, <t, e>), Q[q/\text{qvd}], s> \rightarrow <\text{done}, Q[q. <t, e>/\text{qvd}], s> \]

Thus the 'offer' command 'off((R, S, send), <t, e>)' will place the expression \( e \), along with a token \( t \) identifying the process containing this command, on the rear of the queue denoted by \((R, S, \text{send})\).

3. \[ \text{eval}(e, s) = v \]

\[ <\text{rec}(\text{qvd}, <t', x>), Q[t, e]. q/\text{qvd}], s> \rightarrow <\text{done}, Q[q/\text{qvd}, t/t'], s[v/x]> \]

The 'receive' command makes three changes if qvd in the current communication state denotes a non-empty queue:

1. The element \( <t, e> \) is removed from the queue.

2. The value \( v (= \text{eval}(e, s)) \) is assigned to \( x \). Note this introduces a slight subtlety into the interpretation of \( \text{RV}(<\text{off}(\text{qvd}, <t, e>); r) \) since it is not the sender who evaluates the expression but rather the receiver.

3. The token \( t \) of the sender is associated, in the communication state, with the token \( t' \) of the receiver. This is how the receiver remembers the identity of the sender.

4. \[ <\text{akn}(\text{qvt}, <t'>), Q[q/\text{qvt}, t/t'], s> \rightarrow <\text{done}, Q[q. <t>/\text{qvt}, 0/t'], s> \]
The command to 'acknowledge' has two effects

1. The process containing this command will be identified by the token $t'$; $\text{akn}$ will look up, in the communication state, the token $t$ of its associated sender, and place this on the back end of the queue denoted by $qvt$.

2. The association of $t$ with $t'$ is undone. Though this is not strictly necessary it is helpful for future definitions and proofs.

5. $\langle \text{wait}(qvt,t), Q[qvt,t,qvt], s \rangle \rightarrow^E \langle \text{done}, Q[qvt], s \rangle$

On the appearance of the correct token, $t$, at the front of the queue denoted by $qvt$ the 'wait' command can terminate, after removing $t$ from the queue.

**Commands**

6. $\langle a_0, Q, s \rangle \rightarrow^E \langle \text{done}, Q', s' \rangle$

---

**Programs**

7. $\langle r, Q, s \rangle \rightarrow^E \langle r', Q', s' \rangle$

8. $\langle \text{pr}_1 \parallel \text{pr}_2, Q, s \rangle \rightarrow^E \langle \text{pr}_1 \parallel \text{pr}_2, Q', s' \rangle$

9. $\langle \text{pr}_1 \parallel \text{pr}_2, Q, s \rangle \rightarrow^E \langle \text{pr}_1 \parallel \text{pr}_2, Q', s' \rangle$
5.1.4. The translation $[\cdot]_1$

In chapter 4 section 2.4 a translation was defined as a function between experiments so it may seem a little premature to be describing the translation before the experiment systems, but for this example the old notion of translation (given in chapter 2) as a function from source language configurations to target language configurations, extends smoothly to experiments. Thus we will give an 'old-style' translation now, and extend it to cover experiments later.

By now the translation must be quite obvious to the reader; however, it still remains to show how tokens are assigned to processes and thence to their constituent atomic commands. The technique, as in chapter 3, is to subscript the usual context-free translation function with some context-sensitive information, in this case the process label and identifying token of the process that will contain the translated form.

As usual the semantic translation between configurations is generated by a syntactic translation. Strictly speaking we are employing a number of translations differing in argument type, subscripts etc., but again we finesse these minor details.

Definition 5.9

Commands

$$[[\text{done}]](R,t) = \text{done}$$

$$[[x:=e;c]](R,t) = x:=e;[[c]](R,t)$$
Programs

$$[[R_1::c_1] \cdots [R_n::c_n]] = R_1::[c_1]\{R_1\} \cdots R_n::[c_n]\{R_n\}$$

Now, the generated semantic translation between configurations is given by

Configurations

$$[[<p,s>]]_1 = [[[p]], \text{initQ}, s]$$

5.1.5. $[.]_2$ and the experiment systems SEXP, TEXP

Having described the Source and Target languages and their associated transition systems we need a notion of experimentation on configurations from those systems in order that we can apply the testing theory of translation correctness developed in chapter 4. Fortunately the notion of experimentation will be the same for both systems, so we will describe the construction of SEXP in detail and mention TEXP only at points where the corresponding construction of TEXP is unclear. SEXP (TEXP) is, of course, the Source (Target) experimentation system.
An experiment is as usual a test-process pair, the test comes in two pieces;

**Dynamic communication testing**, as described in chapter 4 section 2, wherein another process (as part of the test) communicates with the process being tested in a possibly extended dialogue. If the pair reach some 'agreeable' joint configuration then the second part of the test is applied.

**Final state testing**, an 'agreeable' configuration is successful if its final component state (i.e. binding of values to variables) is a member of some set of states \( P \) which, together with the testing process, completely determines the test.

**Definition 5.10**

We first define the processes to be tested:

\[
\text{OBJ} = \text{config(Source)}, \text{ranged over by obj.}
\]

Each test is a pair; a piece of program text to do the dynamic testing plus a subset of States for the final test:

\[
\text{TST} = \{(p, P) \in \text{Prog}, \text{P} \subseteq \text{States}\}, \text{ranged over by tst}
\]

The application of a test to a process involves merging the two program texts, to be run as a single program:

\[
\text{tst[obj]} = (\langle p_1 || p_2, s \rangle, P)
\]

if \( \text{tst} = (p_1, P), \text{obj} = \langle p_2, s \rangle \), \( p_1 || p_2 \)

The merged program texts must form a valid (i.e. \( \vdash \) from the static semantics of Source) program, otherwise the test might interfere in the internal workings of the tested process by altering some (shared) variable. It might also be that the test could gain privileged knowledge by reading the value of some variable altered by the tested process. While there is no reason
that such tests shouldn't be applied in such a way, it is not the intended form of testing for our example and is therefore excluded.

Experiment configurations are just applications of tests to processes:

\[ E = \{ \text{tst} \in \text{TST}, \text{obj} \in \text{OBJ} \} \]

Success has two components, the experiment must have finished and it must have an acceptable final state:

\[ \text{success}(\langle p, s, P \rangle) \iff \langle p, s \rangle \in \text{term(Source)}, s \in P \]

It remains to define the transition relation between experiments. The set \( P \) takes no part in these transitions, it's just "there for the ride".

\[
\begin{align*}
\langle p, s \rangle & \xrightarrow{E} \langle p', s' \rangle \\
\hline
\langle p, s, P \rangle & \rightarrow \langle p', s', P \rangle
\end{align*}
\]

where \( \xrightarrow{E} \subseteq \text{trans(Source)} \). Thus experiments will proceed with no outside interference by any observer in any way. Finally,

\[ \text{SEXP} = \langle E, \rightarrow, \text{success} \rangle \]

The definition of \( \text{TEXP} \) is analogous to that for \( \text{SEXP} \), thus here we will give the bare definitions with little justification.

**Definition 5.11**

1. \( \text{TOBJ} = \text{config(Target)} \), ranged over by \( \text{tobj} \)
2. \( \text{TTST} = \{(\text{pr}, P): \text{pr} \in \text{Prog2}, P \subseteq \text{States}\} \), ranged over by \( \text{ttst} \)
3. \( \text{ttst}[	ext{tobj}] = \langle \text{pr}_1 || \text{pr}_2, Q, s \rangle, P \) if \( \text{ttst} = (\text{pr}_1, P), \text{tobj} = \langle \text{pr}_2, Q, s \rangle, -\text{pr}_1 || \text{pr}_2 \)
4. \( F = \{ \text{ttst}[	ext{tobj}] : \text{ttst}[	ext{tobj}] \text{ is defined} \} \), ranged over by \( f \)

5. \( \text{tsuccess}((\text{pr}, Q, s), P) \) iff \( (\text{pr}, Q, s) \in \text{term}(\text{Target}), s \in \mathcal{P} \)

Note we will generally write \( \text{tsuccess} \) as 'success' where no confusion can arise.

\[
\begin{align*}
\langle \text{pr}, Q, s \rangle \to \langle \text{pr}', Q', s' \rangle \\
\hline
\langle \text{pr}', Q', s', P \rangle \\
\end{align*}
\]

And so to ...

\[ \text{TEXP} = \langle F, \to, \text{success} \rangle \]

It is now a simple matter to give the extended translation, i.e. a total function from source experiments to target experiments.

**Definition 5.12**

\[ \llbracket (p, s), P \rrbracket_2 = (\llbracket p, s \rrbracket, P) \]

Thus the state part of the test is passed over verbatim and the remainder of the experiment is translated as a configuration of Source, so that the process part of the test is translated into a form compatible with the translation of the process under test.

To emphasise the point once more we can say that we have in fact 'translated' the test as well as the process, and in considering translation correctness we will compare the results of applying test to process with the results of applying the 'translated' test to the translated process.
5.2. Proving correctness of the translation $[.]_2$

This section has two main aims,

1. To demonstrate that all of the translation theories discussed so far (i.e. the bisimulation theory of chapter 2, the 'correctness' theory of Li [Li 83], the testing theory of chapter 4, and the theory proposed in [Astesiano and Zucca 82]) only the testing theory can cope with this example translation.

2. To provide a set of four conditions which are sufficient to guarantee a translation between experiment systems is a complete implementation, given two demands on the nature of the systems. We then establish the truth of these four conditions for the example translation, and finally go on to discuss how one of the two demands could be relaxed by the addition of a fifth condition.

It will be helpful to introduce now the demands on the nature of the experiment systems, taken together they will amount to requiring that the computation set of every experiment (in either system) contains only finite computations, and every computation ending in an unsuccessful configuration contains no successful configurations. That is to say; every computation is finite and the computation is successful if and only if the final (or tip) configuration is successful.

Definition 5.13
An experiment system $\text{EXP} = \langle E, \rightarrow, \text{success} \rangle$ is \textit{computationally finite} if $\forall e \in E. \langle c \rangle^e \in \text{Comp}(e)$ implies $K$ is finite.
Definition 5.14

An experiment system $\text{EXP} = \langle E, \rightarrow, \text{success} \rangle$ is success terminating if $\forall e \in E. \text{success}(e) \implies e \not\rightarrow 0$.

Proposition 5.1

$\text{SEXP}$ and $\text{TEXP}$ are both computationally finite and success-terminating.

5.2.1. Why the bisimulation approach won't work

In the previous section we defined two translations

$[.]_1: \text{config(Source)} \rightarrow \text{config(Target)}$

$[.]_2: \text{config(SEXP)} \rightarrow \text{config(TEXP)}$

We now examine the applicability of each translation theory to $[.]_1$ and $[.]_2$, primarily from the context of a bisimulation approach.

$[.]_1$: Clearly the bisimulation theory and the equivalence theory of Astesiano and Zucca must deny $[.]_1$ as a 'correct' translation because

$R:: \text{Slv} \notin \text{labels(Target)}$

but

$<R:: \text{Sle}, s> \overset{R:: \text{Slv}}{\rightarrow} <\text{done}, s> \text{ if eval}(e,s) = v.$

In effect all the 'observable' actions on the arrow for the Source system have been removed. Any attempt to 'observe' an action in the Target system involves manipulation of a communication state, which is, essentially, how the testing approach works for $[.]_2$. The testing theory of course is not applicable to $[.]_1$ directly since Source and Target are not experiment systems.
Li's correctness theory fails immediately for a similar reason. Part of Li's definition of correctness [Li 83 page 196] was a condition

$$\forall r. \text{tr}(R(r)) = R(\text{tr}(r))$$

where tr is a semantic translation (extended here on the left hand side to cover sets of configurations), r is a source configuration and the 'result set' R(r) is defined for source (target) configurations as ([Li 83 Definition 5.1 page 195]):

$$R(r) = \{ r' : r \xrightarrow{\mathcal{M}} r', r' \in T \}$$

where T is the set of terminal source (target) configurations. Thus A1 essentially says that the possible successfully terminated configurations reachable from the translation of configuration r are exactly the translations of successfully terminated configurations reachable from r.

We can see that A1 does not hold for tr = $\mathcal{M}_1$; for example, take

$$r = \langle R:: \text{Sle}, s \rangle \in \text{config(Source)}$$

then, by Li's definitions we have

$$R(r) = \{ \langle \text{done}, s \rangle \}$$

Now,

$$\text{tr}(r) = \langle R:: \text{off}(_); \text{wait}(_), \text{initQ}, s \rangle$$

and

$$R(\text{tr}(r)) = \emptyset$$

since $\forall Q', s'$

$$\langle R:: \text{wait}(_), Q', s' \rangle \notin \text{term(Target)}$$

What has happened? The problem is caused by Li's heavy emphasis on 'final result' states, to the detriment of notions about observation. Correct translations, essentially, assume
configurations interact with their environment through the labels on their transitions, which for this example translation is not so; communication is affected by altering a shared communication state.

Li then goes on, in the definition of the sets R and K (a similar definition to R but for deadlocked states) to ignore action labels altogether when gathering result configurations and deadlocked configurations. There is no concern for how a configuration may be 'guided', by an observer specifying a particular sequence of action labels, into one set or the other.

\([\mathcal{L}]_2\) - Since experiment systems were constructed specifically for the testing theory, applying any of the other theories to a translation between them is, at least, a partial acceptance of that paradigm. The acceptance is essentially that configurations from a transition system should be observed (tested) in a manner relevant to that system. What might not be accepted is the use of must and may in defining correctness. We will show that in eschewing the must/may approach, the bisimulation (and also the equivalence theory of \([\text{Astesiano and Zucca 82}]\)) runs into serious difficulties.

Li's theory fares rather better on \([\mathcal{L}]_2\) since it always did ignore transition labels and there are none in experiment systems. However, the 'adequacy' conditions are very much concerned with action labels and thus do not present a viable proof technique in this case. As to the exact nature of any result proven about \([\mathcal{L}]_2\), we merely note that, under the testing interpretation of \([\mathcal{L}]_2\), the conditions for correctness are reasonable if rather stringent.
(Note complete-implementation does not imply A1, ..., A3). Thus 'correctness' could supplant 'complete-implementation' in this case, but in general would probably be too restrictive, particularly if the translation was only an 'implementation' or a different notion of test was required.

Before we go into any further details it must be appreciated that it is very difficult to completely discredit the bisimulation approach, and probably any other, since it may give some kind of interesting positive result when applied in an unusual way. That is of course exactly what we are trying to do in applying it to \( [.]_2 \) rather than \( [.]_1 \). We will, however, endeavour to demonstrate why many intuitively reasonable applications of the bisimulation theory must fail, by attempting to prove \( [.]_2 \) is P-good for a 'reasonable' P (and failing). We start by choosing P.

It should be noted that now we are translating between experiment systems the only action label is c so most of the work must be done by P. Probably the weakest 'reasonable' choice is to compare states at the tips of the, in this case, finite computation tree. Firstly we need an auxiliary definition to gather the possible final states of an experiment; let \( \rightarrow \rightarrow^* \),

**Definition 5.15**

Let \( e \in \text{exp}(S\text{EXP}) \), \( f \in \text{exp}(T\text{EXP}) \)

\[
\text{resultset}(e) = \{ s : e \rightarrow (p, s, P) \rightarrow \}
\]

\[
\text{resultset}(f) = \{ s : f \rightarrow (p_r, Q, s, P) \rightarrow \}
\]

Now, we will just demand that any experiments which cannot proceed
and which are paired in the bisimulation must have the same state
component. (The example to follow will have no deadlocked states
so confining comparisons to pairs of terminal states (say) will
not help).

Definition 5.16

\[ ([p, s], P) \text{ result } ([p', Q, s'], P) \]

iff

\[ ([p, s], \not \rightarrow) \land ([p', Q, s'], \not \rightarrow) \implies s = s' \]

The following proposition establishes a crucial property of any
result-bisimulation, namely that any experiments paired in the
bisimulation must have the same set of final states.

Proposition 5.2

1. \( e \rightarrow e' \) implies \( \text{resultset}(e') \subseteq \text{resultset}(e) \)
2. \( f \rightarrow f' \) implies \( \text{resultset}(f') \subseteq \text{resultset}(f) \)
3. Suppose \( e \mathrel{R} f, R \) a result-bisimulation (so \( R \subseteq \exp(SEXP) \times \exp(TEXP) \)),
then \( \text{resultset}(e) = \text{resultset}(f) \).

Proof

1. Suppose \( e \rightarrow e' \) and

\[ e' \rightarrow e'' \not \rightarrow \]

then

\[ e \rightarrow e'' \not \rightarrow \]

so \( \text{resultset}(e') \subseteq \text{resultset}(e) \)

Similarly for \( f \).

2. Suppose

\[ e \rightarrow e' \not \rightarrow \]

then
Now, since $e' \not\rightarrow^*$
$\forall f'' . f' \rightarrow f'' \not\rightarrow^*$ implies $e' R f''$
and, because all computations are finite,
$\exists f'' . f' \rightarrow f'' \not\rightarrow^*$
Thus,
$\exists f'' . f \rightarrow f'' \not\rightarrow^*$, $e' R f''$
so by Definition 5.15, Definition 5.16
resultset($e$) $\subseteq$ resultset($f$)
Similarly, resultset($f$) $\subseteq$ resultset($e$)

In the following example we will demonstrate that $[\cdot]_2$ is not a result-good translation by showing that a translated experiment can reach a configuration which cannot be paired, in a result-bisimulation, with any configuration attainable by the original experiment.

**Example**

The experiment we wish to translate is composed of 5 processes in parallel, two senders (named R) and three receivers (named S). Any complete computation of this experiment will determine a pairing of senders with receivers which will be reflected in the final state.

**Notation** States will be represented as

$$s = <v_1, v_2, v_3>$$ if $eval(x,s) = v_1$, $eval(y,s) = v_2$, $eval(z,s) = v_3$

So, take
The set of 'successful' final states can be $\emptyset$ because we are not going to use it here, similarly the distinction between test and process is ignored. The initial state is chosen to show up the final assignments, since all assignments are non-zero. Now, by consideration of the operational semantics it is easily seen that

$$[e]_2 \rightarrow \begin{cases} \{<R::wait(_, _), R::wait(_, _), _, \} \\
\text{initQ}[<1,4>, <2,5>/(R,S,send)] \\
<0,0,0>, \\
\emptyset \end{cases}$$

Thus, in effect, $f$ has contracted to choose the receiver for the value 4 before choosing the receiver for the value 5. The experiment $e$ cannot express such a contract and so no pairing can be found for $f$ in a result-bisimulation containing $(e,[e]_2)$ as we shall show.

We have

$$[e]_2 \rightarrow f$$

thus if $[.]_2$ is result-good, with result-bisimulation $R$, then

$$\{e'. e \rightarrow e', e' \in R \}$$

(Note we are using $\rightarrow$ rather than $\Rightarrow$ as the transition relation).

The example is simple enough that we can search for $e'$ by simple case analysis

\text{case } e = e': but
\[
\text{e} \rightarrow (\langle R::S14 \mid R::\text{done} \mid S::\text{done} \mid \_\rangle,
\langle 5,0,0\rangle, \\
\phi)
\]

\[= e''\]

i.e. e can choose the receiver for 5 first. Thus

\[\text{resultset}(e'') = \{\langle 5,0,4\rangle, \langle 5,4,0\rangle\}\]

We now show f has no 'corresponding' derivative; suppose

\[f \rightarrow f', e'' \text{ R f'}\]

then either

\[f = f': \text{ but then}\]

\[\langle 4,0,5\rangle \in \text{resultset}(f')\]

and

\[\langle 4,0,5\rangle \in \text{resultset}(e'')\]

which contradicts Proposition 5.2(2).

Or, since the next move of f \textbf{must} be to pick the receiver for 4,

one of the following must hold by Proposition 5.2(1)

1. \[\text{resultset}(f') \subseteq \{\langle 4,0,5\rangle, \langle 4,5,0\rangle\} = \text{set}_1\]
2. \[\text{resultset}(f') \subseteq \{\langle 0,4,5\rangle, \langle 5,4,0\rangle\} = \text{set}_2\]
3. \[\text{resultset}(f') \subseteq \{\langle 0,5,4\rangle, \langle 5,0,4\rangle\} = \text{set}_3\]

depending on which of x (set_1) y (set_2) or z (set_3) was chosen. However,

\[\forall i \in \{1,2,3\}. \text{resultset}(e'') \not\subseteq \text{set}_i\]

thus by Proposition 5.2(2) there can be no f'.

In effect this demonstrates that, in some sense, f has less choice than e (or has made more decisions).

\[\text{case } e' = (\langle R::\text{done} \mid R::S15 \mid S::\text{done} \mid \_\rangle,
\langle 4,0,0\rangle,\]

\[\_\rangle,\]

\[\_\rangle,\]

\[\_\rangle,\]

\[\_\rangle,\]

\[\_\rangle,\]

\[\_\rangle,\]

\[\_\rangle,\]

\[\_\rangle,\]

\[\_\rangle,\]

\[\_\rangle,\]

\[\_\rangle,\]
but then
\[
\text{resultset}(e') = \{<4,0,5>, <4,5,0>\}
\]

whereas
\[
<5,0,4> \in \text{resultset}(f)
\]

so by Proposition 5.2(1,2) neither \(e'\) nor any of its derivatives can be paired with \(f\). Thus in this case \(f\) has more choice than \(e'\), and this is true of the other remaining cases which we will not explore.

We have thus demonstrated that \(f\) cannot be paired in a result-bisimulation, and thus \([f]_2\) cannot be result-good.

5.2.2. The four conditions

The technique we will employ to prove \([f]_2\) is a complete implementation relies on characterising the tips (final configurations) of the computations for translated experiments as elements of a set \([e]\) (for some \(e\) dependent on the particular tip). The set \([e]\) comprises those configurations \(f\) which can be obtained from \([e]\) by putting all possible communication offers in the appropriate send-queue. This enables, from conditions \(c_2\) and \(c_3\) to be defined, the construction of a correspondence between tips of the source computations and tips of the target computations. Since we are not interested in choice points along the way, the downfall of the bisimulation theory, we needn't consider all possible forms the translated experiment might attain; just the forms taken at the tips. This will give condition \(c_1\). Condition \(c_4\) will establish criteria of success for
these tip forms.

At the end of this chapter we will discuss how these conditions may be generalised and extended to apply to experiment systems which are not computationally finite.

Theorem 5.1

Suppose

1. \( S = <E_s, \rightarrow_s, \text{success}_s>, T = <E_t, \rightarrow_t, \text{success}_t> \)

are computationally finite success terminating experiment systems, \( e \) ranges over \( E_s \) and \( f \) ranges over \( E_t \). (Subscripts will be dropped from now on where unnecessary).

2. \([.] : E_s \rightarrow E_t, [.] : E_s \rightarrow \mathcal{P}(E_t)\)

then

\begin{align*}
\text{c1. } & \forall e, e'. e \rightarrow e' \text{ iff } [f] \in [e']. [e] \rightarrow f \\
\text{c2. } & \forall e, f. [e] \rightarrow f \rightarrow \text{ implies } [e']. e \rightarrow f \in [e'] \\
\text{c3. } & \forall e. \forall f \in [e]. e \rightarrow \text{ iff } f \rightarrow \\
\text{c4. } & \forall e. \forall f \in [e]. \text{success}(e) \text{ iff success}(f) \text{ implies } \\
& \forall e. [e] \text{ completely-implements } e
\end{align*}

If we were only interested in implementation then it would suffice to prove,

a) The 'if' part of c1
b) c2
c) The 'only if' part of c3
d) c4

It will turn out that most of the work in the proof of complete implementation is concerned with establishing a) and b)
Proof of Theorem 5.1:

1. ([e] must implies [e] must): Suppose e must and take any computation (which will be finite) of the translation

\[ [e] \rightarrow f \rightarrow > \]

then by c2

\[ \rightarrow e', \ f \epsilon \ [e'] \]

so by c1

\[ e \rightarrow e' \]

and by c3

\[ e' \rightarrow > \]

Thus, since S is success terminating and e must (by hypothesis),

\[ \text{success}(e') \]

and hence by c4

\[ \text{success}(f) \]

2. (e may implies [e] may): Suppose e may, then

\[ \rightarrow e', \ e \rightarrow e', \ \text{success}(e') \]

Thus by c1

\[ \rightarrow f \epsilon \ [e']. \ [e] \rightarrow f \]

and by c4

\[ \text{success}(f) \]

3. ([e] must implies e must): Suppose [e] must; take any computation (finite by supposition 1))

\[ e \rightarrow e' \rightarrow > \]

then by c1

\[ \rightarrow f \epsilon \ [e']. \ [e] \rightarrow f \]

and by c3

\[ f \rightarrow > \]

Thus, since [e] must,
success(f)
and hence by c4
success(e')

4. ([e] may implies e may): Suppose [e] may, then since T is
success terminating

[e] \rightarrow f \rightarrow^{*}, success(f)

By c2
\{ e'. f e [e'] 
so by c1
e \rightarrow e'
and c4 gives
success(e')

We now pause to outline the method by which we will prove
conditions c1 to c4. The proof rests on three crucial definitions

\[ k \] - is the set of target configurations which can be obtained
from [e] by 'running' through one step all and only (and in any
order) those labelled commands in [e] whose first step is to
place a communication offer in a send queue. c3 and c4 follow
immediately from this definition.

cons(f) - is true if the target experiment f has a program and
communication state which are consistent in the sense that f
can be regarded as '[e] doing something' for some e.

inv(f) - if f can be regarded as '[e] doing something' then
inv(f) = e; basically inv is an inverse translation function
defined on consistent f.
The following properties of these definitions will hold:

1. \( \forall e. (\text{cons}(\ll e\rr), \forall f \in \ll e\rr, \text{cons}(f)) \)

2. \( (\text{cons}(f), f \to f') \) implies \( \text{cons}(f') \)

3. \( \text{cons}(f) \) implies \( \text{inv}(f) \) is defined.

4. \( \forall e. (f \in \ll e\rr) \) implies \( \text{inv}(f) = e, \text{inv}(\ll e\rr) = e \)

5. \( (\text{cons}(f), f \to f') \) implies \( \text{inv}(f) \to \text{inv}(f') \)

6. \( (\text{cons}(f), f \not\to f') \) implies \( f \in \ll \text{inv}(f) \rr \)

Using these 6 properties we can prove c2 and the 'if' part of c1 by:

\[ \text{c1 (if): Suppose} \]

\[ f. \ll e\rr \to f \in \ll e'\rr \]

then

\[ \text{cons}(f), \text{cons}(\ll e\rr) \quad (1 \text{ and } 2) \]

so

\[ \text{inv}(\ll e\rr) \to \text{inv}(f) \quad (3 \text{ and } 5) \]

hence

\[ e \to e' \quad (4) \]

\[ \text{c2: Suppose} \]

\[ \ll e\rr \to f \not\to f' \]

then

\[ \text{cons}(f) \quad (1) \]

so

\[ f \in \ll \text{inv}(f) \rr \quad (6 \text{ and } 3) \]

We have already noted that c3 and c4 are immediate from the definition of \( \ll . \rr \) so all that remains is the 'only if' part of c1 which can be straightforwardly proved without recourse to the above definitions.
5.2.3. Proving the four conditions for $[.]_2$

Before we can prove conditions c1 to c4 in this case we must define exactly what $[.]$ refers to. This involves a number of prior definitions which, of course, we give first.

IMAGE is the set of all possible derivatives $pr \in Prog2$ which may occur in a computation of some translated experiment. This will be proved later but should be intuitively acceptable anyway.

Definition 5.17

First define $INTER_t$ the intermediate syntactic forms for labelled commands with identifying token t, ranged over by $inter_t$, by

\[
INTER_t := \begin{cases} 
R::c[(R,t)] & \\
R::\text{wait}((R,S,\text{sent}), <t>); [c]((R,t)) & \\
S::\text{akn}((R,S,\text{sent}), <t>); [c]((S,t)) 
\end{cases}
\]

and define

\[
INTER = \sum_{t \in \text{Nat}} INTER_t
\]

Note carefully the use of subscripts within $inter_t$ (i.e. $inter_t$ 'knows' it is the t'th labelled command from the left).

Now define

\[
IMAGE = \text{ranged over by } u, \quad u ::= \text{inter}_1 \mid \ldots \mid \text{inter}_n, n \geq 1.
\]

For each configuration occurring in a computation of a translated experiment we need to know which labelled commands are doing what.

Definition 5.18

Let $f = \langle u, Q, s \rangle, P \in \text{exp}(TEXP)$
then we will define the multisets (in fact they will contain no repetitions, but it helps the definitions along)

waiting\(f, R, S\) - the identifying tokens of all labelled commands in \(f\) which have offered to send a value from \(R\) to \(S\), but which have not (yet) received an acknowledgment from a receiver.

akning\(f, R, S\) - the identifying tokens of all labelled commands in \(f\) which have just received a value as part of a communication, between some sender \(R\) and receiver \(S\), but have not yet acknowledged the appropriate sender.

assigners\(f\) - the identifying tokens of all labelled commands in \(f\) which are about to execute an assignment statement.

offering\(f, R, S\) - the identifying tokens of all labelled commands in \(f\) which are about to offer a communication between sender \(R\) and receiver \(S\).

by

\[
\begin{align*}
\text{waiting}(f, R, S) &= \{ t : \text{inter}_t = R::\text{wait}(\langle R, S, \text{sent} \rangle, \langle t \rangle); [\cdot](R, t) \} \\
\text{akning}(f, R, S) &= \{ t : \text{inter}_t = S::\text{akn}(\langle R, S, \text{sent} \rangle, \langle t \rangle); [\cdot](S, t) \} \\
\text{assigners}(f) &= \{ t : \text{inter}_t = R::\text{x}=e; [\cdot](R, t) \} \\
\text{offering}(f, R, S) &= \{ t : \text{inter}_t = [R::\text{S}=e; [\cdot](R, t) \}
\end{align*}
\]

Further, we take

\[
\begin{align*}
\text{senders}(f) &= \sum_{R, S \in \text{Proc}} \text{waiting}(f, R, S) \\
\text{receivers}(f) &= \sum_{R, S \in \text{Proc}} \text{akning}(f, R, S) \\
\text{offerers}(f) &= \sum_{R, S \in \text{Proc}} \text{offering}(f, R, S)
\end{align*}
\]
Note all these multisets will in fact have no repetitions (i.e. they will be normal sets) but defining them as multisets will facilitate the definition of 'cons' (to follow) because multisets would show up any repetitions and part of the statement of 'cons' is that there aren't any repetitions.

We now define a relation 'run' to determine the derivatives of an experiment \( f \) given a set of process tokens identifying those labelled commands to be 'advanced one step' (in any order).

**Definition 5.19**

In 'do-one' we formalise the notion of executing a single labelled command determined by some token

Let \( f = (\langle u, Q, s \rangle, P) \)

\( u = \text{inter}_1 \mid \ldots \mid \text{inter}_n \in \text{IMAGE} \)

then define

\[
\text{do-one}(t, f, f') \text{ if } \\
1. 1 \leq t \leq n, \\
2. \langle \text{inter}_t, Q, s \rangle \succ \langle \text{inter}_t, Q, s' \rangle, \\
3. \forall i, 1 \leq i \leq n. \\
\text{inter}_1'' = \begin{cases} 
\text{inter}_i' & \text{if } i = t \\
\text{inter}_i' & \text{otherwise} 
\end{cases} \\
4. u' = \text{inter}_1'' \mid \ldots \mid \text{inter}_n'', \\
5. f' = (\langle u', Q', s' \rangle, P).
\]

To run a set of tokens do them one at a time in any order:

\[
\text{run}(\phi, f, f').
\]

\[
\text{run}(S \cup \{ t \}, f, f'') \text{ if } \\
\text{do-one}(t, f, f'), \\
\text{run}(S, f', f'').
\]
Note $S$ is not a multiset so each process can advance at most one step.

We will of course need the following:

**Proposition 5.3**

Suppose $f = (\langle u, Q, s \rangle, P)$,

$$u = \text{inter}_1 | \cdots | \text{inter}_n \in \text{IMAGE},$$

$$S \subseteq \{1, \ldots, n\}$$

then

1. $\text{run}(S, f, f')$ implies $f \rightarrow f'$
2. $S = \emptyset$, $\text{run}(S, f, f')$ implies $f \rightarrow \rightarrow f'$

It is now quite easy to characterise the tips of computations for translated experiments as 'almost-translated' experiments, i.e. configurations reached from a translated experiment by making all possible offers of communication in some order.

**Definition 5.20**

$$[e] = \{ f : \text{offerers}([e]) = S, \text{run}(S, [e], f) \}$$

Note that although all tips are of this form, not everything of this form is a tip, e.g.

$$[\{\langle R:: x= e, s \rangle, P \}] = \{ \langle R:: x= e, \text{initQ}, s \rangle, P \}$$

whose single element has a non-trivial derivation.

**Proposition 5.4**

1. $\forall e. \forall f \in [e]. [e] \rightarrow f$
2. Suppose $f \in [e]$ for some $e$, $f = (\langle _, Q, _ \rangle, _$) so $Q$ is the
communication state of \( f \), then

1. \( \forall t. \ Q(t) = 0 \)
2. \( \forall R, S. \ Q(R, S, \text{sent}) = \text{empty} \)
3. \( \forall R, S. \)
   \[ [t: \ t e. \ <t,e> \in \text{mset}(Q(R, S, \text{send})] = \text{offering}([e], R, S) \]
4. \( \forall e. \ [e] = \emptyset \)

Now, the crucial element in the proof of conditions c1 to c4 is the recognition of an invariant property in all derivatives of a translated experiment. This property relates program text to communication state, saying essentially that

If there is a sender (in the program text) waiting for acknowledgement then its unique identifying token is in one (and only one) of:

1. The appropriate send queue.
2. The 'pending tray' of a single receiver (and the receiver has a particular syntactic form).
3. The appropriate sent queue.

Given said invariant, to be called 'cons' (for 'consistent', we need 'inv' for 'inverse'), we can construct an inverse translation 'inv' so that (loosely)

\[ f \rightarrow \neg, \ \text{cons}(f) \implies f \in \{\text{inv}(f)\} \]

(which will give condition c2 eventually). So, on with the definitions ...

Definition 5.21
Suppose \( f = (<u, Q, s>, P) \), we start with a few auxiliaries
corresponding to the three categories mentioned above:

\[ \text{in-send}(f, R, S) = \{ t : \langle t, e \rangle \in \text{ms}(Q(R, S, \text{send})) \} \]

\[ \text{in-pending}(f, R, S) = \{ t' : \forall t. t \in \text{asking}(f, R, S), Q(t) = t' \} \]

\[ \text{in-sent}(f, R, S) = \{ t : \langle t \rangle \in \text{ms}(Q(R, S, \text{sent})) \} \]

Now just one little definition to help keep things 'clean' (i.e. functional) later on in the proof:

\[ \text{clean}(f) \iff \forall t. (Q(t) = 0 \iff t \notin \text{receivers}(f)) \]

This means that in a clean configuration \(Q(t) = 0\) only when the process identified by \(t\) is about to acknowledge a communication.

Thus we can now define

\[ \text{cons}(f) \iff \]

1. \(f = (\langle u, Q, S \rangle, P), u \in \text{IMAGE}\)

2. \(\forall R, S.
\]

\[ \text{waiting}(f, R, S) = \text{in-send}(f, R, S) \]
\[ +\]
\[ \text{in-pending}(f, R, S) \]
\[ +\]
\[ \text{in-sent}(f, R, S) \]

3. \text{clean}(f)

\[ \)

At this point we can justify our use of multisets because it is simple to show there can be no repetitions in \(\text{waiting}(f, R, S)\) so the three multisets on the right-hand side of the equation in 2 are disjoint sets (i.e. no repetitions) if 2 is true. This is expressed in the following Proposition,

\[ \text{Proposition 5.5} \]

Suppose \(f = (\langle u, Q, S \rangle, P), u \in \text{IMAGE}, \) then

1. \(\forall R, S. \text{norep}(\text{waiting}(f, R, S))\)

2. If \(\text{cons}(f)\) then \(\forall R, S. \)
norep(in-send(f, R, S))
norep(in-pending(f, R, S))
norep(in-sent(f, R, S))

and these sets are pairwise disjoint.

Two relevant points are noted in the following Proposition,

Proposition 5.6
1. \( \forall e. \text{cons}([e]) \)
2. \( \forall e. \forall f \in [e]. \text{cons}(f) \)

For each consistent experiment \( f \) we can now define 'inv(f)' as a
source experiment \( e \) such that \( f \) becomes a member of \([e]\) when all
communications received but not fully acknowledged are run to
completion by both participants. No new actions are started. We
will explain a little more fully in the following definition,

Definition 5.22

Suppose \( \text{cons}(f), f = \langle u, Q, s \rangle, P \), then

\[
\text{inv}(f) = \langle p, s \rangle, P
\]

if \( u = \text{inter}_1 | \cdots | \text{inter}_n \)

\[
p = R_1::c_1 | \cdots | R_n::c_n
\]

\[
\text{match}(\text{inter}_1, Q) = R_1::c_1, i = 1, \ldots, n
\]

where

\[
\text{match}([R::c](R, t), Q) = R::c
\]

\[
\text{match}(R::\text{wait}((R, S, \text{sent}), \langle t \rangle); [e](R, t), Q) =
\]

\[
\begin{cases}
R::c & \text{if } t \in \text{in-pending}(f, R, S) + \text{in-sent}(f, R, S) \\
R::S!e;c & \text{if } \langle t, e \rangle \in \text{mset}(Q(R, S, \text{send}))
\end{cases}
\]
Note that if only the value of e was stored then match would be very hard to define. That's why the expression e was stored rather than the value.

\[ \text{match}(S::\text{akn}((R,S,\text{sent}),\langle t \rangle);[c]\langle S,t \rangle, Q) = S::c \]

The reasoning is as follows,

1. \(\text{inv}(f)\) has the same state as \(f\), any communications which have been received have already had their associated value bound in the state.

2. The progress of any particular labelled command \(i\) (or rather the progress of the matching source command) can be found by reference to the communication state. (It should be noted at this point that it is necessary to prove match is indeed a function, this result follows from the consistency of \(f\) and will not be pursued further here). Match is defined by case analysis of the definition of IMAGE:

The match of a translated command is of course the command itself.

The match of a sender waiting for acknowledgement depends on whether or not its offer has been received, if so its wait is ended, if not then the expression from the queue is used to reconstruct the sending command whose action placed it on the queue and the communication is deemed not to have taken place.

The match of a receiver about to acknowledge assumes the receive action to be finished
The following Proposition establishes some important points about \( \text{inv} \).

**Proposition 5.7**

1. \( \text{inv}(f) \) is well-defined if \( \text{cons}(f) \).
2. \( \forall e . \text{inv}([e]) = e \)
3. \( \forall e . \forall f \in [e] . \text{inv}(f) = e \)

The heart of the proof of conditions c1 to c4 lies in the following Proposition and two Lemmas. Proposition 5.8 will establish condition c2 while Lemma 5.1 and Lemma 5.2 determine the forward and reverse (respectively) implications of condition c1. Conditions c3 and c4 are readily established directly from the appropriate definitions.

**Proposition 5.8**

If \( \text{cons}(f) \), \( f = (u,Q,s), P \), \( f \not\rightarrow \), then

1. \( \forall R, S . \text{waiting}(f,R,S) \not\rightarrow \text{in-send}(f,R,S) \)
2. \( \exists e . f \in [e] \)

**Proof**

1. Suppose there exist \( R \) and \( S \) such that the consequent is not true. Then, since \( \text{cons}(f) \), we must have

\[
\text{in-pending}(f,R,S) \not\rightarrow \text{in-sent}(f,R,S) = \phi
\]

**case** \( \text{in-pending}(f,R,S) = \phi \): then

\[
\exists t . t \in \text{akning}(f,R,S)
\]

hence, by the operational semantics,

\[
\exists f' . \text{run}([t],f,f')
\]
i.e. \( f \rightarrow \rightarrow f' \) (by Proposition 5.3 (2)).

**Case** \( \text{in-pending}(f, R, S) = \phi, \text{in-sent}(f, R, S) = \phi \): but we must have

\( \text{in-sent}(f, R, S) \subseteq \text{waiting}(f, R, S) \)

hence

\[ j \omega, t. Q(R, S, \text{sent}) = \langle t \rangle. q, t \in \text{waiting}(f, R, S) \]

thus, by the operational semantics,

\[ j f'. \text{run}([t], f, f') \]

i.e. \( f \rightarrow \rightarrow f' \) (by Proposition 5.3 (2)).

2. First we need to establish a few points about \( f \); suppose

\[ u = \text{inter}_1 | ... | \text{inter}_n \]

then

1. \( \forall R, S. \text{waiting}(f, R, S) = \text{in-send}(f, R, S), \text{in-knowing}(f, R, S) = \text{in-sent}(f, R, S) \)

This was established in part 1, except to note that

\( \text{in-pending}(f, R, S) = \phi \) implies \( \text{in-knowing}(f, R, S) = \phi \)

2. \( \text{assigners}(f) = \phi \), else by the operational semantics \( f \rightarrow \rightarrow \).

Note if eval were not a total function then this might not hold since \( \text{eval}(e, s) = v \) might not be provable, thus robbing the assignment semantic rule (section 1.3) of its antecedent. However, the result would still hold.

3. \( \forall t \in \text{Nat.} Q(t) = 0 \), this follows immediately since \( \text{clean}(f) \).

Thus, every \( \text{inter}_i \) is in one of the forms

- a) \( R:: \text{wait}(_) ; [c]_{(R, t)} \)
- b) \( [S::R?x ; c]_{(S, t)} \)
- c) done

Now, take \( e = \text{inv}(f) \)
In the following arguments we will need to single out from a program (in either Source or Target) the particular labelled command(s) involved in a transition, of which there are at most two. We do this by defining a set of injective functions:

Definition 5.23

A (binary) (Source) program context

\[ pc: \text{Prog} \times \text{Prog} \rightarrow \text{Prog} \]

is a (partial) function defined for some \( n \geq 2 \),

\[ i, j \in \{1, \ldots, n\}, i \neq j, \]

\[ \{R_k::c_k : k \in \{1, \ldots, n\}-\{i, j\}\} \]

by

\[ pc(p_1, p_2) = R_1::c_1 | \ldots | R_n::c_n = p \]

if \( p_1 = R_i::c_i \),

\( p_2 = R_j::c_j \),

\( \vdash p \)

Thus a context is essentially a program with two holes for (compatible) labelled commands. A similar definition also serves for unary (Source) program contexts, also ranged over by \( pc \).

For binary Target contexts, ranged over by \( uc \), we must take into account the identifying token of an argument labelled command;

\[ uc: \text{INTER}_t \times \text{INTER}_u \rightarrow \text{IMAGE} \]

is a partial function defined, for some \( n \geq 2 \),

\[ t, u \in \{1, \ldots, n\}, t \neq u, \]

\[ \{\text{inter}_k : k \in \{1, \ldots, n\}-\{t, u\}\} \]
by \( uc(\text{inter}_t, \text{inter}_u) = r \)

if \( r = \text{inter}_1 \parallel \ldots \parallel \text{inter}_n \),

\( \vdash r \)

Again, unary contexts are defined similarly and also ranged over by \( uc \).

\( \Box \)

Lemma 5.1

\( \forall e. ( e \rightarrow e' \text{ implies } \llbracket e \rrbracket \rightarrow \rightarrow \llbracket e' \rrbracket ) \)

Proof

The proof proceeds by cases on the derivation of \( e \rightarrow e' \); we will just demonstrate the difficult case wherein

\( e = (\text{pc}[R::Sle; c_1, S::R?x;c_2], s>, P), \)

\( e' = (\text{pc}[R::c_1, S::c_2], s[v/x]>), P) \)

where \( \text{eval}(e,s) = v \). Then \( t_1, t_2 \).

\( \llbracket e \rrbracket = \langle uc[ R::Sle; c_1](R,t_1), S::R?x;c_2](S,t_2), \)

\( \text{initQ,} \)

\( s>, P \)

where \( uc: \text{INTER}_{t_1} \times \text{INTER}_{t_2} \rightarrow \text{IMAGE} \)

is the binary program context such that \( \forall p_1, p_2 \).

\( uc[ \llbracket p_1 \rrbracket (R,t_1), \llbracket p_2 \rrbracket (S,t_2) ] = \llbracket \text{pc}[p_1,p_2] \rrbracket \)

whenever both sides are defined. Now, we proceed to 'run' the translation by asserting certain instances of the relation 'run'

Step 1

\( \text{run}(\llbracket t_1 \rrbracket, \llbracket e \rrbracket, f_1) \)

where
\[ r_1 = (\langle \text{uc}[R::\text{wait}(\langle R,S,\text{sent}\rangle,\langle t_1\rangle); [c](R,t_1)\rangle, \]
\[ [S::R?x; c_2](S,t_2)], \]
\[ \text{initQ}[(t_1,e)/(R,S,\text{send})], \]
\[ s>, P) \]

Step 2
\[ \text{run}([t_2], r_1, r_2) \]
where
\[ r_2 = (\langle \text{uc}[_, \]
\[ S::\text{kn}(\langle R,S,\text{sent}\rangle,\langle t_2\rangle); [c_2](S,t_2)], \]
\[ \text{initQ}[t_1/t_2], \]
\[ s[v/x]>, P) \]
and eval(e,s) = v.

Step 3
\[ \text{run}([t_2], r_2, r_3) \]
where
\[ r_3 = (\langle \text{uc}[_, \]
\[ S::[c_2](S,t_2)], \]
\[ \text{initQ}[(t_1)/(R,S,\text{sent})], \]
\[ s[v/x]>, P) \]

Step 4
\[ \text{run}([t_1], r_3, r_4) \]
where
\[ f_4 = (\langle u_c R: [c_1]_R t_1 \rangle, \]
\[ S: [c_2]_{S, t_2}), \]
\[ \text{initQ}, \]
\[ s[\nu x], P) \]

By the construction of \( u_c \),
\[ f_4 = [e'], \]
hence
\[ [e] \rightarrow f_1 \rightarrow f_2 \rightarrow f_3 \rightarrow f_4 = [e'] \]
\[ \Box \]

**Corollary 5.1**

\( \forall e. \) if \( e \rightarrow e' \) then \( [e] \rightarrow [e'] \)
\[ \Box \]

**Lemma 5.2**

\( \forall f. \) if \( \text{cons}(f), f \rightarrow f' \) then

1. \( \text{cons}(f') \)
2. \( \text{inv}(f) \rightarrow \text{inv}(f') \)

**Proof**

We proceed by cases on the derivation of \( f \rightarrow f' \) (i.e. by cases on \text{IMAGE}). In proving 1. we will concentrate on the second part of \text{cons}, to the detriment of the first and third, mainly by demonstrating how changes to the configuration 'balance out'.

The proof of 2. will involve proving either \( \text{inv}(f) = \text{inv}(f') \) (cases 2, 4 and 5) or \( \text{inv}(f) \rightarrow \text{inv}(f') \) (cases 1 and 3) wherein 'things actually get done' by changing the state and, in case 3, making a firm commitment to a particular communication.
case 1:

\[ f = (u, Q, s), P \]
\[ f' = (u', Q, s[\nu/x], P) \]
\[ u = uc[ R::x=e; c](R, t) \]
\[ inter_1 || ... || inter_n \]
\[ u' = uc[ R::c](R, t) \]
\[ inter'_1 || ... || inter'_n \]
\[ eval(e, s) = v \]

1. Clearly \( con(f') \) since none of the related entities changed.

2. Firstly,

\[ 1 = t \implies match(inter_1, Q) = match(inter'_1, Q) \]

and thus since

\[ match(R::x=e; c)(R, t), Q) = R::x=e; c, \]
\[ match(R::c)(R, t), Q) = R::c \]

we have by Definition 5.22 and the semantics of Source

\[ inv(f) \Rightarrow inv(f') \]

case 2:

\[ f = (u, Q, s), P \]
\[ f' = (u', Q', s), P \]
\[ u = uc[ R::Sle; c](R, t) \]
\[ inter_1 || ... || inter_n \]
\[ u' = uc[ R::wait((R, S, sent), <t>); c](R, t) \]
\[ inter'_1 || ... || inter'_n \]
\[ Q' = Q[<t, e>, Q(R, S, send)/(R, S, send)] \]

1. The only changes interesting to \( con \) are:

\[ waiting(f', T, U) = \begin{cases} 
\text{waiting}(f, R, S)[+ [t]] & \text{if } (T, U) = (R, S) \\
\text{waiting}(f, T, U) & \text{otherwise}
\end{cases} \]
\[
\text{sending}(f', T, U) = \begin{cases} 
\text{sending}(f, R, S) + [t] & \text{if } (T, U) = (R, S) \\
\text{sending}(f, T, U) & \text{otherwise}
\end{cases}
\]

So the invariant is preserved because both waiting and sending, with respect to \((R, S)\), is augmented by \(t\).

2. Firstly, from the above identities and the definition of match we get
\[
\text{match}(\text{inter}_i, Q) = \text{match}(\text{inter}'_i, Q')
\]

and since
\[
\langle t, e \rangle \in \text{in-send}(f', Q)
\]

we get
\[
\text{match}(R::\text{wait}((R, S, \text{sent}), \langle t \rangle); \llbracket c \rrbracket_{(R, t'), Q'}) = R::S!e; c
\]

Thus
\[
\text{inv}(f) = \text{inv}(f').
\]

\textbf{case 3:}
\[
f = \langle \langle u, Q[\langle t, e \rangle.q/(R, S, \text{send})], s \rangle, P \rangle
\]
\[
f' = \langle \langle u', Q[q/(R, S, \text{send}), t/t'], s[v/x] \rangle, P \rangle
\]
\[
u = \text{uc}[ \llbracket S::R?x; c \rrbracket_{(R, t')} ]
\]
\[
= \text{inter}_1 || \ldots || \text{inter}_n
\]
\[
u' = \text{uc}[ S::\text{ack}((R, S, \text{sent}), \langle t' \rangle); \llbracket c \rrbracket_{(R, t')} ]
\]
\[
= \text{inter}_1 || \ldots || \text{inter}_n
\]

1. The relevant changes are:
\[
\text{in-send}(f', T, U) = \begin{cases} 
\text{in-send}(f, R, S) - [t] & \text{if } (T, U) = (R, S) \\
\text{in-send}(f, T, U) & \text{otherwise}
\end{cases}
\]
\[
\text{in-pending}(f', T, U) = \begin{cases} 
\text{in-pending}(f, R, S) + [t] & \text{if } (T, U) = (R, S) \\
\text{in-pending}(f, T, U) & \text{otherwise}
\end{cases}
\]
It is easily deduced that
\[ \text{clean}(f') \]

since
\[ \text{clean}(f), \ t \in \text{receivers}(f') \]

2. We need to show

a) \( \forall i \in \{t,t'\}. \ \text{match}(\text{inter}_i, Q) = \text{match}(\text{inter}'_i, Q') \)

b) \( \text{match}(\text{inter}_t, Q) = R::\text{Sel}; c' \)

c) \( \text{match}(\text{inter}_t, Q) = S::R?x;c \)

d) \( \text{match}(\text{inter}'_t, Q') = R::c' \)

e) \( \text{match}(\text{inter}'_t, Q') = S::c \)

so here goes,

a) First, by the operational semantics,
\[ \forall i \in \{t,t'\}. \ \text{inter}_i = \text{inter}'_i \]

We proceed by cases on \( \text{inter}_i \),

\textbf{case} \( \text{inter}_i = [T::c](t'',i) \):

Trivial since the result is independent of \( Q \), ie
\[ \text{match}(\text{inter}_i, Q) = \text{match}(\text{inter}'_i, Q') = T::c \]

\textbf{case} \( \text{inter}_i = T::\text{wait}((T,U,\text{sent}), <i>); [c](t'',i) \):

Then from the description of changes in 1. and 1 = t we must have
\[ \text{in-sent}(f,T,U) = \text{in-sent}(f',T,U), \]
\[ i \in \text{in-pending}(f,T,U) \iff i \in \text{in-pending}(f',T,U), \]
\[ \text{in-send}(f,T,U) = \text{in-send}(f',T,U). \]

Hence result by the definition of match in this case.

\textbf{case} \( \text{inter}_i = U::\text{akn}((T,U,\text{sent}), <i>); [c](t'',i) \):

Trivial, again, since the result is independent of \( Q \).

b) Well,
\[ t \in \text{in-send}(f,R, S) \]
so by cons
\[ t \in \text{waiting}(f, R, S) \]
hence
\[ \{ c'. \inter_t = R::\text{wait}((R, S, \text{sent}), <t>); [c']_{(R, t)} \] 
thus
\[ \{ e. \text{match}(\inter_t, Q) = R::\text{Sle}; c' \]
c) Immediate from the definition of match.
d) Reasoning as in b) we note that now
\[ t \in \text{in-pending}(f', R, S) \]
e) Immediate from the definition of match.

Now clearly
\[ \text{inv}(f) \rightarrow \text{inv}(f') \]
by the communication from R to S.

case 4:
\[ f' = (\langle u', Q[Q/(R, S, \text{sent}), t/t'], s \rangle, P) \]
\[ f' = (\langle u', Q[Q.Q/(R, S, \text{sent}), 0/t'], s \rangle, P) \]
\[ u = uc[ S::\text{akn}((R, S, \text{sent}), <t>); [c]_{(S, t')} \]
\[ \inter_1 \mid \ldots \mid \inter_n \]
\[ u' = uc[ S::[c]_{(S, t')} \]
\[ \inter_1 \mid \ldots \mid \inter_n' \]

1. As usual we present the relevant changes

\[
\text{in-sent}(f', T, U) = \begin{cases} 
\text{in-sent}(f, R, S) + [t] & \text{if } (T, U) = (R, S) \\
\text{in-sent}(f, T, U) & \text{otherwise}
\end{cases}
\]

\[
\text{in-pending}(f', T, U) = \begin{cases} 
\text{in-pending}(f, R, S) - [t] & \text{if } (T, U) = (R, S) \\
\text{in-pending}(f, T, U) & \text{otherwise}
\end{cases}
\]
and of course $\text{clean}(f')$ holds.

2. We need to prove

$$\forall i \in \{1, \ldots, n\}. \text{match}(\text{inter}_i, Q) = \text{match}(\text{inter}'_i, Q')$$

If $i = t'$ the result is immediate by the definition of match, and if $\text{inter}_i$ is not a waiting process then the result is similarly obtained. So the only case left is

$$\text{inter}_i = T::\text{wait}((T, U, \text{sent}), t''); [c''_{T, t''}],$$

$$t'' = t' \quad \text{(but it's possible } t'' = t)$$

However, from the discussion in 1. we have

$$\text{in-pending}(f, T, U) + \text{in-sent}(f, T, U) = \text{in-pending}(f', T, U) + \text{in-sent}(f', T, U),$$

$$\text{in-send}(f, T, U) = \text{in-send}(f', T, U)$$

thus, by the definition of match,

$$\text{match}(\text{inter}_i, Q) = \text{match}(\text{inter}'_i, Q')$$

since

$$\text{inter}'_i = \text{inter}_i$$

Hence

$$\text{inv}(f) = \text{inv}(f')$$

**case 5:**

$$f = (u, Q[t].q/(R, S, \text{sent})), s, p)$$

$$f' = (u', Q[q/(R, S, \text{sent})], s, p)$$

$$u = uc[ R::\text{wait}((R, S, \text{sent}), t); [c](R, t) ]$$

$$u' = uc[ R::[c](R, t) ]$$

This case presents no further difficulties, suffice it to say

$$\text{inv}(f) = \text{inv}(f').$$

$\Box$
Corollary 5.2

∀ f. if cons(f), f → f', then

1. cons(f')
2. inv(f) → inv(f')

We now turn to the proof of conditions c1 to c4 given our accumulated armoury of Propositions, Lemmas and Corollaries.

Theorem 5.2

[[.]] : SEXP → TEXP is a complete implementation.

Proof

By Proposition 5.1, SEXP and TEXP are computationally finite and success terminating so we prove conditions c1 to c4 in the statement of Theorem 5.1

c1. Suppose e → e', then by Corollary 5.1

[[e]] → [[e']]

and by Proposition 5.4

| f, [[e']] → f e [[e']]

Now suppose | f, [[e]] → f e [[e]], then by Proposition 5.6 and Corollary 5.2

inv([[e]]) → inv(f)

but by Proposition 5.7

inv([[e]]) = e, inv(f) = e'

thus

e → e'

c2. By Proposition 5.6, Corollary 5.2

cons(f)

and thus from Proposition 5.8
c3. Follows by simple case analysis on the proof of the derivation.
c4. Immediate from the definitions.

5.2.4. Alternative results

In this section we have shown that any translation between computationally finite success terminating experiment systems which satisfies conditions c1 to c4 is a complete implementation (proved in Theorem 5.1). Though an interesting result (we hope!) there is no reason, beyond its use in the current example, why it should stand out from many similar results as 'the one to be proved'. We would like to suggest that what is important here is the use of an 'abstract layer', underneath the testing definition of translation correctness, which emphasises properties of experiment systems and their computations. This allows a flexible 'mix and match' approach to verifying translations. For instance we can prove a translation is an implementation (only) with a similar set of conditions to c1 to c4, by employing the following Definition and Theorem;

**Definition 5.24**

Suppose \( <E, \rightarrow, \text{success}> \) is an experiment system, \( e \in E \), then define

\[
\text{div}(e) \iff e \text{ has an infinite computation, i.e.}
\]

\[
\text{div}(e) \iff \exists e' \in [e'] \text{ such that } e = e_0 \rightarrow e_1 \rightarrow \ldots
\]

**Theorem 5.3**

Suppose
1. $S = \langle E_S \to S, \text{success}_S \rangle$, $e$ ranges over $E_S$.

$T = \langle E_T \to T, \text{success}_T \rangle$, $f$ ranges over $E_T$

are success terminating experiment systems, (subscripts will be dropped from now on where unnecessary).

2. $[.] : E_S \to E_T$, $[.] : E_S \to P(E_T)$

then

C1. $\forall e, e'. (\exists f \in [e'). [e] \Rightarrow f) \Rightarrow e \Rightarrow e'$

C2. $\forall e, f. [e] \Rightarrow f \Rightarrow e'$ implies $\exists e'. f \in [e']$

C3. $\forall e. \forall f \in [e]. e \Rightarrow f$ implies $f \Rightarrow e$

C4. $\forall e. \forall f \in [e]. \text{success}(e) \iff \text{success}(f)$

C5. $\forall e. \text{div}([e]) \Rightarrow \text{div}(e)$

implies

$\forall e. [e]$ implements $e$.

Proof

Let $e \in E$

(e must implies $[e]$ must): Suppose $e$ must; take any computation of $[e]$, then it must be finite since by C5

$\text{div}([e]) \iff \text{div}(e)$

which contradicts $e$ must since $E_S$ is success terminating. Thus we can proceed, as before, on the finite case,

$[e] \Rightarrow f \Rightarrow e'$

then by C2

$\exists e'. f \in [e']$
so by C1

\[ e \rightarrow e' \]

and by C3

\[ e' \rightarrow \]

Thus since \( E_s \) is success terminating and \( e \) must (by hypothesis) we have

\[ \text{success}(e') \]

and hence by C4

\[ \text{success}(f) \]

(\( [e] \) may implies \( e \) may): Then since \( E_t \) is success terminating we have

\[ f, [e] \rightarrow f \rightarrow, \text{success}(f) \]

Now, by C2

\[ f, e'. f \in [e] \]

so by C1 and C4

\[ e \rightarrow e' \]

\( \square \)

Of course, another similar result would be that for computationally finite success sustaining systems conditions c1 to c4 lacking the forward implication of condition c1 are sufficient to guarantee an implementation.

Condition C5 is the only one we haven't already proved for our example translation, simply because there are no infinite computations in SEXP or TEXP. If we were to add some kind of iterative or recursive construct to the Source language then we might like to establish this condition for our new translation. One way of doing this is through the following Proposition (given
Proposition 5.9

Let \( f, f' \) range over \( \text{config(TEXP)} \) and suppose for all \( f \) such that \( \text{cons}(f) \).

\[
\exists k \geq 1. f \rightarrow_k f' \text{ implies } \text{inv}(f) \rightarrow \text{inv}(f')
\]

then

\[
\forall e. \text{div}[\{e\}] \text{ implies } \text{div}(e)
\]

Proof Simply noting \( \text{cons}(\{e\}) \) we see that there must be an infinite computation

\[
\{e\} \rightarrow_k f_1 \rightarrow_k f_2 \rightarrow_k \ldots
\]

so there must be an infinite computation

\[
e \rightarrow \Rightarrow \text{inv}(f_1) \rightarrow \Rightarrow \text{inv}(f_2) \rightarrow \Rightarrow \ldots
\]

In this Proposition the number \( k \) sets an upper bound on how many 'housekeeping' steps the translation can make before doing some of the real work performed by the original. In our proof the real work was done when the state was changed by assigning some value to a variable during either an assignment statement or a receipt \( \text{rec}(qvd,<t',x>) \) statement. Whenever some experiment \( f \) performed one of these statements in a derivation \( f \rightarrow f' \) we showed \( \text{inv}(f) \rightarrow \text{inv}(f') \), whereas for the other statements \( \text{inv}(f) = \text{inv}(f') \) so only (necessary!) 'housekeeping' work has been performed. For the example translation it is easily seen that if

\[
f = (\langle r_1 || \ldots || r_n, s \rangle, p)
\]

then taking \( k = 3n \) establishes the antecedent of Proposition 5.9.

A more general version of this approach would be to replace C5 by the following condition:
C5.1 ∀e. ∃k ≥ 1. ∀f. 

(}\ e \rightarrow_k f \text{ implies } \exists e'. e \rightarrow e', [e'] \rightarrow f)\)

i.e. if a translated configuration takes \(k\) steps to reach \(f\) then it must have simulated at least one source action. Then we endeavour to prove C5 by:

Proof of C5:

Suppose \(\text{div}(\mathcal{[e]}), \text{then}\)

\(\exists f, \mathcal{[e]} \rightarrow_k f, \text{div}(f)\)

for that \(k\) provided by C5.1. Hence

\(\exists e_1. e \rightarrow e_1, [e_1] \rightarrow f\)

so \(\text{div}([e_1])\)

and the construction can be repeated from \(e_1\) to produce \(e_n\) for any \(n\), giving an infinite computation

\(e \rightarrow e_1 \rightarrow e_2 \rightarrow ...\)

\(\Box\)

To establish C5.1 for our (extended) translation would probably be quite simple using (an extended) Lemma 5.1. However, finding a general condition to establish C5.2 stated in terms akin to C5.1 may be problematic since Lemma 5.1 says something about the configurations which a translation may take up during a computation rather than just the end points characterised by \([\cdot]\) and \([\cdot]\). For our example (and hopefully the extended version) it is the case that configurations which are also translations of source configurations can arise almost as required in a computation but this is not to be expected in general.

A proof of complete implementation for the infinite case would require adding the converse (C1.1) of C1 and the converse (C5.2)
of $C_5$ to the conditions $C_1, \ldots, C_5$:

$C_{1.1} \forall e, e'. \ e \Rightarrow e' \text{ implies } \exists f \in \llbracket e' \rrbracket. \ \llbracket e \rrbracket \rightarrow f$

$C_{5.2} \forall e. \ \text{div}(e) \text{ implies } \text{div}(\llbracket e \rrbracket)$

then we can prove $\forall e. \ \llbracket e \rrbracket \text{ must implies } e \text{ must}$ as follows:

Proof of must:

Suppose $\llbracket e \rrbracket \text{ must}$; take any complete computation (finite or infinite) of $e$

case $e \Rightarrow e' \not\Rightarrow$: then by $C_{1.1}$

$\exists f \in \llbracket e' \rrbracket. \ \llbracket e \rrbracket \rightarrow f$

and by $C_2$

$f \not\Rightarrow$

Thus since $\llbracket e \rrbracket \text{ must}$

success($f$)

and then by $C_4$

success($e'$)

case div($e$): then by $C_{5.2}$

div($\llbracket e \rrbracket$)

which is not possible since $\llbracket e \rrbracket \text{ must}$ in a success terminating system.

$\Box$

To prove $\forall e. \ e \text{ may implies } \llbracket e \rrbracket \text{ may}$:

Proof of may:

Suppose $e \text{ may}$; then we have a finite computation

$e \Rightarrow e', \ \text{success}(e')$

hence by $C_{1.1}$

$\exists f \in \llbracket e' \rrbracket. \ \llbracket e \rrbracket \rightarrow f$

and by $C_4$

success($f$)
5.2.5. Extending the translation

Having translated a very small core of CSP to a specially devised language the question of possible extensions to the translation becomes intertwined with the question of possible extensions to the Target language. Thus we should concentrate mainly on the translation method in discussing extensions of the translation and the proof of correctness. Perhaps the most important missing command forms are the conditional (if go fi), repetitive (do go od) and (nested) parallel (c1||c2) forms.

if go fi: It would seem the main problem here lies in translating guarded commands like

\[ b_1 \rightarrow R_1!e_1 \circ b_2 \rightarrow R_2!e_2 \circ b_3 \rightarrow R_3!e_3 \]

wherein the first action of the translation of \( R_1!e_1 \) will be to put an offer in a send queue. For the above guarded command three offers should be put in queues, but at most one should be accepted. Thus there are two related subproblems

1. Making a communication offer should not select a guarded command for execution.

2. Only one communication offer should be accepted.

do go od: Aside from the problems of translating the guarded command there is the difficulty of the infinite computations introduced into the experiment systems by this iterative command. This point was discussed earlier, where it was shown how the proof technique might be extended to handle such cases.
It seems that this construct would pose no difficulties for this translation method in general, however it really all depends on the chosen target language.
6. Weavesort; an example implementation

This chapter has three main aims:

1. To introduce a CCS operator $ as an aid to specification, allowing freedom of choice to the implementer and ease of description to the specifier.

2. To provide an example wherein the relationship of complete implementation, or of having a bisimulation, does not hold in a natural way between the described process and its specification.

3. To show how object implementation may be proved using a technique similar to that employed in the bisimulation theory.

6.1. Extending EXP(CCS) to NEXP(CCS)

Since our aim in this chapter is to establish the relation of object implementation (see Definition 4.11 chapter 4 section 2.5) between a sorting machine and its specification we need to define an experiment system in which the former implements the latter. The system we choose is very similar to EXP(CCS) (see chapter 4 section 1) so experiments are CCS terms, the transition relation is $\rightarrow$, and success is the ability to perform the distinguished action $w$. However, we need to extend EXP(CCS) (to NEXP(CCS)) in two ways:

1. Adding indeterminacy; we will need to express in a single CCS term a number of possible processes, so that whenever a term is encountered an arbitrary choice can be made between the
possibilities. This will greatly aid both the description and specification of the sorting machine.

2. Extending the set of experiments: an experiment in \( \text{EXP}(\text{CCS}) \) is of the form 'test|object', so given an experiment it is always clear which part is the test and which part is the object. Part of our proof technique will involve blurring the distinction between test and object, and to this end we allow an experiment to be (loosely) a term one of whose subterms is the object. The remainder is the test.

Unfortunately we will find it necessary to restrict \( \text{Val} \), the set of communicable values, to be a finite set. This requirement comes up at two distinct places and is pointed out as it arises.

6.1.1. Adding indeterminacy

To aid in the specification and implementation of behaviours in CCS, and thereby \( \text{NEXP}(\text{CCS}) \), we introduce the notion of indeterminacy via a "don't care" or "don't know" operator \( \Theta \) which we add to the CCS of chapter 1. In the interests of brevity we will not completely describe the new CCS, but rather concentrate on the changes.

We expand the possible CCS terms by adding a new operator \( \Theta \) to those already present; if \( p, q \) are terms then \( p \Theta q \) is a term. This term \( p \Theta q \) represents either the process represented by the term \( p \) or the process represented by the term \( q \). The difference between \( p \Theta q \) and \( p \Theta q \) is rather subtle. However, the former represents a process which is altogether different from both the processes
represented by p and q, whereas the latter represents precisely one or the other; it is just that we don't know (or care) which.

The rules for inferring transition relationships are very similar to those given in chapter 1 but the new operator $ must be accommodated. For example we would like the relations $\rightarrow^\dagger$ to be such that

$$A!.(B! + C!) \rightarrow^\dagger p \iff p = B! + C!$$

but

$$A!.(B! \& C!) \rightarrow^\dagger p \iff p = B! \text{ or } p = C!$$

Thus whenever a term p@q comes to be executed we want to replace it with either p or q (we don't care which) and execute that. We achieve this by introducing the notion of a 'deterministic state' of a term.

Definition 6.1

Suppose we are employing a definition set D such that

$$P(x_1, \ldots, x_n) \leftarrow q$$

For any term p let ds(p) be the set of terms defined by induction on p as follows:

$$ds(\text{Nil}) = \text{Nil}$$

$$ds(a.p) = \{a.p\}$$

$$ds(p_1 \cdot p_2) = \{p_1 \cdot p_2' : p_1' \in ds(p_1), p_2' \in ds(p_2)\}$$

$$ds(p_1 \oplus p_2) = ds(p_1) + ds(p_2)$$

$$ds(p_1 \mid p_2) = \{p_1 \mid p_2' : p_1' \in ds(p_1), p_2' \in ds(p_2)\}$$

$$ds(p[E]) = \{p'[E] : p' \in ds(p)\}$$

$$ds(\text{if } b \text{ then } p \text{ else } q) = \begin{cases} ds(p) & \text{if eval}(b) = \text{tt} \\ ds(q) & \text{if eval}(b) = \text{ff} \end{cases}$$

$$ds(p(e_1, \ldots, e_n)) = ds(q[v_1/x_1, \ldots, v_n/x_n])$$
where \( \text{eval}(e_i) = v_i, i = 1, \ldots, n \).

so for example

\[
ds(B! + C!) = \{B! + C!\}
\]

whereas

\[
ds(B! \otimes C!) = \{B!, C!\}.
\]

Note however that

\[
ds(A!(B! \otimes C!)) = \{A!(B! \otimes C!}\}
\]

so \( ds \) only looks at the topmost level of a process. Note also that \( ds(p) \) is finite and non-empty for any \( p \).

The relations \( \rightarrow \) are now defined so that whenever \( p \rightarrow q \), \( q \) is a deterministic state, that is \( ds(q) = \{q\} \) (We may say \( q \) is determinate, or a determinate term). So a computation evolves by a process changing, under the stimulus of actions, from deterministic state to deterministic state. However, a computation may begin from a nondeterministic state and it is only for this state that the derivation rule, given below, for \( \otimes \) is required.

**Definition 6.2**

Let \( \rightarrow \) be the least relation contained in Terms x Terms such that the following rules hold:

1. **Inaction**

   Nil has no transitions.

2. **Action** (different from chapter 1)

   2.1 \( 1.p \rightarrow p' \) if \( p' \in ds(p) \)

   2.2 \( \text{eval}(e) = v, p' \in ds(p) \)  

   \[ \frac{}{A!e.p \rightarrow A!v} p' \]

   2.3 \( A?x.p \rightarrow A?v \) if \( p' \in ds(p) \).
3. Choice (as chapter 1)

3.1 \[ p \rightarrow^a p' \]
\[ p+q \rightarrow^a p' \]

3.2 \[ q \rightarrow^a q' \]
\[ p+q \rightarrow^a q' \]

4. Indeterminacy

4.1 \[ p \rightarrow^a p' \]
\[ p\&q \rightarrow^a p' \]

4.2 \[ q \rightarrow^a q' \]
\[ p\&q \rightarrow^a q' \]

This definition may seem a little odd at first since it gives exactly the same derivations to \( p\&q \) and \( p+q \). However, it can be justified on the grounds that it is only used in the first step of a computation. In the testing theory we are interested in the set of all possible computations of an experiment (which in this case is a term) and not (so much) in the particular computation we get in a single trial. Hence, the fact that we have given

\( (1): \ ((A! \& B!) \mid A?) \ \land \ A \)

only the single computation

\( ((A! \& B!) \mid A?) \ \land \ A \rightarrow^1 (Nil \mid Nil) \ \land \ A \)

which is essentially identical with the only computation of

\( (2): \ ((A! + B!) \mid A?) \ \land \ A \)

is (for our purposes) redressed by the following convention.

Convention The computations in \( \text{comp}(p) \) of length zero are precisely the determinate terms \( p' \in ds(p) \) such that \( p' \rightarrow^? \).

Then \( (1) \) has a single computation of length zero, viz.

\( (B! \mid A?) \ \land \ A \)

while \( (2) \) has none.

\[ \square \]

5. Parallel (different from chapter 1)
The substitution of determinate terms is necessary so that all derivatives are determinate.

5.3 \( p \rightarrow p', \; q \rightarrow q' \)
\[ p|q \rightarrow p'|q' \]

6. Renaming (as chapter 1)

6.1 \( p \rightarrow p' \)
\[ p[E] \rightarrow p'[E] \]

If \( F(a) \) is defined, where
\[
F(A) = \begin{cases} 
1 & \text{if } a = 1 \\
E(A)v & \text{if } a = A!v, \; E(A) \text{ defined} \\
E(A)?v & \text{if } a = A?v, \; E(A) \text{ defined} \\
\text{undefined} & \text{otherwise} 
\end{cases}
\]

7. Conditional (as chapter 1)

7.1 \( \text{beval}(b) = \text{tt}, \; p \rightarrow p' \)
\[ \text{if } b \text{ then } p \text{ else } q \rightarrow p' \]

7.2 \( \text{beval}(b) = \text{ff}, \; q \rightarrow q' \)
\[ \text{if } b \text{ then } p \text{ else } q \rightarrow q' \]

Note that by Definition 6.1 these rules will only be used at the beginning of a computation.

8. Procedure call (as chapter 1)

Suppose \( D(P(x_1, \ldots, x_n)) = p \) is defined, then
\[ p[v_1/x_1, \ldots, v_n/x_n] \rightarrow p' \]
\[ P(e_1, \ldots, e_n) \rightarrow p' \]
where \( \text{eval} (e_i) = v_i, i = 1, \ldots, n \)

Note that, in common with the rules for \( \Theta \), this rule will only be used at the beginning of a computation since the function \( ds \) substitutes the body of the call immediately on encounter. It is important to note that in what follows we will not bother to distinguish the body from the call, employing the call as a description of the body. This is permissible because in only one case will the body of the call (to \( WS^n(M) \)) not be a determinate term and for that case we will distinguish body from call.

\( \square \)

The following proposition holds;

**Proposition 6.1**

\[
p \overset{\approx}{\rightarrow} p' \iff \exists dp \in ds(p). dp \overset{\approx}{\rightarrow} p'
\]

\( \square \)

Next we introduce a little notation to enable simple description of complicated terms,

**Notation** In analogy with + and \( \sum \) we introduce

\[
\overset{\otimes}{\sum_{k \in K}} q_{k} \quad \text{to denote} \quad q_{k_1} \otimes \cdots \otimes q_{k_n}
\]

where \( K = \{k_1, \ldots, k_n\} \) is some finite index set for \( n \geq 1 \). This is permissible since \( \otimes \) will behave at all times like an associative commutative operator.

\( \square \)

The definitions so far are fine if we are dealing with determinate terms but we really want to talk about indeterminate terms during
the course of a computation. From the definition of the transition relations \( \rightarrow \) we have seen such terms don't occur within a computation. However, it will be useful in the proof of the may part of implementation to behave as if they did occur; this will allow us to defer decisions about which of a number of possible processes has been chosen, we will know it is one of the definite states but not which one.

We really want to act as if we had a slightly different transition relation wherein indeterminacy can remain unresolved arbitrarily far past its first encounter, the essential idea being that if \( p \rightarrow a r \) and \( p \rightarrow a q \) then if \( p \rightarrow a p' \) it's possible that \( p' = q \) or \( p' = r \), which we would like to write as

\[ p \rightarrow a q \otimes r \]

which may be read "p may perform the action a, thereby becoming one of q or r". We define this new transition relation between (indeterminate) terms as follows:

**Definition 6.3**

\[ p \rightarrow a q \iff \forall dq \in ds(q). p \rightarrow a dq \]

Note \( ds(q) \) is always non-empty and if \( q \) is determinate then the relationship is as before (since \( ds(q) = \{q\} \) in that case).

Thus for example: 

1. \( (p \otimes q) \rightarrow a p \otimes q \) and \( (1 . p \otimes 1 . q) \rightarrow a p \otimes q \).

At this point we should emphasise that the 'true' transition relation is \( \rightarrow \) whereas \( \rightarrow \) will be useful for establishing relationships in \( \rightarrow \) such as:
Proposition 6.2
\[ p \stackrel{\cdot}{\rightarrow} \stackrel{\cdot}{\rightarrow} q \text{ implies } \forall dq \in ds(q). \ p \stackrel{\cdot}{\rightarrow} dq \]

Proof by induction on the length of the derivation \( \stackrel{\cdot}{\rightarrow} \) using Proposition 6.1.

6.1.2. Extending the set of tests

So far we have decided that an experiment is a CCS term (possibly indeterminate) and that success is achieved when that term reaches a derivative which offers the distinguished action \( \omega \). Since we are dealing in this chapter with the relation implements between objects rather than experiments it is necessary to separate test and object in an experiment. To this end we introduce tests as contexts and objects as terms not containing the action \( \omega \), so only the test part may signal success.

A test \( t \), often written \( t[\_] \) to distinguish it as such, is a CCS term with a single distinguished parameterless procedure call to its argument. Interfacing a test involves substitution of this procedure call by the argument.

Definition 6.4

Suppose we have a distinguished procedure name \( \text{Arg} \in \text{Proc} \), then a test \( t \) is a closed term with a single occurrence of \( \text{Arg} \) as a subterm. Writing \( t[p] \) for \( t[p/\text{Arg}] \) define the set \( T \) of tests, ranged over by \( t \), by

\[ t ::= \text{Arg} \ | \ p|t \ | t[E] \ | t_1[t_2] \]

where \( p \) ranges over Terms. We will assume each test \( t \) is syntactically valid, i.e. \( \vdash t \). \( \text{Arg} \) is assumed to have no
definition, its occurrence serves merely as a marker for the point of substitution. We may write \[.\] for Arg when describing tests, thus for example

\[ p \mid \text{Arg} \] may be written \[ p \mid [.] \]

\[ \square \]

The objects to be tested are valid CCS terms not involving \( \omega \), so the signal of success must come from the test.

**Definition 6.5**

The set \( P \) of objects is defined by

\[ P = \{ p : \omega \notin FL(p), \models p \} \]

\[ \square \]

The application of a test to an object is of course achieved by substituting the object for Arg, yielding a CCS term (an experiment) with the following property:

**Proposition 6.3**

If \( t \in T \) and \( p \in P \) then

\[ ds(t[p]) = \{ t'[p'] : t' \in ds(t), p' \in ds(p) \} \]

\[ \square \]

Now, in analysing and synthesising transitions of experiments it is useful to separate the actions of test and object. To this end we introduce transition relations for contexts.

**Definition 6.6**

Suppose we add to our definition set \( D \) the definitions

\[ \text{Test}_a \triangleq a.1.Did_a \]

\[ \text{Did}_a \triangleq \text{Nil} \]
for every $a \in AD$ (see chapter 1 section 2.2), assuming $(\text{Test}_a, 0)$, $(\text{Did}_a, 0) \in \text{Defined}(D)$, then for every possible action we have a process $\text{Test}_a$ which offers it alone. The derivation rules for tests are then as follows for $t \in T$:

1. $t[\text{Test}_a] \xrightarrow{a} t'[1.\text{Did}_a]$

The consequent should be read as

If $t$'s argument may perform action $a$ then the whole term may perform action $a$ as $t$ becomes $t'$, a test for whatever the argument has become.

The reason for not defining $\text{Test}_a \leftarrow a.\text{Did}_a$

is that from our definition of $ds$ and the action rules,

$\text{Test}_a \xrightarrow{a} \text{Nil}$

whereas

$\text{Test}_a \xrightarrow{a} 1.\text{Did}_a$

The second relationship is better for keeping track of the point of substitution, for the subterm $1.\text{Did}_a$ will remain at the point of substitution of $\text{Test}_a$ but $\text{Nil}$ may occur at many points in a term $t'[\text{Nil}]$ so we might have

$t'[\text{Nil}] = t''[\text{Nil}], t' \neq t''$

in which case we have lost track of what the test was. Proposition 6.4 (1) will affirm the property we require.

Note that the consequent of rule 1 is not intended to describe a new form of action or transition relation for general CCS terms, it is merely a device for describing how a test interacts with its
argument.

In the case \( a = 1 \), which is the most interesting to us, we write

\[ t[.] -[a] \rightarrow t'[.]. \]

2. \( t[\text{Test}_a] \rightarrow q \rightarrow t'[\text{Test}_a] \)

\[ t[.] -[a] \rightarrow t'[.]. \]

The consequent may be read

Regardless of \( t \)'s argument the whole term performs action \( a \),
\( t \) becoming \( t' \) and the argument remaining unchanged.

\[ \Box \]

The definition of test transitions now enables, and is justified by, the following proposition addressing the relationship between test and object in a single transition step of an experiment.

**Proposition 6.4**

Suppose \( t \in T, p \in P \) then

1. \( t[\text{Test}_a] \rightarrow q \Rightarrow q \ implies \)

\[ \exists \ unique \ t'. \ (q = t'[\text{Test}_a] \ or \ q = t'[1. \text{Did}_a]) \]

2. \( t[.] \rightarrow t'[.]. \ implies \)

\[ \forall p \in P. \ \exists \ dp' \in \text{ds}(p). \ t[p] \rightarrow t'[dp'] \]

3. \( t[.] \rightarrow t'[.] \ implies \ t' \in \text{ds}(t) \)

4. \( t[p] \rightarrow q \iff \exists t', p'. \ q = t'[p'] \) and one of the following holds

1. \( t[.] \rightarrow t'[.], p' \in \text{ds}(p) \)

2. \( \exists a. \ t[.] \rightarrow[a] t'[.], p \rightarrow p' \)

\[ \Box \]
From now on we will describe the progression of experiments as
\[ t[p] \rightarrow t'[p'] \rightarrow t''[p''] \rightarrow \ldots \]
as justified by Proposition 6.4 (4).

We are now in a position to define \( NEXP(\text{CCS}) \) and refer the reader to chapter 4 section 1.1 for general motivation.

**Definition 6.7**

An experiment is the application of a test to an object:

\[ \exp(NEXP(\text{CCS})) = \{ t[p] : t \in T, p \in P \} \]

An experiment is successful if it offers \( \omega \):

\[ \text{succ}(NEXP(\text{CCS})) = \{ e : e \in \exp(NEXP(\text{CCS})), e \rightarrow \omega \} \]

An experiment proceeds by the derivation rules of CCS:

\[ \text{trans}(NEXP(\text{CCS})) = \rightarrow \]

\[ \Box \]

**6.2. The weavesort machine**

The example we have chosen is taken from the area of VLSI design wherein great emphasis is placed on regularity of structure and a high degree of concurrency. The task to be performed by the machine is that of sorting a multiset of natural numbers into numerical order. Of course the machine will be of finite capacity so there is a limit to the size of the multiset.

**6.2.1. An informal description of weavesort**

A weavesort machine of size \( n \), capable of sorting a multiset of \( 2n \) values, is a linear array of \( n \) sorting cells each of which may
hold at most 2 values. The machine is assumed to be initially empty so each cell is empty. Diagrammatically this state is represented as

```
input   _______ = _______ = ... = _______
          |      |      |      |      |
output   --- --------
```

where each box represents an empty cell, and the connections between cells are written '='. Each cell, except the rightmost, is connected to its right neighbour by a data path across which values can pass in either direction. Data enters and leaves the array through the left cell only, the rightmost cell only communicates with its left neighbour.

The action of this machine is most easily understood in terms of the actions of its constituent cells. The action of a single cell depends on how many values it holds in its two registers (a left register to hold the smaller value and a right register to hold the larger value (if two are values are held)) according with the following rules.

R.0. An empty cell is willing to receive a value from its left and become a cell with one value.

R.1.1. A cell with one value is willing to receive another value from its left and become a cell with two values, the smaller in its left register and the larger in its right.

R.1.2. Alternatively, a cell with one value is also willing to deliver its only value to the left and thereby become a cell with no values.
R.2.1. A cell with two values (the smaller in its left register, the larger in its right) is willing to receive a value from its left and then pass the value in its right register to the right. (Note that the cell is momentarily holding three values, a point we shall return to later), thereby becoming a cell with the smaller of its remaining values in its left register and the larger in its right.

R.2.2. Alternatively, a cell with two values is also willing to deliver its smaller value to the left and then request a value from its right. If the request is granted then it remains a cell with two elements, shuffling its two values to get the smaller in its left register. If the request is denied then it becomes a cell holding its one remaining value.

R.3. If not requested to perform any action it can perform a cell (of any number of values) will do nothing.

An example should make this clear; suppose $n = 2$ so we have two cells and can sort multisets of up to 4 elements. In this informal presentation we will describe the progress of the machine diagrammatically where

\[
\begin{array}{c}
\text{\_\_\_}\text{\_}\text{\_}\text{\_}\text{\_}
\end{array}
\]

is a cell with no values.

\[
\begin{array}{c}
\text{v}\text{\_\_\_}\text{\_}\text{\_}\text{\_}\text{\_}
\end{array}
\]

is a cell with the single value $v$ in its left register.
| u | v |
---|---|

is a cell with two elements; u (≠ v) in its left register and v in its right.

\[ d_1 \rightarrow v_1 \rightarrow d_2 \] means v is output (to the left) by the machine (diagram) \( d_1 \), thereby becoming \( d_2 \).

\[ d_1 \rightarrow v_2 \rightarrow d_2 \] means v is input (from the left) by the machine (diagram) \( d_1 \), thereby becoming \( d_2 \).

In describing a computation we will write under each cell of \( d_1 \) the rule it applies for that transition.

So, suppose we want to sort the multiset \([1, 2, 3, 4]\); then all we have to do is input them in any order to the empty machine (of size 2 in our case) and then accept the values it offers back:

\[
\begin{array}{c}
| \quad \quad \quad | = | \quad \quad \quad |
\
R.0 & R.3
\end{array}
\]

\[
\begin{array}{c}
| 2 \quad \quad | = | \quad \quad |
\
R.1.1 & R.3
\end{array}
\]

\[
\begin{array}{c}
| 2 \quad 4 | = | \quad \quad |
\
R.2.1 & R.0
\end{array}
\]

\[
\begin{array}{c}
| 1 \quad 2 | = | 4 \quad \quad |
\
R.2.1 & R.1.1
\end{array}
\]

\[
\begin{array}{c}
| 1 \quad 3 | = | 2 \quad 4 |
\
\end{array}
\]

Note the machine is now full so we must now start removing
elements. The subject of what happens if we try to keep feeding in more values is deliberately under-defined in this informal description.

\[
\begin{array}{c}
13 \rightarrow 24 \\
\text{R.2.2} \text{ R.2.2}
\end{array}
\]

\[
\begin{array}{c}
23 \rightarrow 4 \\
\text{R.2.2} \text{ R.1.2}
\end{array}
\]

\[
\begin{array}{c}
34 \rightarrow 100000 \\
\text{R.2.2} \text{ R.3}
\end{array}
\]

\[
\begin{array}{c}
4 \rightarrow 500000 \\
\text{R.1.2} \text{ R.3}
\end{array}
\]

Thus the numbers are retrieved in numerical order and the machine state ends as it began, with two empty cells.

6.2.2. The description and specification in CCS

The aim of this subsection is to formalise, in CCS, the informal description of the weavesort machine introduced in the previous subsection. For the purposes of explanation we will find it useful to have a close relationship between the diagrammatic descriptions used earlier and the CCS terms we will define. Each sorting-cell-agent we define can be viewed diagrammatically as
where we may refer to the left as the 'front' and

l, r are value passing channels modelling those data paths to
the left and right (respectively) of a cell.

emptyl, emptyr are used for signalling and sensing
(respectively) an empty cell.

x, y are bound variables modelling the two registers of a sort
cell; x will hold the smaller value if two values are held.

A weavesort machine of size n containing a multiset M of values
(|M| ≤ 2n) is a linear array of n sorting cells over which the
elements of M are distributed in a particular fashion. The
description of such a machine, to be written WS^n(M) is best
understood inductively on its size n and thence by case analysis
on its contents M:

BASE n = 0: The machine is willing to receive values from its
left but immediately forgets them. However, it is always willing
to admit it is empty (i.e. it contains no values). We describe it
in CCS by an agent 'stopper' and diagrammatically by '=':

        stopper <- l?z.stopper
        + emptyl!.stopper

STEP Suppose we have weavesort machines of size n-1 for all
multisets of size up to 2(n-1): we construct weavesort machines
of size n for multisets M of size up to 2n by case analysis on M.
The (inductive) aim of the construction is always to keep the
smallest value held in the leftmost register of the leftmost cell.

case $M = \emptyset$: We want an 'empty cell' 'connected' to a 'smaller weavesort machine holding no values'. Thus if we describe

'empty cell' by CELLO

'connected' by $\leftrightarrow$

'smaller weavesort machine holding no values'

by $WS^{n-1}(\emptyset)$

then, first in CCS and then diagrammatically, we want

\[
\text{CELEO} \leftrightarrow WS^{n-1}(\emptyset)
\]

We still need to define $\leftrightarrow$ and CELLO in CCS, beginning with the former which is a notational convention:

Definition 6.8

Let $p, q$ be CCS terms, then

\[
p \leftrightarrow q = (p[m, empty/r, empty/r] \mid q[m, empty/l, empty/l]) \setminus \{m, empty\}
\]

The intended use of $\leftrightarrow$ is that $q$ is some smaller sorting machine, a linear array of cells, and $p$ is a cell to go on the front; i.e.

\[
p \leftrightarrow q
\]

So we connect $q$'s left value channel $l$ to $p$'s right value channel $r$ and $q$'s empty-signalling channel $empty_l$ to $p$'s empty-sensing channel $empty_r$. Since $\leftrightarrow$ acts like an associative operator the
above diagrammatic description of $q$ is permissible.

Now, an empty cell can either receive a value from the left, thereby becoming a cell holding that value, or it can signal to the left that it is empty (since our construction will guarantee that everything to its right is also empty):

$$\text{CELL0} \leftarrow \begin{cases} 1?z.\text{CELL0}(z) \\ \text{emptyll.CELL0} \end{cases}$$

**case $M = \{m\}$:** we want a cell containing just $m$, the smallest value in $M$, connected to a smaller machine holding no values; ie

$$\text{CELL1}(m) \leftrightarrow WS^{n-1}(\emptyset)$$

A cell with a single value $x$ can receive another value $z$, thereby becoming a cell with two values, $x$ and $z$, the smaller being in the left register. Alternatively, it can output $x$ to the left and become a cell with no values (our construction will guarantee that everything to its right is empty).

$$\text{CELL1}(x) \leftarrow 1?z.\text{SWAP(x,z)} + \text{l!x.CELLO}$$

and

$$\text{SWAP}(x,y) \leftarrow \text{if } x < y \text{ then CELL2}(x,y) \text{ else CELL2}(y,x)$$

**case $M = M^{++}[m,v], m = \min(M)$:** This is the most interesting case. The leftmost cell must hold the smallest value $m$ and some (in fact any) other value $v$ from $M$. The remainder of $M$, i.e. $M'$, is distributed recursively over $WS^{n-1}(M')$; thus we want:

$$\Phi_{v \in M--[m]}(CELL2(m,v) \leftrightarrow WS^{n-1}(M--[m,v]))$$
We expressed our indifference to the choice of \( v \) by using \( \theta \); this allows succinct expression of a number of machines with similar behaviour, i.e. they will all be implementations of the specification to follow. However, their behaviour can appear very different; consider (loosely):

\[
A: \begin{bmatrix} 1 & 2 \\ \hline 3 & 4 \end{bmatrix} \in \varepsilon \text{ ds}(\text{WS}^2([1,2,3,4]))
\]

\[
B: \begin{bmatrix} 1 & 4 \\ \hline 2 & 3 \end{bmatrix} \in \varepsilon \text{ ds}(\text{WS}^2([1,2,3,4]))
\]

If we add a new value, say 0, then the value at the right end (i.e. 4 for A, 3 for B) 'drops off':

\[
A
\begin{array}{c}
\text{0i}\rightarrow
\\
\begin{bmatrix} 0 & 1 \\ \hline 2 & 3 \end{bmatrix} \in \varepsilon \\
\text{0i}\rightarrow
\\
\begin{bmatrix} 1 & 2 \\ \hline 3 & \_ \end{bmatrix} \in \varepsilon \\
\text{2i}\rightarrow
\\
\begin{bmatrix} 2 & 3 \\ \hline \_ & \_ \end{bmatrix} \in \varepsilon \\
\text{3i}\rightarrow
\\
\begin{bmatrix} 3 & \_ & \_ & \_ \\ \hline \_ & \_ & \_ & \_ \end{bmatrix} \in \varepsilon
\end{array}
\]

\[
B
\begin{array}{c}
\text{0i}\rightarrow
\\
\begin{bmatrix} 0 & 1 \\ \hline 2 & 4 \end{bmatrix} \in \varepsilon \\
\text{0i}\rightarrow
\\
\begin{bmatrix} 1 & 2 \\ \hline 4 & \_ \end{bmatrix} \in \varepsilon \\
\text{2i}\rightarrow
\\
\begin{bmatrix} 2 & 4 \\ \hline \_ & \_ \end{bmatrix} \in \varepsilon \\
\text{3i}\rightarrow
\\
\begin{bmatrix} 4 \_ & \_ & \_ & \_ \\ \hline \_ & \_ & \_ & \_ \end{bmatrix} \in \varepsilon
\end{array}
\]

Thus \( \text{WS}^n(M) \) will describe a family of possible machines (i.e. \( \text{ds}(\text{WS}^n(M)) \)) each with slightly different behaviour. We have chosen to abstract away from a complete-description of the
distribution of elements of the multiset $M$ over $WS^{2n}(M)$ because such a description would (probably) depend on the exact structure of a weavesort machine and be very unwieldy. However, it can easily be seen that every distribution described (i.e., every member of $ds(WS^{2n}(M))$) can arise (the proof of implementation establishes the converse) by the following informal argument.

Claim Let $n \geq 0$, $M$ a multiset of not more than $2n$ natural numbers; then any of the distributions described above of $M$ over an $n$ cell weavesort machine is attainable by a sequence of $|M|$ inputs to $WS^n(\phi)$.

Proof by induction on $n$ and cases on $M$:

BASE $n = 0$: Trivial.

STEP Suppose the result is true for $n-1$;

case $M = \phi$: Trivial (the empty sequence).

case $M = [m]$: Just input $m$.

case $M = M'[++[m,v]]$, $m = \min(M)$: by the inductive hypothesis we have, for any distribution of $M'$ in $WS^{n-1}(M')$, a sequence $v_1, \ldots, v_k$, $k \leq n-1$, of values such that $M' = [v_1, \ldots, v_k]$ and inputting this sequence to $WS^{n-1}(\phi)$ produces that distribution.

The required sequence for the larger machine is then $m, v_1, \ldots, v_k, v$.

It remains to describe CELL2 more formally. A cell with two elements has two options, firstly it can accept a value from the left and then output to the right its larger value for storage by the smaller weavesort machine. It remains as a cell with two elements but now with the new smallest value (i.e., the smaller of the new value and the old smallest value) in the left register.
Alternatively, a cell with two elements can output to the left the current smallest value and then receive from the right either a value or a signal that the smaller machine is empty. If a value is received it will be (by recursive reasoning) the smallest value held by the smaller machine; comparing this with the value remaining in this leftmost cell determines the new smallest value. If the empty signal is received then the only value remaining is that in this leftmost cell.

\[
\text{CELL2}(x,y) \leftarrow 1?z. y. \text{SWAP}(x,y) + 1?x. (r?z. \text{SWAP}(y,z) + \text{ emptyr?}. \text{CELL1}(y))
\]

Note there is a slight problem here with the interpretation of \text{CELL2} as a VLSI structure; after receipt of a new value from the left \text{CELL2} is holding (i.e. binding in CCS) 3 values in supposedly only 2 registers. Moving to a synchronous description of the machine would solve this problem, but for our purposes it doesn't really matter.

Gathering together all these definitions we get the following definition set:

Let \(n \geq 0\), \(M \in \text{multisets}(\mathbb{Nat})\), \(|M| \leq 2n\), \(m = \min(M)\) if \(M \neq \phi\);

\[
\text{WS}^n(M) \leftarrow \begin{cases} 
\text{stopper} & \text{if } n = 0 \\
\text{CELLO} & \text{if } M = \phi \text{ then } \text{CELLO} \leftrightarrow \text{WS}^n(\phi) \\
\text{CELL1}(m) & \text{if } M = [m] \text{ then } \text{CELL1}(m) \leftrightarrow \text{WS}^{n-1}(\phi) \\
\bigwedge_{v \in M \setminus [m]} (\text{CELL2}(m,v) \leftrightarrow \text{WS}^{n-1}(M \setminus [m,v])) \\
\text{stopper} & \text{stopper} \quad + \text{ emptyr}. \text{stopper}
\end{cases}
\]
Having described a weavesort machine we now turn to the task of finding a concise expression of its intended behaviour.

Let \( n \geq 0 \), \( M \in \text{multisets}(\text{Nat}) \), \( |M| \leq 2n \), \( m = \min(M) \) if \( M \neq \emptyset \); then define a sorting machine, of capacity for \( 2n \) elements, holding \( M \) by:

\[
\text{SORT}^{2n}(M) \leftarrow \begin{cases} 
\emptyset & \text{if } |M| = 0 \\
1!m.\text{SORT}^{2n}(M-[m]) & \text{if } |M| > 0 \\
1?z.\text{SORT}^{2n}(M+[z]) & \text{if } |M| < 2n \\
1?z.\text{SORT}^{2n}((M+[z])-\emptyset) & \text{if } |M| = 2n
\end{cases}
\]

Informally this may be read:

1. If the machine is empty it can signal that fact.

2. If the machine holds any values it should be prepared to deliver the smallest, adjusting its contents accordingly.

3. If the machine is not yet full it should be prepared to accept and hold another value.
4. If the machine is full it should still be prepared to accept values but it is conceded that, since its capacity is exceeded, some (unspecified) value must be lost from the contents.

It is the indeterminacy of this last clause which enables \(\text{SORT}^{2n}(M)\) to describe every definite state of \(\text{WS}^n(M)\); in each such process the element to be thrown away is completely determined by its distribution of values.

6.2.3. The statement of correctness

We would like some way of saying that \(\text{SORT}^{2n}(M)\) is a 'good-description' of \(\text{WS}^n(M)\) for appropriate \(n\) and \(M\). So far we have seen three possible ways of expressing this:

a) \(\text{WS}^n(M) = \text{SORT}^{2n}(M)\)

b) \(\text{WS}^n(M)\) completely-implements \(\text{SORT}^{2n}(M)\)

c) \(\text{WS}^n(M)\) implements \(\text{SORT}^{2n}(M)\)

The point we wish to make in this subsection is that of the above only c) holds, for the following reason.

The weavesort machine \(\text{WS}\) is always willing to accept another value even if it means throwing away a value already held (or just accepted). Which value gets thrown away depends on the distribution of values in the machine, which in turn depends on the history of the machine (as was shown in the earlier Claim).

The sorting machine described by \(\text{SORT}\) is also willing to accept a value, only to throw another away if it is full. However, the
description of \textsc{sort} is sufficiently perspicuous that we can see it may throw away \textit{any} value it holds, or has just accepted, if its capacity is exceeded.

It is not the case, in general, that \textsc{ws} has license to throw away whichever value it chooses, so, in general again, \textsc{sort} has more capabilities than \textsc{ws}. In fact \textit{c)} will express our ignorance of exactly what \textsc{ws} will throw away, but if \textsc{sort} must pass a test then so must \textsc{ws}, and \textsc{ws} cannot do anything \textsc{sort} cannot do.

With a few examples we will show why \textit{a)} and \textit{b)} do not hold in general.

\textbf{Example} \textsc{ws}^2([1,2,3,4]) \# \textsc{sort}^4([1,2,3,4])

First consider the definite states of \textsc{ws}^2([1,2,3,4]):
\[
\text{ds(\textsc{ws}^2([1,2,3,4])) = }
\{ \text{CELL2(1,2) <-> CELL2(3,4) <-> stopper, } \\
    \text{CELL2(1,3) <-> CELL2(2,4) <-> stopper, } \\
    \text{CELL2(1,4) <-> CELL2(2,3) <-> stopper} \}
\]

Now, the sorting machine can accept a new value (5, say) and throw away any value in [1,2,3,4], so we choose 2 just to be awkward:
\[
\textsc{sort}^4([1,2,3,4]) \rightarrow \textsc{sort}^4([1,3,4,5])
\]

However
\[
\textsc{ws}^2([1,2,3,4]) \rightarrow 5 \rightarrow \textsc{p}
\]
implies \textsc{p} still holds the value 2, so
\[
\textsc{p} \rightarrow 1 \rightarrow 2
\]
but
\[
\text{\# q. \textsc{sort}^4([1,3,4,5]) \rightarrow \rightarrow q}
\]

Hence there can be no bisimulation containing the pair.
\( (WS^2([1,2,3,4]), \text{SORT}^h([1,2,3,4])) \)

For exactly the same reasons b) is not in general true:

Example  In NEXP(CCS)

\( WS^2([1,2,3,4]) \text{ completely-implements } \text{SORT}^h([1,2,3,4]) \):

As in the bisimulation example

\[ 1!5.1?x.1?y.\text{if } y=2 \text{ then } \omega I WS^2([1,2,3,4]) \text{ must} \]

whereas

\[ 1!5.1?x.1?y.\text{if } y=2 \text{ then } \omega I \text{SORT}^h([1,2,3,4]) \text{ must} \]

It only remains to show why c) is an appropriate way of saying that \( \text{SORT}^{2n}(M) \) is a 'good description' of \( WS^n(M) \). The idea is that the behaviour of the machines differs only when an overflow occurs, i.e. too many values have been input. The weavesort machine will throw away a value determined completely, but in a non-trivial way, by its initial distribution of values and its subsequent interactions. The abstract sorting machine can throw away any of the values it holds. One of the values the sorting machine could throw away is the value actually thrown away by the weavesort machine. Thus if we are given the definition of \( \text{SORT}^{2n}(M) \) and told

\( WS^n(M) \text{ implements } \text{SORT}^{2n}(M) \)

then we can deduce when overflow occurs but not exactly which value is lost.
6.3. Proving implementation

In this section we prove that \( \forall n \geq 0, M \in \text{multisets(Nat)}, |M| \leq 2n \).

\( WS^n(M) \) implements \( \text{SORT}^{2n}(M) \)

that is \( \forall t \in T. \)

\( t[WS^n(M)] \text{ may implies } t[\text{SORT}^{2n}(M)] \text{ may,} \)

\( t[\text{SORT}^{2n}(M)] \text{ must implies } t[WS^n(M)] \text{ must} \)

We have already seen in the preceding section that \( \text{SORT}^{2n} \) has more capabilities than \( WS^n \) so the relationship of complete-implementation or of having a bisimulation does not hold. The most we can hope for is implementation, and that is what we shall prove.

6.3.1. Proving the may part

In this subsection we aim to prove (loosely)

\( t[WS^n(M)] \text{ may implies } t[\text{SORT}^{2n}(M)] \text{ may} \)

and we achieve this essentially by noting that if \( t[WS^n(M)] \text{ may} \)

then it is because \( WS^n(M) \) can perform some sequence of actions which may satisfy \( t \), and any process which can perform this sequence can similarly satisfy \( t \). We accordingly introduce the notion of simulation wherein if \( p_1 \text{ simulates } p_2 \) then any sequence of (external) actions performable by \( p_1 \) is performable by \( p_2 \) also.

Demonstrating that \( WS^n(M) \text{ simulates } \text{SORT}^{2n}(M) \) then completes the proof.

We begin by employing a strong version of half the bisimulation relation in the notion of 'simulation', whence we can employ something similar to half of the bisimulation proof technique.
The basic idea is to assume that the process being simulated has no silent actions, as is often reasonable when it is a specification.

**Definition 6.9**

A relation $S \subseteq \text{Terms} \times \text{Terms}$ is a simulation if whenever $p \xrightarrow{S} q$,
1. $p \xrightarrow{-1} p'$ implies $p' \xrightarrow{S} q$
2. $p \xrightarrow{-a} p'$, $a \neq 1$ implies $q \xrightarrow{-a} q'$, $p' \xrightarrow{S} q'$

If $S$ is a simulation and $p \xrightarrow{S} q$ then we say $p$ simulates $q$.

We would now like to slightly extend Definition 6.9, via the following proposition, so as to remove the need for pairs of the form $(S^n(H), p)$ in the simulation.

**Proposition 6.5**

$(\forall \ dp \in \text{ds}(p). \ dp \text{ simulates } q)$ iff $p \text{ simulates } q$.

**Proof**

Suppose $dp \in \text{ds}(p)$, $p \xrightarrow{S} q$, $S$ a simulation, then take $S' = S + \{(dp, q)\}$

Since $S$ is a simulation we need only examine the pair $(dp, q)$

**case** $dp \xrightarrow{-1} p'$: then by Proposition 6.1

$p \xrightarrow{-1} p'$

so

$p' \xrightarrow{S} q$.

**case** $dp \xrightarrow{-a} p'$, $a \neq 1$: then by Proposition 6.1

$p \xrightarrow{-a} p'$, $q \xrightarrow{-a} q'$, $p' \xrightarrow{S} q'$.

The converse is proved in a similar manner.

**Proposition 6.6**
If \( t[p] \rightarrow t'[p'] \), \( p \) simulates \( q \)
then \( ds(t') = \{ t' \} \) and one of the following holds:
1. \( t[q] \rightarrow t'[q] \), \( p' \) simulates \( q \)
2. \( t' \in ds(t) \), \( p \rightarrow t' \)
3. \( q' \). \( t[q] \rightarrow t'[q'] \), \( p' \) simulates \( q' \)

Proof we proceed by cases on Proposition 6.4 (4),

**case** \( t[.] \rightarrow t'[.], p' \in ds(p) \): then by Proposition 6.4 (2)

and Proposition 6.3
\( t[q] \rightarrow t'[q] \)

and by Proposition 6.5
\( p' \) simulates \( q \).

**case** \( t[.] \rightarrow t'[.], p \rightarrow t'[.] \): then by Proposition 6.4 (2)

\( t' \in ds(t) \).

**case** \( t[.] \rightarrow t'[.], p \rightarrow a \rightarrow t'[.] \), \( a = 1 \): then

\( q', q' \rightarrow a \rightarrow q', p' \) simulates \( q' \)

Hence by Definition 6.3 and Proposition 6.4 (4)

\( \forall dq' \in ds(q') \). \( t[q] \rightarrow t'[dq'] \)

so by Definition 6.3 and Proposition 6.3

\( t[q] \rightarrow t'[q'] \)

since

\( t' = ds(t') \)

\( \Box \)

**Corollary 6.1**

If \( t[p] \rightarrow \rightarrow t'[p'] \), \( p \) simulates \( q \)
then \( t'', q'. t' \in ds(t'') \), \( t[q] \rightarrow t''[q'] \), \( p' \) simulates \( q' \)

Proof by induction on the length \( n \) of the derivation \( \rightarrow' \),

**case** \( n = 0 \): Immediate by Proposition 6.6.

**case** \( n > 0 \): then
\[ t_1, p_1, t[p] \rightarrow \Rightarrow t_1[p_1] \rightarrow t'[p'] \]

so

\[ t_2, q_1, t[q] \rightarrow^* t_2[q_1], t_1 \in ds(t_2), p_1 \text{ simulates } q_1 \]

By Proposition 6.6 we have 3 cases;

**case** \[ t_1[q_1] \rightarrow^* t'[q_1], ds(t') = \{t'\}, p' \text{ simulates } q_1: \]

So by Proposition 6.1

\[ t_2[q_1] \rightarrow t'[q_1] \]

**case** \[ t' \in ds(t_1), p_1 \rightarrow^1 p': \text{ then by Definition 6.9} \]

\[ p' \text{ simulates } q_1 \]

and

\[ t' = t_1 \in ds(t_2) \]

since

\[ ds(t_1) = \{t_1\}. \]

**case** \[ t'[q'], t_1[q_1] \rightarrow^1 t'[q'], p' \text{ simulates } q': \]

then by Proposition 6.1 and Proposition 6.3

\[ t_2[q_1] \rightarrow t'[q']. \]

Now we can state and prove the principle proposition by which we establish \( t[\text{WS}^n(M)] \text{ may } \Rightarrow t[\text{SORT}^2n(M)] \text{ may } \)

**Proposition 6.7**

\( p \text{ simulates } q \text{ implies } \forall t \in T. \ (t[p] \text{ may } \Rightarrow t[q] \text{ may}) \)

**Proof** Suppose \( p \text{ simulates } q \text{ and } t[p] \text{ may } \), then by Definition 4.8 and Proposition 6.4 there is some successful computation

\(<t_k[p_k]>_k \)

such that if \( K = \{0\} \) then

\[ t[p] = t_0[p_0] \rightarrow t_1[p_1] \rightarrow ... \]

and if \( K = \{0\} \) then we have a computation of length 0 and, by our
convention and Proposition 6.3,
\[ t_0 \in ds(t), \ p_0 \in ds(p) \]

**case** \( K = \{0\} \): then by Proposition 6.4 and Definition 6.5
\[ t_0[.] \mathrel{\cdots} \]

so by Proposition 6.1 and Proposition 6.3
\[ t[.] \mathrel{\cdots} \]

Now we have two cases depending on the computations of \( t[q] \):
- **case** \( t[q] \rightarrow \): then \( t[q] \) is the first configuration in some computation, and by Proposition 6.4 \( t[q] \mathrel{\cdots} \).
- **case** \( t[q] \nrightarrow \): then we have a computation of zero length, but by Proposition 6.3 and Proposition 6.4 (2)
\[ \forall q_0 \in ds(q), t_0[q_0] \in ds(t[q]), t_0[q_0] \mathrel{\cdots} \]

**case** \( K = \{0\} \): then
\[ j \geq 1, t[p] \rightarrow \rightarrow t_j[p_j] \mathrel{\cdots} \]

so by Corollary 6.1
\[ q', t'^n. t[q] \mathrel{\rightarrow^*} t^n[q'], t_j \in ds(t^n) \]

whence by Proposition 6.1 and Proposition 6.2
\[ r. t[q] \mathrel{\rightarrow^*} r \mathrel{\cdots} \]

\( \Box \)

The next step is now clear, we need to prove that for each \( n \geq 0 \) and \( M \in \text{multisets(Nat)}, |M| < 2n \)
\[ \forall dp \in ds(WS^n(M)), WS^n(M) \text{ simulates SORT}^{2n}(M) \]

We do this by defining for each \( n \geq 0 \) the required simulation \( S_n \) for weavesort machines of \( n \) cells and any \( M, |M| \leq 2n \), by induction on \( n \):  

**Definition 6.10**

**BASE** \( n=0 \):
$S_0 = \{\text{(stopper, SORT}^0(\phi))\}$

**STEP:** Suppose we have constructed $S_{n-1}$, define $S_n$ to be the union of the following sets

1. $\{(\text{CELL0} \leftrightarrow p, \text{SORT}^{2n}(\phi)):$
   
   \[ p \ s_{n-1} \ \text{SORT}^{2(n-1)}(\phi) \]

2. $\{(\text{CELL1}(x) \leftrightarrow p, \text{SORT}^{2n}(\{x\})):$
   
   \[ p \ s_{n-1} \ \text{SORT}^{2(n-1)}(\phi) \]

3. $\{(\text{CELL2}(m,v) \leftrightarrow p, \text{SORT}^{2n}(N)):$
   
   \[ p \ s_{n-1} V_{n} \ \text{SORT}^{2(n-1)}(\{m,v\}), \]
   
   \[ M \subseteq N_{s} \subseteq |M| < 2n, [m,v] \subseteq M, m = \text{min}(M)\]

4. $\{(\text{r}1v. \text{SWAP}(m,v') \leftrightarrow p, \text{SORT}^{2n}(N++[v'])):$
   
   \[ p \ s_{n-1} V_{n} \ \text{SORT}^{2(n-1)}(\{m,v\}), \]
   
   \[ M \subseteq N_{s} \subseteq |M| < 2n, [m,v] \subseteq M, m = \text{min}(M)\]

5. $\{(\text{r}1v. \text{SWAP}(m,v') \leftrightarrow p, \text{SORT}^{2n}(N)):$
   
   \[ p \ s_{n-1} V_{n} \ \text{SORT}^{2(n-1)}(\{m,v\}), \]
   
   \[ M \subseteq N_{s} \subseteq |M| = 2n, [m,v] \subseteq M, m = \text{min}(M), \]
   
   \[ N_{s} = \{N++[v']\}--[w]: N \in N_{s}, w \in N++[v']\}

6. $\{(r?x. \text{SWAP}(v,x)++emptyp? \\text{CELL1}(v) \leftrightarrow p, \text{SORT}^{2n}(N)):$
   
   \[ p \ s_{n-1} V_{n} \ \text{SORT}^{2(n-1)}(\{v\}), \]
   
   \[ M \subseteq N_{s} \subseteq |M| < 2n, [v] \subseteq M\]

Note if Val were allowed to be infinite then $\text{SORT}^{2n}(N)$ could be an infinite indeterminate summation in many of the above cases.

The motivations behind this seemingly complex definition are quite
simple, regarding a n+1 cell weavesort machine as a single cell on the front of the SORT machine of size 2(n-1). We motivate S_{n-1} and S_n by considering a simplified first attempt at both:

\[3.1\] \(\{\text{CELL2}(m,v) \leftrightarrow p, \text{SORT}^{2n}(M)\}:\)
\[p S_{n-1} \text{SORT}^{2(n-1)}(M--[m,v]),
\]
\[M \in \{M: |M| < 2n, [m,v] \subseteq M, m = \min(M)\}\]

\[5.1\] \(\{\text{riv.SWAP}(m,v') \leftrightarrow p, \quad \emptyset_{w \in M++[v']} \text{SORT}^{2n}(M++[v']--[w])\}:
\[p S_{n-1} \text{SORT}^{2(n-1)}(M--[m,v]),
\]
\[M \in \{M: |M| = 2n, [m,v] \subseteq M, m = \min(M)\}\]

For example \(S_n[5.1]\) might (loosely) be read;

Suppose \(M \in \text{multisets(Nat)}, |M| = 2n, [m,v] \subseteq M, m = \min(M),\)
and \(p\) simulates \(\text{SORT}^{2(n-1)}(M--[m,v])\), then

\(\text{riv.SWAP}(m,v') \leftrightarrow p\) simulates \(\emptyset_{w \in M++[v']} \text{SORT}^{2n}(M++[v']--[w])\)

This is essentially because \(\text{SORT}^{2(n-1)}(M--[m,v])\) has put an 'upper bound' on what \(p\) can do, so we can substitute \(\text{SORT}^{2(n-1)}(M--[m,v])\) for \(p\) to get an upper bound on what \(\text{riv.SWAP}(m,v') \leftrightarrow p\) can do.

So now we see the first problem; the occurrences of 'simulates' above should be replaced by \(S_{n-1}\) in the first instance and \(S_n\) in the second. This means that what will be the inductive hypothesis is of the form

\(q S_{n-1} \text{SORT}^{2(n-1)}(N)\)

while the consequent is of the form

\(q' S_n \emptyset_{N \in N_S} \text{SORT}^{2n}(N)\)

Thus we need a slightly weaker hypothesis, replacing [5.1] by
[5.2] 

\{ rlv.SWAP(m,v') \leftrightarrow p, \quad \emptyset_{M \subseteq \text{MSR}} \text{SORT}^{2n}(M) ; \}

\begin{align*}
& \quad p \quad \emptyset_{M \subseteq \text{MSR}} \text{SORT}^{2(n-1)}([M \rightarrow (m,v')], \\
& \quad M \subseteq [M] : |M| = 2n, [m,v'] \subseteq M, m = \min(M), \\
& \quad \text{MSR} = \{M ++[v'] \rightarrow [w] : M \in MS, w \in M ++[v']\}
\end{align*}

Note we have rewritten

\emptyset_{M \subseteq \text{MSR}} \text{SORT}^{2n}(M)

as

\emptyset_{M \subseteq \text{MSR}} \text{SORT}^{2n}(M)

so that Proposition 6.10 (2.1) will hold.

The second problem arises when considering the case

\begin{align*}
& \quad \text{rlv.SWAP}(m,v') \leftrightarrow p \quad \rightarrow \quad p' \quad \rightarrow \quad q, \\
& \quad p \quad \rightarrow \quad p', q = \text{SWAP}(m,v') \leftrightarrow p',
\end{align*}

then by Proposition 6.11 and the inductive hypothesis

\begin{align*}
& \quad p' \quad \emptyset_{M \subseteq \text{MSR}} \text{SORT}^{2n}(M'), \\
& \quad \text{MSR} \subseteq \{M \rightarrow [m,w] : M \in MS, w \in M \rightarrow [m]\}
\end{align*}

However, we want to use \( S_n[3.1] \). If we used (1) in \( S_n[3.1] \) then we would get

\begin{align*}
& \quad q \quad S_n[3.1] \quad \emptyset_{M \subseteq \text{MSR}} \text{SORT}^{2n}(M'') \quad \text{where}
\end{align*}

\begin{align*}
& \quad \text{MSR}'' = \{M ++[v'] \rightarrow [w] : M \in MS, w \in M \rightarrow [m]\}
\end{align*}

and in general \( \text{MSR}'' \) may be a proper subset of \( \text{MSR} \). What has happened?

The problem is that we have gained information about which values can be thrown away (viz. we can't throw away \( m \) or \( v' \)) but this gain of information can't be expressed in even a silent transition of the indeterminate \text{SORT} machine. To get around this we draw a veil over our information gain, essentially by noting that
\[ \forall q_1, q_2, r. \ q_1 \text{ simulates } q_2 \text{ implies } q_1 \text{ simulates } q_2 \otimes r \]

By restricting the allowable forms of \( r \), for our purposes, to suitable sorting machines we can use a set \( N_sR \), bounded above by conditions of suitability and below by conditions of necessity, to give the desired leeway. This is the purpose of the set \( N_sR \) in [3] and [5]; to see how it works in practice this example is covered formally in the proof.

For each \( n, S_n \) has some simple properties which will be useful in proving \( S_n \) to be a simulation,

**Proposition 6.10**

Let \( n \leq 0, M \in \text{ multisets}(\text{Nat}), |M| \leq 2n \), then

1. \( \forall pp \in \text{ ds}(WS^n(M)). \ pp \ S_n \text{ SORT}^{2n}(M) \)

2. If \( pp \ S_n q \) then \( \exists \) finite \( M_s \subseteq \text{ multisets}(\text{Nat}) \) such that
   1. \( q = \bigoplus_{M \in M_s} \text{ SORT}^{2n}(M) \)
   2. \( \forall M, M' \in M_s. \ |M| = |M'| \)

**Proof** by induction on \( n \).

Hereafter we allow \( M_s \) to range only over finite subsets of \( \text{ multisets}(\text{Nat}) \) and write \( |M_s| \) when referring to the cardinality of the members of \( M_s \) rather than the cardinality of \( M_s \) itself.

From Proposition 6.10 (2) we can see that we will be interested in transitions between particular forms of indeterminate terms. We need some way of employing the definition of \( \text{ SORT} \) to generate these transitions,

**Proposition 6.11**

Suppose \( n \leq 0, \) and \( M_s, N_s \subseteq \text{ multisets}(\text{Nat}) \), then
\[
\text{SORT}_2^n(M) \overset{\beta}{\rightarrow} \text{SORT}_2^n(N)
\]

iff

\[
\text{NS} \subseteq \{N : \text{M} \in \text{Ms. SORT}_2^n(M) \overset{\beta}{\rightarrow} \text{SORT}_2^n(N)\}
\]

Proof Immediate by Definition 6.3.

\[\square\]

Proposition 6.12

Suppose \( n \geq 0, \text{M} \in \text{multisets(Nat)}, |\text{M}| \leq 2n \), then

\( \text{WS}_n^2(\text{M}) \) simulates \( \text{SORT}_2^n(\text{M}) \)

Proof We will prove

\( \forall \text{pp} \in \text{ds(WS}_n^2(\text{M})), \text{pp} \) simulates \( \text{SORT}_2^n(\text{M}) \)

and we do this by showing

1. \( \forall \text{pp} \in \text{ds(WS}_n^2(\text{M})), \text{pp} \text{SN} \text{SORT}_2^n(\text{M}) \)

2. \( \text{SN} \) is a simulation.

Part 1 has already been shown in Proposition 6.10. Part 2 most properly proceeds by induction on \( n \) but, as with bisimulations, the great amount of case analysis and duplication in the proof dictates that, in this presentation, we consider two exemplars only; pairs of the forms given in \( S_n[3] \) and \( S_n[5] \).

We begin with \( S_n[3] \), so assume \( S_{n-1} \) is a simulation,

\[
(\text{CELL}_{2(m,v)} \leftrightarrow p, \overset{\beta}{\text{SORT}_2^{n-1}(M)} \in S_{n-1}[3])
\]

and

\[
\text{CELL}_{2(m,v)} \leftrightarrow p \overset{\beta}{\rightarrow} r.
\]

We proceed by cases on the derivation:

**Case a=1, r = CELL_{2(m,v)} \leftrightarrow p', p \overset{\beta}{\rightarrow} p':**

From the inductive hypothesis,

\[
p' S_{n-1} \overset{\beta}{\text{SORT}_2^{n-1}(M-\text{[m,v]})}
\]

Hence,
\[
\begin{align*}
&\text{case } a = \exists m, r = (r \exists z. \text{SWAP}(v, z) + \text{empty} r, \text{CELL} 1(v)) \iff p: \text{ then} \\
&\quad r \ S_n[3] \ \overset{\delta}{\overset{N \in \text{NS}}{\text{SORT}^{2n}(N)}} \end{align*}
\]

where
\[
N_s' = \{N - [m] \in \text{NS} \}
\]

By Proposition 6.11
\[
\overset{\delta}{\overset{N \in \text{NS}}{\text{SORT}^{2n}(N)}} \overset{\exists \exists m, r}{\overset{N \in \text{NS}, \text{SORT}^{2n}(N')}}
\]

\[
\begin{align*}
\text{case } a = \exists \exists v', r = \exists v'. \text{SWAP}(m, v') \iff p: \\
\quad \text{We have two cases depending on whether or not the submachine } p \\
\quad \text{will overflow upon receipt of the value } v, \\
\quad \text{case } |M_s| < 2^n: \text{ then the submachine won't overflow and} \\
\quad r \ S_n[4] \ \overset{\delta}{\overset{N \in \text{NS}}{\text{SORT}^{2n}(N+ [v'])}} \\
\quad \text{By Proposition 6.11} \\
\quad \overset{\delta}{\overset{N \in \text{NS}}{\text{SORT}^{2n}(N)}} \overset{\exists \exists m, r}{\overset{N \in \text{NS}, \text{SORT}^{2n}(N + [v'])}}
\end{align*}
\]

\[
\begin{align*}
\text{case } |M_s| = 2^n: \text{ then the submachine will overflow and} \\
\quad r \ S_n[5] \ \overset{\delta}{\overset{N \in \text{NS}, \text{SORT}^{2n}(N')}} \\
\quad \text{where} \\
\quad N_s' = \{(N + [v']) - [w] \in \text{NS}, w \in N + [v']\}
\end{align*}
\]

By Proposition 6.11
\[
\overset{\delta}{\overset{N \in \text{NS}}{\text{SORT}^{2n}(N)}} \overset{\exists \exists m, r}{\overset{N \in \text{NS}, \text{SORT}^{2n}(N')}}
\]

That concludes the proof for \(S_n[3]\), so now we turn to \(S_n[5]\);

**suppose** \(S_{n-1}\) is a simulation and

\[
(\exists v'. \text{SWAP}(m, v') \iff p, \overset{\delta}{\overset{N \in \text{NS}}{\text{SORT}^{2n}(N)}) \in S_n[5])
\]

and

\[
\exists v'. \text{SWAP}(m, v') \iff p \overset{-A}{\iff} r
\]

We proceed by cases on the derivation

\[
\text{case } a = 1, r = \text{SWAP}(m, v') \iff p', p \overset{\exists \exists v'}{\iff} p,'
\]

From the inductive hypothesis, Proposition 6.10 (2.1),
Proposition 6.11 and the definition of \( \text{SORT} \) we have
\[
p' \overset{\varnothing}{\longrightarrow} R_{n-1} \overset{\text{SORT}}{\longrightarrow} 2n(M'),
\]
\[
S\{M[M=m, w], M \in Ms, w \in M[M=m]\}
\]
\[
= \{(M[M=m, w], M \in Ms, w \in M[M=m])
\}
\]
\[
C \{(N[M=m, w], N \in Ns, w \in N[M=m])
\}
\]
\[
C \{(N[M=m, w], N \in Ns, w \in N[M=m])
\}
\]
\[
= \{N[M=m, w], N \in NsR\}
\]
thus
\[
r S_n[5] \overset{\varnothing}{\longrightarrow} R_{n} \overset{\text{SORT}}{\longrightarrow} 2n(N)
\]

\textbf{case a=1, r = r1v.SWAP(m,v') \iff p', p \rightarrow p':}

From the inductive hypothesis, Proposition 6.10 (2.1), Proposition 6.11 and the definition of \( \text{SORT} \) we have
\[
p' \overset{\varnothing}{\longrightarrow} R_{n-1} \overset{\text{SORT}}{\longrightarrow} 2n(M[M=m, v])
\]
so
\[
r S_n[5] \overset{\varnothing}{\longrightarrow} R_{n} \overset{\text{SORT}}{\longrightarrow} 2n(N)
\]

\textbf{Corollary 6.2}

Suppose \( n \leq 0 \), \( M \in \text{multisets(Nat)} \), \( |M| \leq 2n \), then
\[
\forall t \in T. (t[WS^n(M)] \text{ may implies } t[\text{SORT}^{2n}(M)] \text{ may})
\]

\textbf{6.3.2. Proving the must part}

In this section we aim to prove (loosely)
\[
t[\text{SORT}^{2n}(M)] \text{ must implies } t[WS^n(M)] \text{ must}
\]
To see how we will do this consider for the moment the case \( n>0 \), \( \{m,v\} \subseteq M, m = \min(M); \) the proof has three steps:
1. We will show

\[ \forall v \in M \rightarrow [m] \cdot t[CELL2(m,v) \leftrightarrow SORT^{2(n-1)}(M \rightarrow [m,v])] \text{ must implies } t[WS^n(M)] \text{ must} \]

that is we can substitute the \(SORT^{2(n-1)}\) specification for the \(WS^n\) implementation in every possible experiment. This is achieved essentially by regarding the front cell together with \(t\) as a test \(t_v\)

\[ t_v[.] = t[CELL2(m,v) \leftrightarrow [.] ] \]

on the smaller machine \(WS^n\).

2. We prove the antecedent in 1. by showing that for any \(k \geq 0\),

\[ t[SORT^{2n}(M)] \text{ must}_k \text{ implies } \forall v \in M \rightarrow [m] \cdot t_v[SORT^{2(n-1)}(M \rightarrow [m,v])] \text{ must}_{2k} \]

where we define for \(k \geq 0\)

\[ e \text{ must}_k \text{ if within } k \text{ transition steps } e \text{ must succeed} \]

whence

\[ e \text{ must}_k \text{ implies } e \text{ must} \]

This is achieved via the notion of a 'ranking', allowing a bisimulation-style proof technique.

3. We establish the antecedent in 2. by showing

\[ t[SORT^{2n}(M)] \text{ must} \text{ implies } \forall k \geq 0. \ t[SORT^{2n}(M)] \text{ must}_k \]
To show this we need to restrict $Val$, the set of values passable in a communication (chapter 1 section 2) to be a finite set, then we are considering finitely branching computation trees and we can apply Konigs Lemma.

Having described the proof 'top-down' we present it bottom-up, thus the order of presentation is 3, 2, and then 1.

6.3.2.1. Proving part 3

As in Chapter 5 the proof of part 3 is motivated by considering the computation trees of experiments, regarding the set of computations of an experiment $e$ as a tree $\text{comptree}(e)$. Suppose $e$ is an experiment and $e$ must, then every computation (i.e. maximal path through the tree) must begin with a finite derivation leading to a successful configuration. Now suppose we have a finitely branching computation tree $T$ and consider the tree $T'$ with every sub-tree of $T$ whose root is a successful configuration replaced by some distinguished leaf. Then $T'$ must be finite (by Konigs Lemma) since all the derivations are finite. Hence the tree $T'$ is of finite depth $k \geq 0$, so $e$ must succeed within $k$ steps, which we formalise in the following predicate:

**Definition 6.11**

Define the family of predicates $\text{must}_k$ $k \geq 0$ by

\[
\begin{align*}
\text{e must}_0 & \iff \text{success}(e) \\
\text{e must}_{k+1} & \iff \\
& \text{either } 1. \text{success}(e) \\
& \text{or } 2. \forall e'. e \to e',
\end{align*}
\]
\( \forall e'. \ e \rightarrow e' \text{ implies } e' \text{ must}_k \)

\( \Box \)

Now we formally state, but don't formally prove here, the upshot of the above informal argument,

**Proposition 6.13**

Suppose \( e \) is an experiment, \( e \text{ must} \), and \( \text{comptree}(e) \) is finitely branching, then \( \exists k \geq 0. \ e \text{ must}_k \)

\( \Box \).

It now remains to prove that the experiments we are interested in have finitely branching computation trees,

**Proposition 6.14**

Suppose \( p \) is a term and we restrict \( \text{Val} \) (the set of values passable in a communication) to be finite, then \( \text{comptree}(p) \) is finitely branching.

**Proof** by structural induction on \( p \); the restriction on \( \text{Val} \) is required for terms \( a.p' \). Note we have not allowed infinite summation (either \( + \) or \( \otimes \)) in the construction of terms.

\( \Box \)

From this point on we consider \( \text{Val} \) to be a finite set, and so

**Corollary 6.3**

\( \forall n \geq 0, \ M \in \text{multisets}(\text{Nat}), \ |M| \leq 2n, \ t \in T. \)

\( \ t[\text{SORT}^{2n}(M)] \text{ must} \text{ implies } \exists k \geq 0. \ t[\text{SORT}^{2n}(M)] \text{ must}_k \)

\( \Box \)

This concludes part 3 of the proof.
6.3.2.2. Proving part 2

We will now employ the results of part 3 in the following fashion,

Proposition 6.15

Suppose \( M \in \text{multisets}(\text{Nat}) \), \( |M| \leq 2n \), \( t \in T \), and

\[ t[\text{SORT}^{2n}(M)] \text{ must}_k \]

for some \( k \geq 0 \), then

1. If \( M = \emptyset \) then

\[ t[\text{CELLO} \leftrightarrow \text{SORT}^{2(n-1)}(\emptyset)] \text{ must}_{2k} \]

2. If \( M = [m] \) then

\[ t[\text{CELL1}(m) \leftrightarrow \text{SORT}^{2(n-1)}(\emptyset)] \text{ must}_{2k} \]

3. If \( 2 \leq |M| \leq 2n \), \( m = \min(M) \) then

\[ \forall v \in M-[m]. \ t[\text{CELL2}(m,v) \leftrightarrow \text{SORT}^{2(n-1)}(M-[m,v])] \text{ must}_{2k} \]

The factor of two difference between the \( \text{must}_k \) of the antecedent and the \( \text{must}_{2k} \) of the consequents arises because a cell holding two values must communicate with both the user and the SORT machine of size \( n-1 \) in order to model the single communication between the user and the SORT machine of size \( n \). Thus if we were to replace the smaller SORT machine by a smaller weavesort machine we would get another \( n-1 \) factors of 2, but that doesn't concern us here.

To prove this proposition we introduce the notion of 'rankings', closely mirroring the definition of \( \text{must}_k \).

Definition 6.12

A ranking of experiments is an indexed family of sets \( \{R_k\}_{k \geq 0} \) such that
1. $0 \in K \subseteq \text{Nat}$

2. All $k \in K, R^k \subseteq \text{Terms}$

3. $q \in R_0$ implies $\text{success}(q)$

4. If $j \in K, q \in R_j$ then
   
   either 1. $\text{success}(q)$
   
   or 2. $\exists q'. q \rightarrow q'$,

   $\forall q'. (q \rightarrow q' \implies \exists i \in K. i < j, q' \in R_i)$

Our intent in this definition is to emulate the bisimulation proof technique; to show $e \text{ must}, j \geq 0$, (and thereby $e \text{ must}$) we merely need provide a ranking $<R^j_K>$ such that $e \in R_j, j \in K$. Then, by 4, either it is already successful or it can progress and every possible progression brings it closer to success. This is formalised in the following proposition,

**Proposition 6.16**

Suppose $<R_j^K>$ is a ranking, $j \in K, q \in R_j$, then

1. $q \text{ must}_j$.

2. $\forall dq \in ds(q). dq \text{ must}_j$

**Proof** by induction on $j \in K$.

So now, in trying to prove Proposition 6.15, we will construct a ranking $<R_j^K>$ by defining, for each $i \geq 0$, $R_{2i}$ and $R_{2i+1}$ as follows;

**Definition 6.13**

Suppose $i \geq 0$, then define:

$R_{2i}$ is the union of the following three sets:
\[1\] \( t[\text{CELLO} \leftrightarrow \text{SORT}^{2(n-1)}(\phi)]:\n\quad t[\text{SORT}^{2n}(\phi)] \text{ must}_1 \]

\[2\] \( t[\text{CELL1}(m) \leftrightarrow \text{SORT}^{2(n-1)}(\phi)]:\n\quad t[\text{SORT}^{2n}(m)] \text{ must}_1 \)

\[3\] \( t[\text{CELL2}(m,v) \leftrightarrow \text{SORT}^{2(n-1)}(M--[m,v])]:\n\quad t[\text{SORT}^{2n}(M)] \text{ must}_1, \n\quad [m,v] \subseteq M, \ m = \min(M), \ |M| \leq 2n \)

\(R_{21+1}\) is the union of the following three sets:

\[4\] \( t[r!v.\text{SWAP}(m,v') \leftrightarrow \text{SORT}^{2(n-1)}(M--[m,v])]:\n\quad t[\text{SORT}^{2n}(M++[v'])] \text{ must}_1, \n\quad [m,v] \subseteq M, \ m = \min(M), \ |M| < 2n \)

\[5\] \( t[r!v.\text{SWAP}(m,v') \leftrightarrow \text{SORT}^{2(n-1)}(M--[m,v])]:\n\quad \forall \ w \in M++[v']. \ t[\text{SORT}^{2n}(M++[v']--[w])] \text{ must}_1 \n\quad [m,v] \subseteq M, \ m = \min(M), \ |M| = 2n \)

\[6\] \( t[(r?z.\text{SWAP}(v,z)+\emptyset r?.\text{CELL1}(v)) \leftrightarrow \text{SORT}^{2(n-1)}(M--[m,v])]:\n\quad t[\text{SORT}^{2n}(M--[m])] \text{ must}_1, \n\quad [m,v] \subseteq M, \ m = \min(M), \ |M| \leq 2n \)

\[\Box\]

**Proposition 6.17**

\(\langle R_1 \rangle_{\text{Nat}}\) is a ranking.

**Proof** we prove each of the conditions of Definition 6.12;

1. Trivial.

2. Trivial.

3. Firstly we note we are dealing with the case \(i=0\). We will just deal with \(R_0[1]\) since all cases are exactly the same; suppose
q = t[\text{CELL} \leftrightarrow \text{SORT}^{2(n-1)}(\phi)] \in R_0[1]

and

t[\text{SORT}^{2n}(\phi)] \text{ must}_0

thus by Proposition 6.4 and Definition 6.11

t[.] \xrightarrow{\omega^1}

so, again by Proposition 6.4

q \xrightarrow{\omega^1}, \text{i.e. success}(q)

4. As with a bisimulation proof there is a great deal of case analysis so, for the purposes of this presentation, we take two exemplars;

\textbf{case [3]} Suppose

q = t[\text{CELL}2(m,v) \leftrightarrow \text{SORT}^{2(n-1)}(M\rightarrow[m,v])] \in R_2[3]

then, following the definition of a ranking, either success(q), in which case there is nothing further to prove, or by Proposition 6.4

t[.] \xrightarrow{\omega^1}

so success(t[\text{SORT}^{2n}(M)]). We do know, however,

t[\text{SORT}^{2n}(M)] \text{ must}_1

and hence by Proposition 6.4, Proposition 6.11, and definitions of \text{SORT} and \text{must}_1, we have some of the following (i.e. at least one):

1. t[.] \xrightarrow{1} t'[.], t'[\text{SORT}^{2n}(M)] \text{ must}_{1-1}
2. |M| < 2n, t[.] \xrightarrow{[1?v']} t'[.], t'[\text{SORT}^{2n}(M++[v'])] \text{ must}_{1-1},
3. |M| = 2n, t[.] \xrightarrow{[1?v']} t'[.],

\forall \ w \in M++[v']. t'[\text{SORT}^{2n}(M++[v']-\rightarrow[w])] \text{ must}_{1-1}
4. t[.] \xrightarrow{[1\uparrow m]} t'[.], t'[\text{SORT}^{2n}(M\rightarrow-[m])] \text{ must}_{1-1}

We now show how the above cases correspond exactly with derivations of q, i.e. if any of the above hold then q has a
corresponding derivative and any derivative of \( q \) is further
down the ranking. We proceed by cases on all the possible
derivatives of \( q \), using Proposition 6.4, Proposition 6.11
and the definitions of CELL2 and SORT; we have \( q \rightarrow q' \) iff
one of the following cases holds:

**case** \( t[. \downarrow \rightarrow t'[. \downarrow \downarrow \),
\[ q' = t'[CELL2(m,v) \leftrightarrow \text{SORT}2(n-1)(M \rightarrow [m,v])] \]

if this case holds then 1. above does, so we have
\[ t'[\text{SORT}2n(M)] \text{ must}_{i-1} \]
and hence
\[ q \rightarrow q' \in R_{2(i-1)}[3] \]

**case** \( |M| < 2n, t[. \downarrow \downarrow [m \rightarrow v'] \rightarrow t'[. \downarrow \downarrow \),
\[ q' = t'[r!v.Swap(m,v') \leftrightarrow \text{SORT}2(n-1)(M \rightarrow [m,v])] \]

if this case holds then 2. above does, so
\[ t'[\text{SORT}2n(M++[v'])] \text{ must}_{i-1} \]
and hence
\[ q \rightarrow q' \in R_{2i-1}[4] \]

**case** \( |M| = 2n, t[. \downarrow \downarrow [m \rightarrow v'] \rightarrow t'[. \downarrow \downarrow \),
\[ q' = t'[r!v.Swap(m,v') \leftrightarrow \text{SORT}2(n-1)(M \rightarrow [m,v])] \]

if this case holds then 3. above does, so
\[ \forall w \in M++[v']. t'[\text{SORT}2n(M++[v'] \rightarrow [w])] \text{ must}_{i-1} \]
and hence
\[ q \rightarrow q' \in R_{2i-1}[5] \]

**case** \( t[. \downarrow \rightarrow t'[. \downarrow \downarrow \),
\[ q' = t'[\_ \leftrightarrow \text{SORT}2(n-1)(M \rightarrow [m,v])] \]

if this case holds then 4. above does, so
\[ t'[\text{SORT}2n(M \rightarrow [m])] \text{ must}_{i-1} \]
and hence
That concludes the case for pairs from $R_{21}[3]$, the other exemplar concerns pairs from $R_{21+1}[5]$;

**case [5]** Suppose

$$q = t \mapsto t[rv].\text{SWAP}(m,v') \leftrightarrow \text{SORT}^2(n-1)(M--[m,v]) \in R_{21+1}[5]$$

then, following the definition of a ranking, either $\text{success}(q)$ in which case there is nothing further to prove, or by Proposition 6.4

$$t[.] \rightarrow^1 \gamma$$

so $\text{success}(t[\text{SORT}^2(n)(M)])$ which we now assume. We proceed by cases on the (non-empty) set of possible derivatives $q'$ of $q$, using Proposition 6.4, Proposition 6.11 and the definitions of $\text{SORT}$ and $\text{CELL2}$:

**case** $t[.] \rightarrow^1 t'[.]$,

$$q' = t'[rv].\text{SWAP}(m,v') \leftrightarrow \text{SORT}^2(n-1)(M--[m,v])$$

then $\forall w \in M++[v']$.

$$t[\text{SORT}^2(n)(M++[v']--[w])] \rightarrow t'[\text{SORT}^2(n)(M++[v']--[w])] \text{ must}_1$$

so

$$q \rightarrow q' \in R_{21-1}[5]$$

**case** $t[.] \rightarrow^1 t'[.], \ t' \in ds(t)$,

$$w \in M--[m]. \ q' = t'[\text{SWAP}(m,v')] \leftrightarrow \text{SORT}^2(n-1)(M--[m,w])$$

But we have

$$\forall w \in M++[v'], \ t[\text{SORT}^2(n)(M++[v']--[w])] \text{ must}_1$$

so by Proposition 6.3 and Proposition 6.16 (2)

$$\forall w \in M++[v'], \ \forall t' \in ds(t).$$

$$t'[\text{SWAP}(m,v')] \leftrightarrow \text{SORT}^2(n-1)(M--[m,w]) \text{ must}_1$$

then

$$q \rightarrow q' \in R_{21}[3]$$
Proof of Proposition 6.15:

We merely note that the ranking \( <R_i>_{\text{Nat}} \) is such that the statement of Proposition 6.15 with \textbf{must} replaced by \( R \) is seen by simple inspection of the pairs \([1], [2], \) and \([3]\) (writing \( R \) as a postfix predicate). The result is then immediate by Proposition 6.16.

6.3.2.3. Proving part 1

In part 1 of the proof we analyse the possible forms of \( t[s^n(M)] \), for \( n > 0 \), to discover forms more amenable to manipulation. We begin by noting that we will only have to deal with the definite states of the weavesort machine.

Proposition 6.18

\[
t[p] \text{ must iff } (\forall dt[dp] \in ds(t[p]). dt[dp] \text{ must})
\]

Proof We demonstrate only the reverse implication here; suppose

\[
\forall dt[dp] \in ds(t[p]). dt[dp] \text{ must}
\]

By Proposition 6.1 and our convention on computations of zero length

\[
<t_k[p_k]>_k \in \text{comp}(t[p])
\]

iff

either

\[ K = [0], t_0[p_0] \in ds(t[p]), t_0[p_0] \rightarrow \]

or

\[ K = [0], t[p] = t_0[p_0] \rightarrow t_1[p_1] \rightarrow \ldots \]

In the former case the supposition implies the computation is successful. In the latter case we employ Proposition 6.1 and
Proposition 6.3 to get
\[
\int \text{d}t[dp] \in \text{ds}(t[p]). \, \text{d}t[dp] \rightarrow t_1[p_1] \rightarrow \ldots
\]
which must, by the supposition, be a successful computation. The only problem that might arise is if
\[
\text{d}t[dp] \rightarrow \text{w1}, \, t[p] \rightarrow \text{w1}
\]
but, by Proposition 6.1, this is impossible.
\[\Box\]

This enables us to employ the following proposition when dealing with non-trivial weavesort machines,

**Proposition 6.19**

Suppose \( n \geq 0 \), and \( \forall t \in T, M \in \text{multisets}(\text{Nat}), |M| \leq 2(n-1). \)
\[
t[\text{SORT}^2(n-1)(M)] \text{ must implies } t[\text{WS}^n(M)] \text{ must}
\]
then for any \( t \in T \)
1. \( t[\text{CELL0} \leftrightarrow \text{SORT}^2(n-1)(\phi)] \text{ must} \)
   implies
   \[
t[\text{WS}^n(\phi)] \text{ must}
\]
2. \( t[\text{CELL1} (m) \leftrightarrow \text{SORT}^2(n-1)(\phi)] \text{ must} \)
   implies
   \[
t[\text{WS}^n([m])] \text{ must}
\]
3. If \( M \in \text{multisets}(\text{Nat}), 2 \leq |M| \leq 2n, m = \min(M), \) then
   \[
   (\forall v \in M-[m]. \, t[\text{CELL2}(m,v) \leftrightarrow \text{SORT}^2(n-1)(M-[m,v])] \text{ must})
   
   implies
   \[
t[\text{WS}^n(M)] \text{ must}
\]

**Proof**

1. Suppose
   \[
t[\text{CELL0} \leftrightarrow \text{SORT}^2(n-1)(\phi)] \text{ must}
\]
and take
\[ t' = t[\text{CELLO} \leftrightarrow \text{[]}] \in T \]

Then
\[ t[\text{CELLO} \leftrightarrow \text{SORT}^2(n-1)(\phi)] = t'[\text{SORT}^2(n-1)(\phi)] \]

What we have done is essentially to move the dividing line between test and object, taking part of the object and moving it into the test:

\[
\begin{array}{c|c}
\text{t} & \text{CELLO} \leftrightarrow \text{SORT}^2(n-1)(\phi) \\
\hline
\text{test} & \hline
\text{object} & \\
\end{array}
\]

rewrites as

\[
\begin{array}{c|c}
\text{t} & \text{CELLO} \leftrightarrow \text{SORT}^2(n-1)(\phi) \\
\hline
\text{new-test} & \hline
\text{new-object} & \\
\end{array}
\]

Now, by our supposition we must have
\[ t'[\text{SORT}^2(n-1)(\phi)] \text{ must implies } t'[\text{WS}^n(\phi)] \text{ must} \]

but the antecedent has been shown true, so it just remains to see
\[ t'[\text{WS}^n(\phi)] = t[\text{CELLO} \leftrightarrow \text{WS}^n(\phi)] = t[\text{WS}^n(\phi)] \]

by moving the dividing line back again.

2. The reasoning is exactly the same as in case 1.

3. Suppose \( M \in \text{multisets(Nat)}, 2 \leq |M| \leq 2n, m = \min(M) \)

and
\[ \forall v \in M--[m]. t[\text{CELL2}(m,v) \leftrightarrow \text{SORT}^2(n-1)(M--[m,v])] \text{ must} \]

then
\[ \forall v \in M--[m]. t_v[\text{SORT}^2(n-1)(M--[m,v])] \text{ must} \]

where
\[ t_v = t[\text{CELL2}(m,v) \leftrightarrow \text{[]} \in T \]

so
\forall v \in M^{[m]}, t_v[WS^{n-1}(M^{[m,v]})] \text{ must}

that is

\forall v \in M^{[m]}, t[\text{CELL2}(m,v) \leftrightarrow WS^{n-1}(M^{[m,v]})] \text{ must}

Now, applying Proposition 6.18,

\forall v \in M^{[m]}, \forall dp \in ds(WS^{n-1}(M^{[m,v]})). \forall dt \in ds(t).

dt[\text{CELL2}(m,v) \leftrightarrow dp] \text{ must}

and so by Definition 6.1 and Proposition 6.3

\forall q \in ds(WS^n(M)). t[q] \text{ must}

and so by Proposition 6.18 again

\forall WS^n(M) \text{ must}

\hfill \Box

6.3.2.4. Gathering the parts

We now gather the three parts together as described earlier

Proposition 6.20

Suppose \( n \geq 0, M \in \text{multisets(Nat), } |M| \leq 2n \), then

\( \forall t \in T. (t[\text{SORT}^{2n}(M)] \text{ must implies } t[WS^n(M)] \text{ must}) \)

Proof by induction on \( n \),

BASE \( n=0 \): Immediate from the definitions of \( WS^0(\phi) \) and \( \text{SORT}^0(\phi) \)

STEP Suppose the result is true for \( n-1 \) and

\( t[\text{SORT}^{2n}(M)] \text{ must} \)

then by Corollary 6.3

\( \forall k \geq 0. t[\text{SORT}^{2n}(M)] \text{ must}_k \)

We proceed by cases on \( M \),

\textbf{case } M = \phi: \text{ then by Proposition 6.15}

\( t[\text{CELL0} \leftrightarrow \text{SORT}^{2[n-1]}(\phi)] \text{ must}_{2k} \)

Now since \( \text{must}_{2k} \subseteq \text{must} \) we can apply the inductive
hypothesis and Proposition 6.19 to get
\[ t[\text{WS}^n(M)] \text{ must} \]

**case** \( M = \{ m \} \): then by Proposition 6.15
\[ t[\text{CELL1}(m) \leftrightarrow \text{SORT}^{2(n-1)}(\emptyset)] \text{ must}_{2^k} \]
so by the inductive hypothesis and Proposition 6.19
\[ t[\text{WS}^n(M)] \text{ must} \]

**case** \( 2 \leq |M| \leq 2n, m = \min(M) \): then by Proposition 6.15
\[ \forall v \in M \setminus \{ m \}. \ t[\text{CELL2}(m,v) \leftrightarrow \text{SORT}^{2(n-1)}(M \setminus \{ m,v \})] \text{ must}_{2^k} \]
so by the inductive hypothesis and Proposition 6.19
\[ t[\text{WS}^n(M)] \text{ must} \]

\[ \Box \]

**Corollary 6.4**
Suppose \( n \geq 0, M \in \text{multisets(Nat)}, |M| \leq 2n, \) then
\[ \text{WS}^n(M) \text{ implements } \text{SORT}^{2n}(M) \]

\[ \Box \]
Conclusion

This thesis has considered two new theories of translation correctness. The contribution of chapters 2 and 3 was the development of a bisimulation theory of translation correctness and a demonstration that it can be applied to non-trivial examples if a sufficiently modular approach is taken. The particular example chosen was an improved version of Li's translation from CSP to CCS, for which we proved result-goodness. The definition of 'result', as the relation of 'goodness', could certainly be strengthened to include notions of convergence and deadlock. Since we already have the bisimulation (which enumerates all the possible states a (CSP or CCS) process can get into) most of the work is already done, it just remains to prove the bisimulation is a subset of the new relation (which presumably is a subset of 'result'). In this way Li's 'correctness' result might be achieved without resort to 'adequacy', but with additional results about the behavioural relationships in the translation.

The main contribution of chapter 4 was the development of the testing equivalence of [de Nicola and Hennessy 84] into a general theory of translation. By contrasting it with a bisimulation theory of translation we were able to attempt a critical comparison of their approaches, wherein a recurrent theme was the problem of trading off nice proof techniques for more interesting translations.

A careful analysis of the testing approach yielded a new notion of correctness called implementation which can be used to capture a relationship of refinement between a specification and the program
that implements it. Later, in chapter 6, we showed through the
example of a simple sorting machine in CCS how implementation
could be used as a specification tool, and demonstrated a first
attempt at a proof technique in CCS.

Much more work needs to be done on implementation before it can be
added to the armoury of techniques and concepts commonly used in
CCS. We have shown how implementation can define a pre-order over
CCS terms but have not given it any axiomatic definition. It
seems such a definition should be readily constructable from the
work already done on similar pre-orders in [de Nicola and Hennessy
84].

Another direction of research on implementation would be to try
giving it a denotational or predicate-transformer interpretation
via the work of [Smyth 83] and then comparing it with Park's
notion of implementation by determinate 'slices' [Park 81]. This
might also enable a comparison, in the denotational style, of the
commuting diagram approach to translation correctness [Morris 73]
with the testing approach presented here.

Chapter 5 demonstrated some advantages of the testing theory over
other theories, most notably its ability to cope with radically
different source and target systems. The proof technique employed
relied on all computations being finite, though it was shown how
this demand could be relaxed.

It is unclear how general this approach is and it would certainly
benefit by application to more examples. However, any search for
a better proof technique may be seriously hampered by two
problems. Firstly, to cope with very different source and target systems the technique must be stated very abstractly and apply as little constraint as possible; we saw how the constraints of the bisimulation theory prevented its effective application to this example. The second problem lies in the sheer volume of work required to prove even a simple translation, some form of automation is highly desirable if more realistic examples are to be attempted.

In chapter 6 we saw how the CCS operator @ could be used both by the implementer, to (under-) determine a distribution of values over the weavesort machine, and the specifier, to express indifference as to which value is lost upon overflow. As a result of the indeterminacy introduced by @ the relationship established between specification and implementation was implements rather than completely-implements or =. Thus the proof had two parts, a may part and a must part.

Proving the may part involved introducing the notion of 'simulation' which has strong similarities to half the bisimulation relator F. It seems quite likely that some weakening of the definition of simulates is desirable, possibly introducing a transition relation \( \Rightarrow \) which is to \( \rightarrow \) as \( \Rightarrow \) is to \( \rightarrow \). However, it should be noted that the definition of simulates as given exploits the asymmetric relationship of specification and implementation, essentially assuming the specification has no 'silent' internal actions. In many examples this will be so since the specification is often just an abstract requirement of the external behaviour, which might require no reference to 'silent'
action.

The proof of the must part depended (in its inductive step) on being able to rewrite a test on a larger object as a larger test on a (no larger) object. This worked because we could pull a small piece (a single cell) off the (larger) object and into the test. It was very important that the piece was small, for then we could essentially (in the ranking) enumerate its possible states. A thorough investigation is required before this technique can be claimed to have any general applicability.
References


Notation

Sets

\[ x: P(x) \] intensional definition

\[ [x_1, x_2, \ldots] \] extensional definition

\( x \in S \) membership

\( x \notin S \) non-membership

\( S_1 \subseteq S_2 \) subset

\( S_1 + S_2 \) union

\( S_1 - S_2 \) difference i.e. \( \{x: x \in S_1 \text{ and } x \notin S_2\}\)

\( S_1 \cap S_2 \) intersection

\( \sum_{i \in I} S_i \) indexed union

\( S_1 \times S_2 \) cartesian product

\( \mathcal{P}(S) \) powerset

\( \mathbb{N} \) \([0, 1, 2, \ldots]\)

Multisets

\[ x: P(x) \] intensional definition

\[ [x_1, x_2, \ldots] \] extensional definition

\( x \in M \) membership

\( x \notin M \) non-membership

\( M_1 \subseteq M_2 \) submultiset

\( M_1 ++ M_2 \) multiset union (e.g. \([1, 2] ++ [2, 3] = [1, 2, 2, 3]\))

\( M_1 -- M_2 \) multiset difference (e.g. \([1, 2, 2, 2] -- [2, 2] = [1, 2]\))

(Defined only if \( M_2 \subseteq M_1 \))

Sequences

\( \langle c_k \rangle_K \) the sequence \( c_{k_1}, c_{k_2}, \ldots \) where \( K = \{k_1, k_2, \ldots\} \) is a countable set.
\[ \langle c \rangle_K \] as above.

Relations

relation the relation 'relation' does not hold.

Not interested

Due to the large syntactic size of many terms occurring in this thesis it is useful to substitute '_' for some terms whose exact form is not of immediate interest but is easily deducable from the surrounding context.
Index of Definitions

The format adopted here is to give first the page number at which the definition occurred, second the number of the relevant definition, third the item(s) defined, and lastly any relevant additional information. The entries are organised first by chapters and then by the smallest subsection containing the definition, given in capitals.

1. Labelled transition systems and operational semantics

THE GENERAL APPROACH

p25 _ Antecedent

Consequent

LABELLED TRANSITION SYSTEMS

p28 1.1 Labelled transition systems L,
    config(L), trans(L), labels(L), term(L),
    c -A>N c',

p29 1.2 A^+, A^*, \phi

p29 1.3 (w-)derivative, derivation,
    c ->, c-\rightarrow, c -A>N, c -A>N, c -A>N,
    c -A>N*, c -A>N, c' -A>N, c'
    comp(c), <o_k>N, c_k

A COMMON CORE OF DEFINITIONS

p31 _ Vars, Exp, Val, Bexp,
    tt, ff

p32 _ States

p32 1.4 eval, beval

CCS

p35 _ Act

Action names A,B,C.
Proc
channels, line names,
CCS
Act
AD
Ren
Terms

\[ \sum_{k \in K} p_k \]
\[ \Pi_{k \in K} p_k \]
\[ p \setminus x, \ p[A'/A], \ \text{if } b \ \text{then } p \]

\[ \text{FV}(p), \ \text{FP}(p), \ \text{FL}(p), \ \text{sort}(p) \]

\[ \text{definition set} \]
\[ p(x_1, x_2, \ldots, x_n) \leq p \]

guardedly well-defined

\[ \vdash \alpha \]

2. A bisimulation theory of translation correctness

REVIEW

\[ \text{correctness, adequacy (Li's theory)} \]

BISIMULATIONS

\[ 2.1 \ ] \text{bisimulation, } F(R), \ c_1 \cong c_2 \]

\[ [n] \xrightarrow{\beta} [m] \]

deadlocked, terminal, spinning

TRANSLATION CORRECTNESS

\[ 2.2 \ ] \text{translation, source, target} \]

\[ 2.4 \ ] \text{good} \]

\[ 2.5 \ ] \text{P-bisimulation} \]

\[ 2.6 \ ] \text{P-good} \]
3. An example translation in the bisimulation theory.

SYNTAX

- Plab: Process labels P, Q, R.
- Pten: Pattern symbols W.
- Gcom: Guarded commands gc.
- Com: Commands gc.
- alternative, conditional, repetitive, input, output

STATIC SEMANTICS

- Syn: Syntax \( \omega \).
- RV(\omega), WV(\omega), FPL(\omega)
- Bool
- FV(\omega)
- separated
- \( \vdash_{\text{Com}} \) \( \vdash_{\text{Gcom}} \)
- Alab
- *
- unknown process name.

THE TRANSLATION

- strong guarding, weak guarding

THE TROUBLESOME VARIABLES
AUXILLIARY CCS DEFINITIONS

\[ [s], \text{RS}(X)[p] \]

p with s, p with s:X, p \{ q, a \rightarrow p \}

p bef q, p par q, p parD q, p parA q

THE TRANSLATION

\[ L_X \]

ren(R), sco(R,c)

\[ L_{\text{sem}} \]

THE STATEMENT OF CORRECTNESS

atf, tf

null, nuld, nula

result

THE PROOF OF CORRECTNESS

\[ *, *_{W1,W2} \] Merging states

commands(gc)

ITER, iter

THE BISIMULATION

END

\[ R_X \]

Actimp

swv, srv, sfv

separated(p,q)


EXPERIMENT SYSTEMS

experiment system \( E_X \)

exp(\( E_X \)), trans(\( E_X \)), succ(\( E_X \))

T[p], t[p]
EXP[CCS]

TRANSLATIONS

p129  4.2  translation

GAMES OBSERVERS PLAY

p135  4.3  \text{success}(e^k_k)

p135  4.4  \text{result}, \text{Res}, \text{Resets}

p136  4.5  \text{SPEC}

p137  4.6  implements

COMPARING RESULTS

p138  4.7  \text{must}, \text{mustnot}

p139  4.8  \text{may}

COMPLETE IMPLEMENTATION

p140  4.9  \text{completely-implements}

TRANSLATION CORRECTNESS

p141  4.10  \text{implementation}

\text{complete-implementation}

IMPLEMENTATIONS IN CCS

p143  4.11  \text{implements}

\text{completely-implements}

5. An example in the testing theory

p155  _  Acom

Com

Prog

\text{Atomic commands ac.}

\text{Commands c.}

\text{Programs p.}

SOURCE LANGUAGE STATIC SEMANTICS

p156  5.1  \text{RV}(c), \text{WV}(c), \text{FV}(c)

p157  5.2  \text{non-inter}(c_1, c_2)

SOURCE LANGUAGE OPERATIONAL SEMANTICS
5.3 Source

THE TARGET LANGUAGE

queue, Queues(A),
mset(q), set(q),
empty,
a.q, q.a

5.4 queue, Queues(A),
mset(q), set(q),
empty,
a.q, q.a

5.5 communication state
Q(R,S,send), Q(R,S,sent), Q(t)

5.6 initQ

THE TARGET LANGUAGE SYNTAX

Acom2 Atomic commands ar.
Com2 Commands r.
Prog2 Programs pr.

5.7 RV(r), WV(r), FV(r)

THE TARGET LANGUAGE STATIC SEMANTICS

5.8 Target

THE TRANSLATION $[.]_1$

$[.]_2$ AND THE EXPERIMENT SYSTEMS SEXP, TEXP

5.10 OBJ Processes obj.

TST Tests tst.
tst[objc]

SEXp

5.11 TOBJ Processes tobj.

TTST Tests ttst.
ttst[tobj]

TEXp
PROVING CORRECTNESS OF THE TRANSLATION $[.]_2$

WHY THE BISIMULATION APPROACH WON'T WORK

THE FOUR CONDITIONS

PROVING THE FOUR CONDITIONS FOR $[.]_2$

ALTERNATIVE RESULTS

$[.]_2$

computationally finite

success terminating

resultset

result

$\mathbf{cl}, \mathbf{c2}, \mathbf{c3}, \mathbf{c4}$

INTER_t, inter_t, INTER, IMAGE

waiting, awaking, assigners, offering

senders, receivers, offerers

do-one, run

$[e]$

in-send, in-sending, in-sent

clean

cons

inv

match

(Source) program context $\mathbf{Context \ pc.}$

(Target) program context $\mathbf{Context \ uc.}$

$\mathbf{c1}, \mathbf{c2}, \mathbf{c3}, \mathbf{c4}, \mathbf{c5}$

$\mathbf{C5.1}$

$\mathbf{C1.1}, \mathbf{C5.2}$
6. Weavesort; an example implementation.

ADDING INDETERMINACY

p219 6.1 $\emptyset$

p220 6.1 ds
determinate, determinate term

p221 6.2 $-a>$

convention on computations of length 0

p224 $\prod_{k \in K} q_k$

p225 6.3 $-a>$

EXTENDING THE SET OF TESTS

p226 6.4 T Tests t.

p227 6.5 P Objects p.

p227 6.6 Test$_a$, Did$_a$

p230 6.7 NEXP[CCS]

THE DESCRIPTION AND SPECIFICATION IN CCS

p236 6.8 $\leftrightarrow$

p240 $\text{WS}^n(M)$

p241 $\text{SORT}^2n(M)$

PROVING THE MAY PART

p246 6.9 simulation, simulates

p249 6.10 $S_n$

PROVING PART 3

p258 6.11 must$_k$

PROVING PART 2

p260 6.12 ranking

p261 6.13 $R_i$